HP-SEE

Introduction to Parallel Computing: The Message Passing and Shared Memory

www.hp-see.eu

Todor Gurov
Assoc. Professor
IICT-BAS
gurov@bas.bg

HP-SEE
High-Performance Computing Infrastructure for South East Europe's Research Communities

The HP-SEE initiative is co-funded by the European Commission under the FP7 Research Infrastructures contract no. 261499

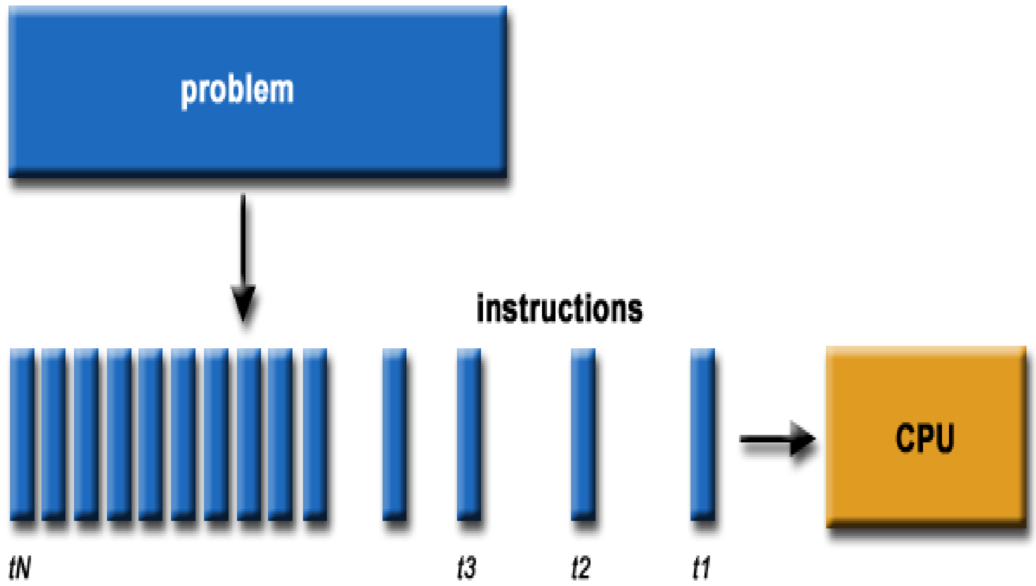
OTLINE

HP-SEE
High-Performance Computing Infrastructure for South East Europe's Research Communities

- Overview of parallel computing
  - What is parallel computing?
  - Why/who use a parallel computing?
  - Concepts and technology/ parallel terminology
  - Communication/Load Balancing/Granuarity
- Parallel Computer Memory Architectures
- Parallel programming models
- Parallel programming paradigms
  - Message passing (MPI)
  - Shared memory (OpenMP)
- General Consideration and Conclusion

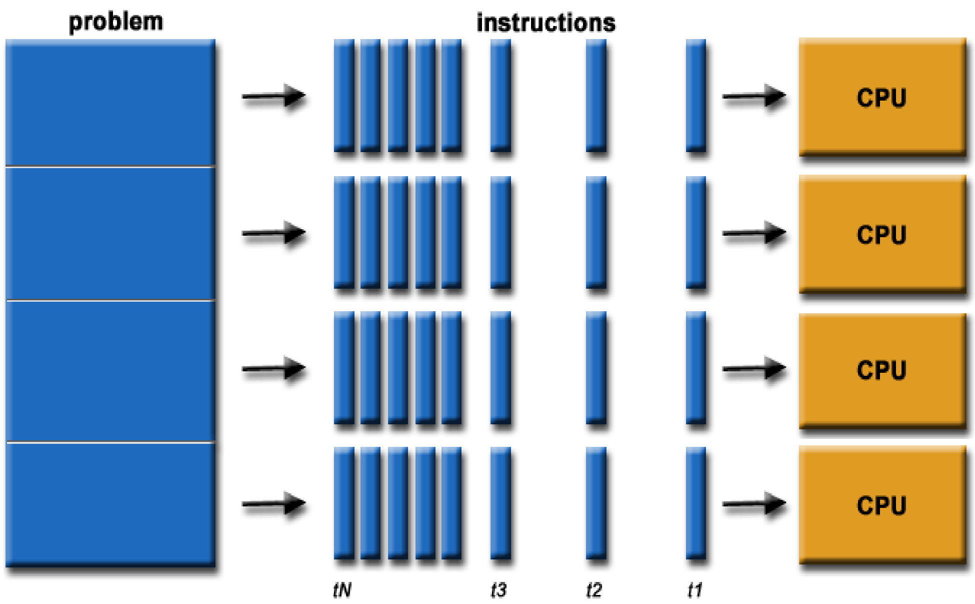
Overview Parallel Computing

- Traditionally, software has been written for ***serial*** computation:
  - To be run on a single computer having a single Central Processing Unit (CPU);
  - A problem is broken into a discrete series of instructions.
  - Instructions are executed one after another.
  - Only one instruction may execute at any moment in time.



What is Parallel Computing?

- ***parallel computing*** is the simultaneous use of multiple compute resources to solve a computational problem.
  - To be run using multiple CPUs
  - A problem is broken into discrete parts that can be solved concurrently
  - Each part is further broken down to a series of instructions
- The compute resources can include:
  - A single computer with multiple processors;
  - An arbitrary number of computers connected by a network;
  - A combination of both.
- The computational problem usually demonstrates characteristics such as the ability to be:
  - Broken apart into discrete pieces of work that can be solved simultaneously;
  - Execute multiple program instructions at any moment in time;
  - Solved in less time with multiple compute resources than with a single compute resource.

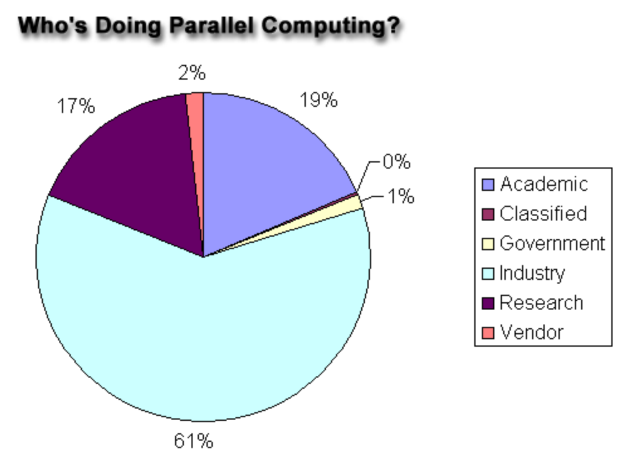


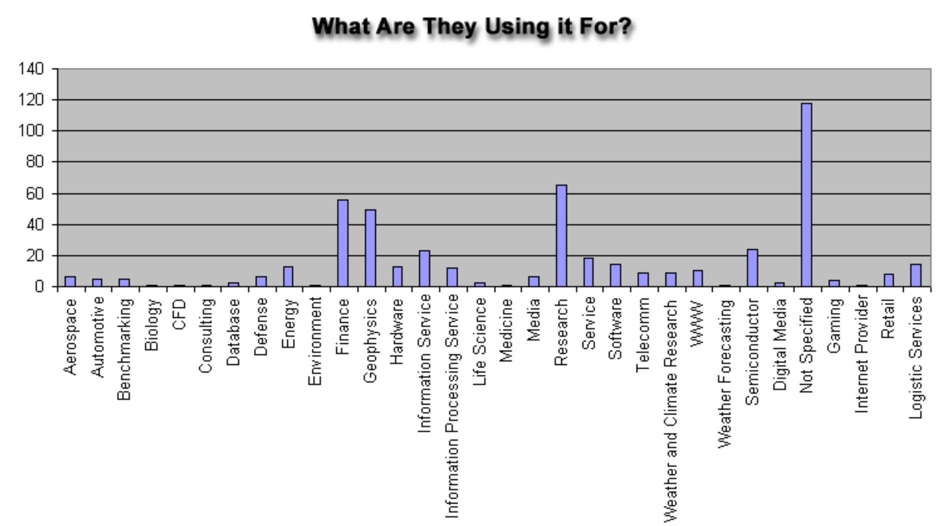
Why Use Parallel Computing?

- Save time and/or money: Parallel clusters can be built from cheap, commodity components.
- Solve larger problems**:**
  - "Grand Challenge" problems (en.wikipedia.org/wiki/Grand_Challenge) requiring PetaFLOPS and PetaBytes of computing resources.
  - Web search engines/databases processing millions of transactions per second.
- Provide concurrency
  - Multiple computing resources can be doing many things simultaneously
- Use of non-local resources (EGEE/EGI infrastructure)
- Limits to serial computing
  - Transmission speeds
  - Limits to miniaturization
  - Economic limitations

Who use Parallel Computing

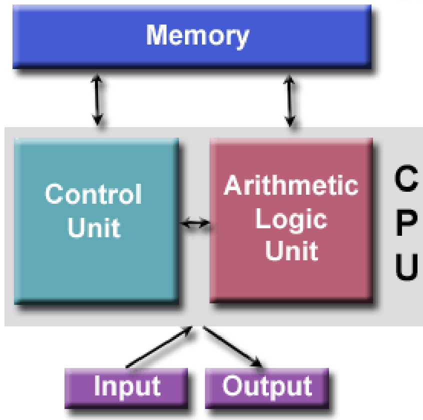
- Statistics on parallel computing users - from top500.org
- Sectors on the Figure may overlap





Concepts and Technology

- **von Neumann Architecture**
  - Named after the Hungarian mathematician John von Neumann who first authored the general requirements for an electronic computer in his 1945 papers.
  - Since then, virtually all computers have followed this basic design.



- **Flynn's Classical Taxonomy**
  - This classification is widely used, in use since 1966.
  - Flynn's taxonomy distinguishes multi-processor computer architectures according to how they can be classified along the two independent dimensions of ***Instruction*** and ***Data***. Each of these dimensions can have only one of two possible states: ***Single*** or ***Multiple***.

S I S D **Single Instruction, Single Data**	**S I M D** **Single Instruction, Multiple Data**
M I S D **Multiple Instruction, Single Data**	**M I M D** **Multiple Instruction, Multiple Data**

Some General Parallel Terminology

HP-SEE
High-Performance Computing Infrastructure for South East Europe's Research Communities

- Task /Parallel Task /Serial or parallel Execution
- Pipelining/Shared Memory /Distributed Memory
- Symmetric Multi-Processor (SMP)
- Communications/Synchronization
- Granularity (coarse, fine)
- Observed Speedup /Parallel Overhead
- Massively Parallel /Embarrassingly Parallel
- Scalability /Latency
- Multi-core Processors /Cluster Computing
- Supercomputing / High Performance Computing

Parallel Computer Memory Architectures

- Shared Memory
- Distributed Memory
- Hybrid Distributed-Shared Memory

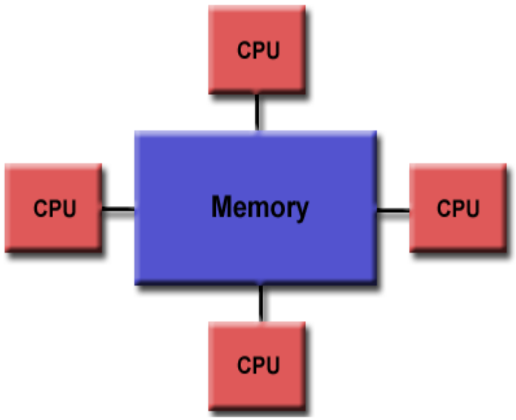
Machine architecture dictates the programming model

- Parallel Programming Models
  - Message Passing Model
  - Shared Memory Model
  - Threads Model
  - Data Parallel Model
  - Other Models
    - Hybrid:
    - Single Program Multiple Data (SPMD):
    - Multiple Program Multiple Data (MPMD):

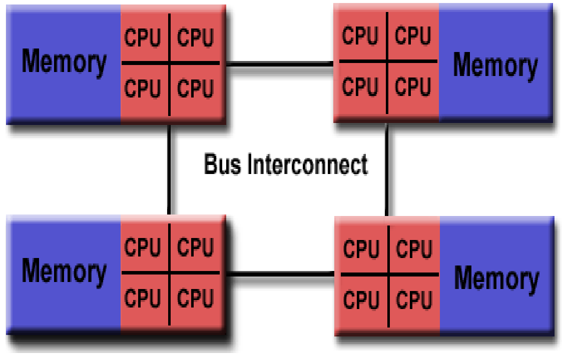
Shared Memory architecture

HP-SEE
High-Performance Computing Infrastructure
for South East Europe's Research Communities

- General Characteristics
  - ability for all processors to access all memory as global address space.
  - Multiple processors operate independently but share the same memory resources.
  - Changes in a memory location effected by one processor are visible to all other processors.
- Uniform Memory Access (UMA):
  - Most commonly represented today by Symmetric Multiprocessor (SMP) machines
  - Identical processors
- Non-Uniform Memory Access (NUMA):
  - Often made by physically linking two or more SMPs
  - Not all processors have equal access time to all memories
- Advantages:
  - Global address space provides a user-friendly programming perspective to memory
  - Data sharing between tasks is both fast and uniform due to the proximity of memory to CPUs
- Disadvantages:
  - Primary disadvantage is the lack of scalability between memory and CPUs
  - Expense: it becomes difficult and expensive to design and produce shared memory machines with ever increasing numbers of processors.



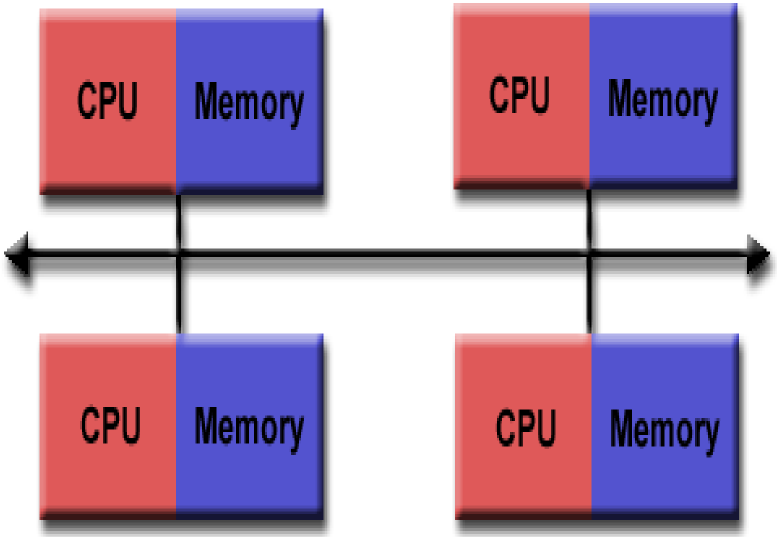
Shared Memory (UMA)



Shared Memory (NUMA)

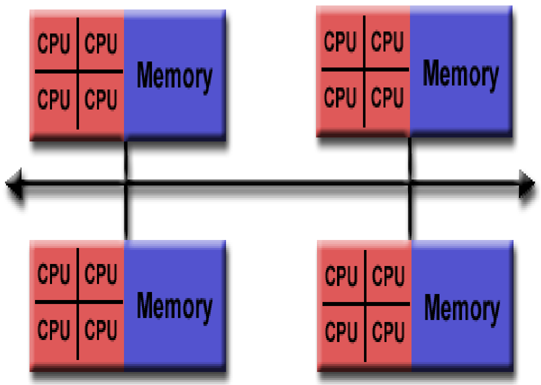
Distributed Memory architecture

- **General Characteristics:**
  - Distributed memory systems require a communication network to connect inter-processor memory
  - Processors have their own local memory. Memory addresses in one processor do not map to another processor
  - Synchronization between tasks is likewise the programmer's responsibility.
  - The network "fabric" used for data transfer varies widely, though it can be as simple as Ethernet.
- **Advantages:**
  - Memory is scalable with number of processors. Increase the number of processors and the size of memory increases proportionately.
  - Each processor can rapidly access its own memory without interference and without the overhead incurred with trying to maintain cache coherency.
  - Cost effectiveness: can use commodity, off-the-shelf processors and networking.
- **Disadvantages:**
  - The programmer is responsible for many of the details associated with data communication between processors.
  - It may be difficult to map existing data structures, based on global memory, to this memory organization.
  - Non-uniform memory access (NUMA) times



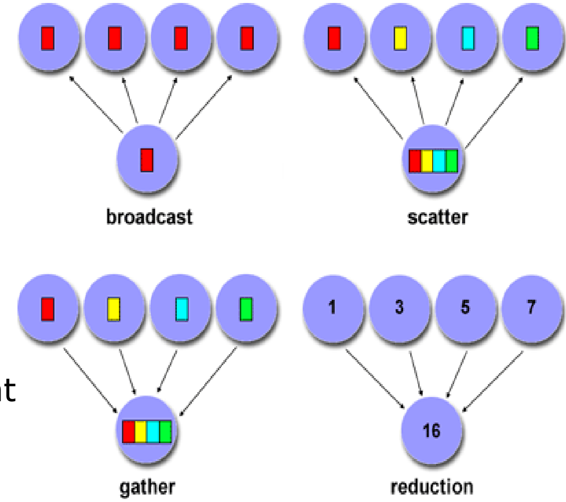
Hybrid Distributed-Shared Memory architecture

- The largest and fastest computers in the world today employ both shared and distributed memory architectures.
- The shared memory component is usually a cache coherent SMP machine. Processors on a given SMP can address that machine's memory as global.
- The distributed memory component is the networking of multiple SMPs. SMPs know only about their own memory - not the memory on another SMP. Therefore, network communications are required to move data from one SMP to another.
- Current trends seem to indicate that this type of memory architecture will continue to prevail and increase at the high end of computing for the foreseeable future.
- Advantages and Disadvantages: whatever is common to both shared and distributed memory architectures.



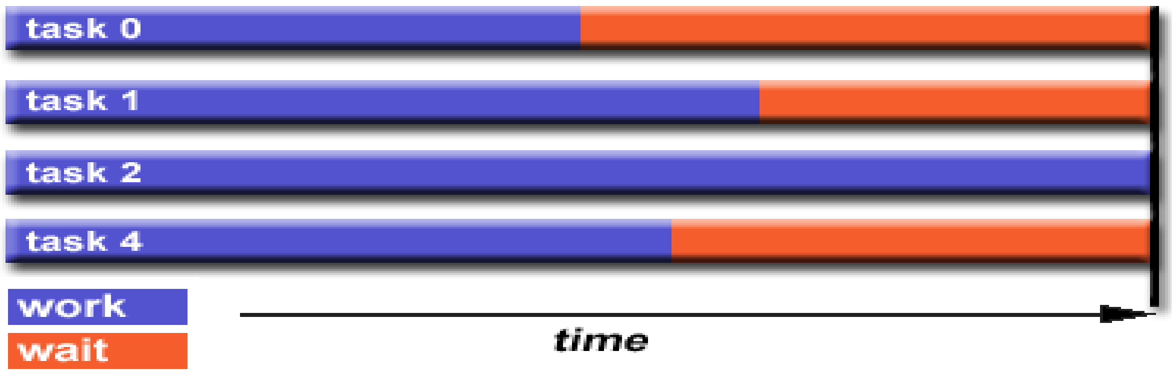
Communications

- When you DON'T need communications
  - Some types of problems can be decomposed and executed in parallel with virtually no need for tasks to share data.
  - These types of problems are often called ***embarrassingly parallel*** because they are so straight-forward.
- When you DO need communications
  - Most parallel applications are not quite so simple, and do require tasks to share data with each other.
- Factor to consider
  - Cost of communications
  - Latency vs. Bandwidth
    - ***latency*** is the time it takes to send a minimal (0 byte) message from point A to point B. Commonly expressed as microseconds.
    - ***bandwidth*** is the amount of data that can be communicated per unit of time. Commonly expressed as megabytes/sec or gigabytes/sec.
  - Visibility of communications
  - Synchronous vs. asynchronous communications
  - Scope of communications
  - Efficiency of communications
  - Overhead and Complexity



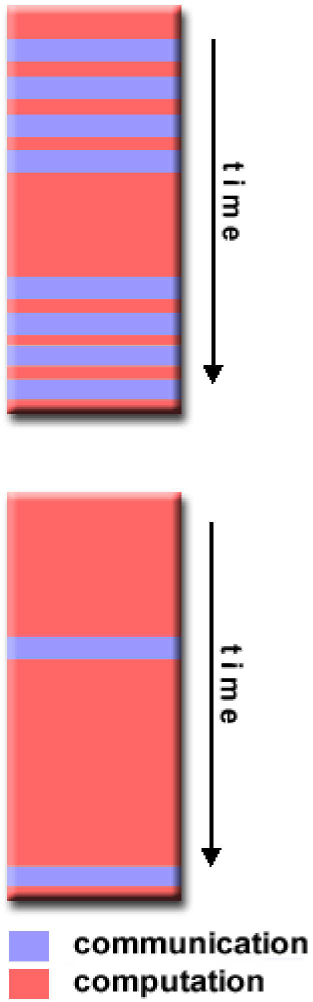
Load Balancing

- Load balancing refers to the practice of distributing work among tasks so that ***all*** tasks are kept busy ***all*** of the time. It can be considered a minimization of task idle time.
- Load balancing is important to parallel programs for performance reasons. For example, if all tasks are subject to a barrier synchronization point, the slowest task will determine the overall performance.



Granularity

- Computation / Communication Ratio:
- Fine-grain Parallelism:
  - Relatively small amounts of computational work are done between communication events
  - Low computation to communication ratio
  - Facilitates load balancing
- Coarse-grain Parallelism:
  - Relatively large amounts of computational work are done between communication/synchronization events
  - High computation to communication ratio
  - Implies more opportunity for performance increase
  - Harder to load balance efficiently
- Which is Best?
  - The most efficient granularity is dependent on the algorithm and the hardware environment in which it runs.
  - In most cases the overhead associated with communications and synchronization is high relative to execution speed so it is advantageous to have coarse granularity.
  - Fine-grain parallelism can help reduce overheads due to load imbalance.



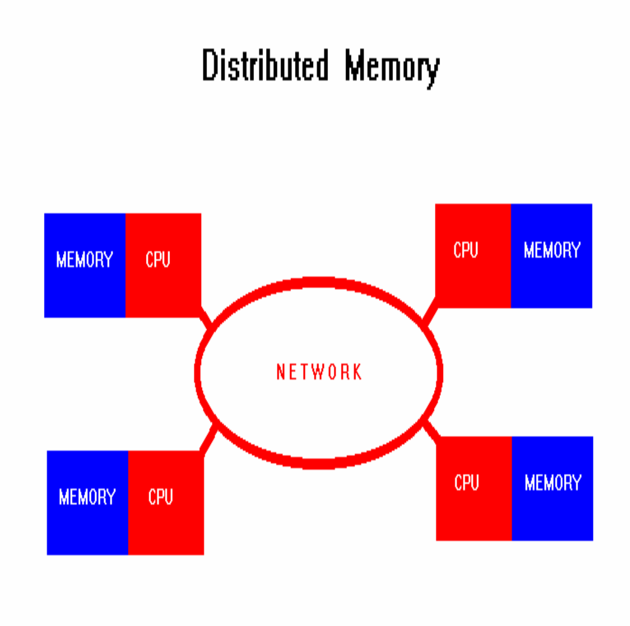
Parallel programming models

HP-SEE
High-Performance Computing Infrastructure for South East Europe's Research Communities

- Distributed memory systems (I)
  - Programmer uses “Message Passing” in order to sync
  - processes and share data among them
  - Message passing libraries
    - MPI
    - PVM
- Shared memory systems (II)
  - Thread based programming approach
  - Compiler directives (openMP)
  - Message passing may also be used
- Programming models on hybrid architectures / Hybrid memory systems (III)

(I) Parallel Programming Models: Distributed Memory

- Each processing element P has its own local memory hierarchy
- Local memory is not remotely accessible by other processing elements
- Processing elements are connected by means of a special network
- Architecture dictates:
  - Data and computational load must be explicitly distributed by the programmer
  - Communication (data exchange) is achieved by messages
  - Probably the oldest paradigm. Several variants: PVM (Parallel Virtual
  - Machine), MPI (Message Passing Interface - ultimate winner)



Message Passing Interface

- Message passing model is a process which may be defined as a program counter and an address space
- Each process may have multiple threads sharing the same address space
- Message Passing is used for communication among processes
  - synchronization
  - data movement between address spaces
- MPI is a message passing library specification
  - not a language or compiler specification
  - no specific implementation
- Source code portability
  - SMPs
  - clusters
  - heterogenous networks

Types of communication

- Initialization, Finalization and Synchronization calls
- Point-to-Point calls
  - data movement
- Collective calls
  - data movement
  - reduction operations
  - synchronization

What is need to know

- MPI_Init
- MPI_Comm_size (get number of processes)
- MPI_Comm_rank (gets a rank value assigned to each process)
- MPI_Send (cooperative point-to-point callused to send data to receiver)
- MPI_Recv (cooperative point-to-point call used to receive data from sender)
- MPI_Finalize

“HelloWord!” using MPI

"HelloWorld!" program that illustrates the basic MPI calls necessary to startup and end an MPI program.

```
#include <stdio.h>
#include "mpi.h"
int main(int argc, char **argv)
{        int me, nprocs, namelen;
        char processor_name[MPI_MAX_PROCESSOR_NAME];
        MPI_Init(&argc, &argv);
        MPI_Comm_size(MPI_COMM_WORLD, &nprocs);
        MPI_Comm_rank(MPI_COMM_WORLD, &me);
        MPI_Get_processor_name(processor_name, &namelen);
     printf("HelloWorld!  I'm process %d of %d on %s\n", me, nprocs,
          processor_name);
        MPI_Finalize();
}
```

- Mpicc ./helloc -o hello.out
- mpirun –n 8 ./hello.out /* the executable *hello* run iteractivly on 8 CPUs */

Starting and exiting the MPI environment

- MPI_Init
  - C style: int MPI_Init(int *argc, char ***argv);
    - accepts argc and argv variables (main arguments)
  - F style: MPI_INIT (IERROR)
    - Almost all Fortran MPI library calls have an integer return code
  - Must be the **first MPI function called in a program**
- MPI_Finalize
  - C style: int MPI_Finalize();
  - F style: MPI_FINALIZE (IERROR)

Communicators

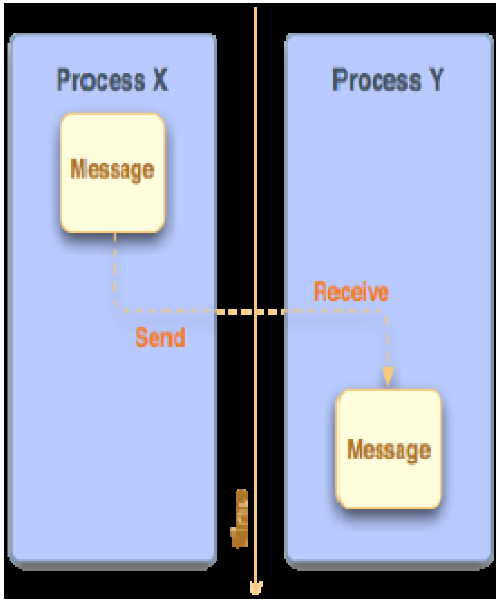
- All mpi specific communications take place with respect to a communicator
- Communicator: A collection of processes and a context
- MPI_COMM_WORLD is the predefined communicator of all processes
- Processes within a communicator are assigned a unique rank value

A few basic considerations

- How many processes are there?
  - (C) MPI_Comm_size(MPI_COMM_WORLD, &size);
  - (F) MPI_COMM_SIZE(MPI_COMM_WORLD, size, ierr)
- Which one is which?
  - (C) MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  - (F) MPI_COMM_RANK(MPI_COMM_WORLD, rank, ierr)
  - The rank number is between 0 and (size - 1)

Sending and receiving messages

- What is contained within a message?
  - message data
    - buffer
    - count
    - datatype
  - message envelope
    - source/destination rank
    - message tag (tags are used to discriminate among messages)
    - communicator

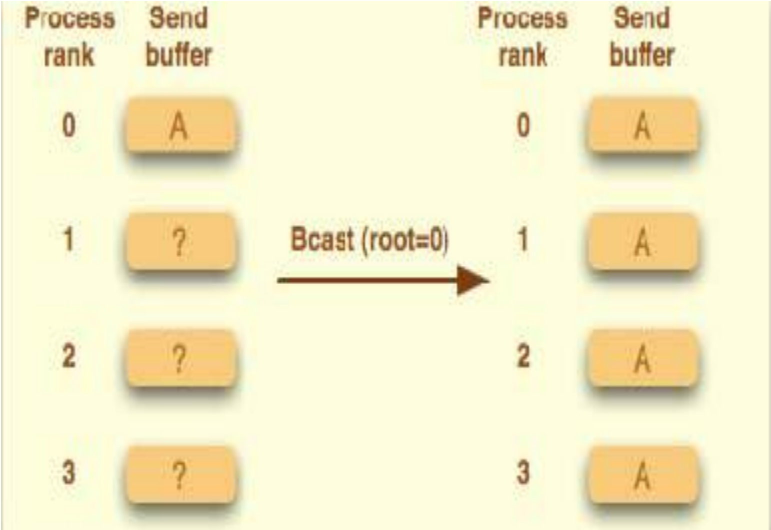


Collective communications

- All processes within the specified communicator participate
- All collective operations are blocking
- All processes must call the collective operation
- No message tags are used
- Three classes of collective communications
  - Data movement
  - Collective computation
  - Synchronization

Examples of collective operations

- **int MPI_Bcast(void** **buffer*, **int** *count*, **MPI_Datatype** *datatype*, **int** *root*, **MPI_Comm** *comm* **);**

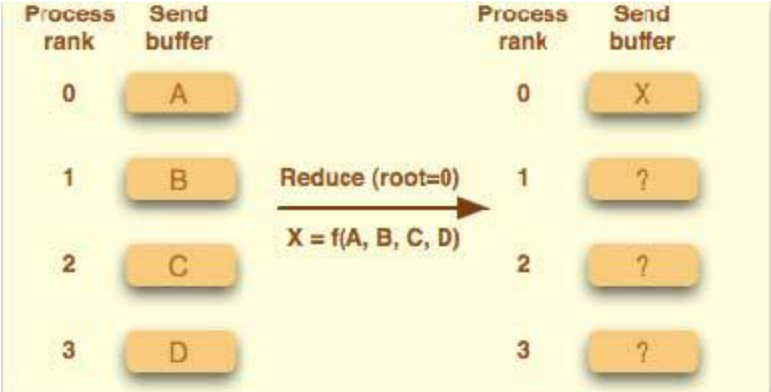


- **Parameters**
  - *buffer* [in/out] starting address of buffer (choice)
  - *count* [in] number of entries in buffer (integer)
  - *datatype* [in] data type of buffer (handle)
  - *root* [in] rank of broadcast root (integer)
  - *comm* [in] communicator (handle)

Examples of collective operations

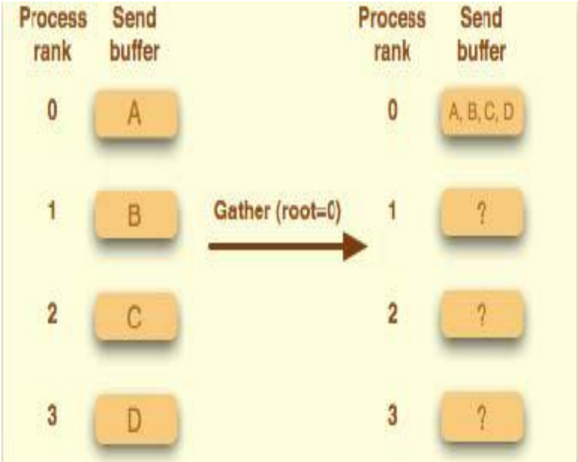
HP-SEE
High-Performance Computing Infrastructure for South East Europe's Research Communities

- **int MPI_Reduce(void** **sendbuf*, **void** **recvbuf*, **int** *count*, **MPI_Datatype** *datatype*, **MPI_Op** *op*, **int** *root*, **MPI_Comm** *comm* **);**
- **Parameters**
  - *sendbuf* [in] address of send buffer (choice)
  - *recvbuf* [out] address of receive buffer (choice, significant only at root)
  - *count* [in] number of elements in send buffer (integer)
  - *datatype* [in] data type of elements of send buffer (handle)
  - *op* [in] reduce operation (handle)
  - *root* [in] rank of root process (integer)
  - *comm* [in] communicator (handle)



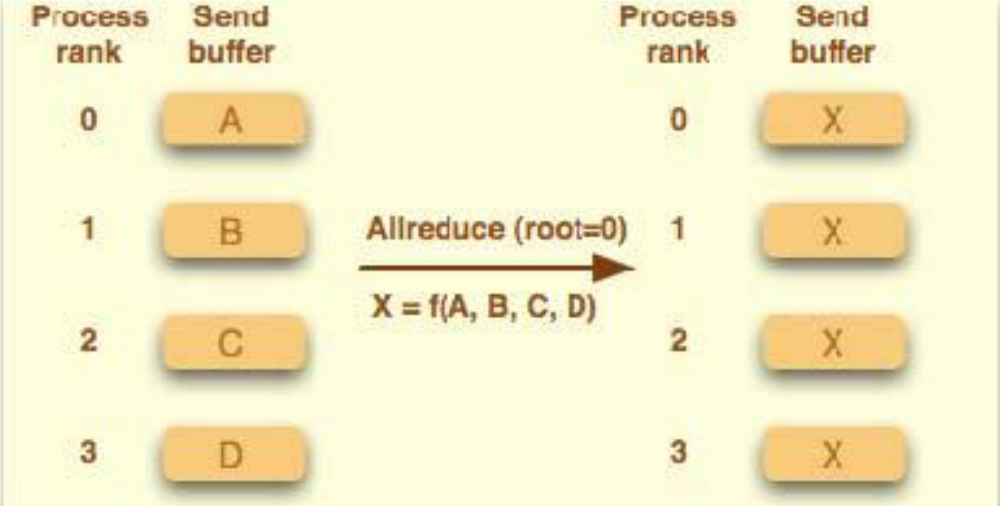
Examples of collective operations

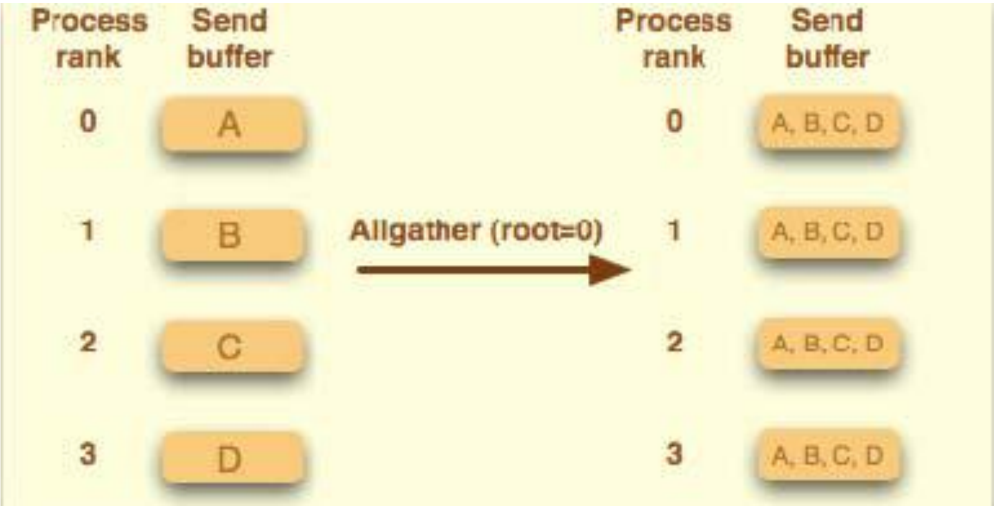
- **int MPI_Gather(void** **sendbuf*, **int** *sendcnt*, **MPI_Datatype** *sendtype*, **void** **recvbuf*, **int** *recvcnt*, **MPI_Datatype** *recvtype*, **int** *root*, **MPI_Comm** *comm* **);**
- **Parameters**
  - *sendbuf* [in] starting address of send buffer (choice)
  - *sendcount* [in] number of elements in send buffer (integer)
  - *sendtype* [in] data type of send buffer elements (handle)
  - *recvbuf* [out] address of receive buffer (choice, significant only at root)
  - *recvcount* [in] number of elements for any single receive (integer, significant only at root)
  - *recvtype* [in] data type of recv buffer elements (significant only at root) (handle)
  - *root* [in] rank of receiving process (integer)
  - *comm* [in] communicator (handle)



Examples of collective operations

- All MPI functions can be find on web:http://mpi.deino.net/mpi_functions/index.htm



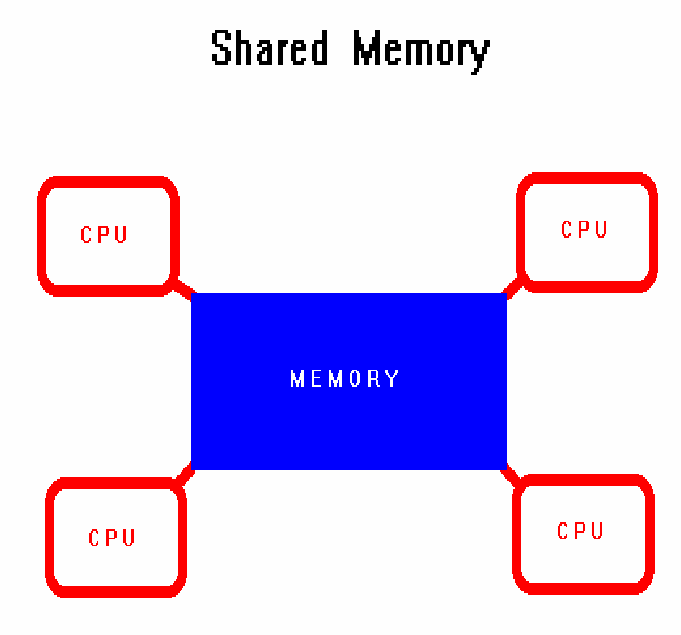


MPI Basic Datatypes

MPI Datatype	C datatype
MPI_CHAR	Signed char
MPI_SHORT	Signed short int
MPI_INT	Signed int
MPI_LONG	Signed long int
MPI_UNSIGNED_CHAR	Unsigned char
MPI_INSIGNED_SHORT	Unsigned short int
MPI_INSIGNED	Unsigned int
MPI_UNSIGNED_LONG	Unsigned long int
MPI_FLOAT	float
MPI_DOUBLE	double
MPI_LONG_DOUBLE	Long double
MPI_BYTE	
MPI_PACKED	

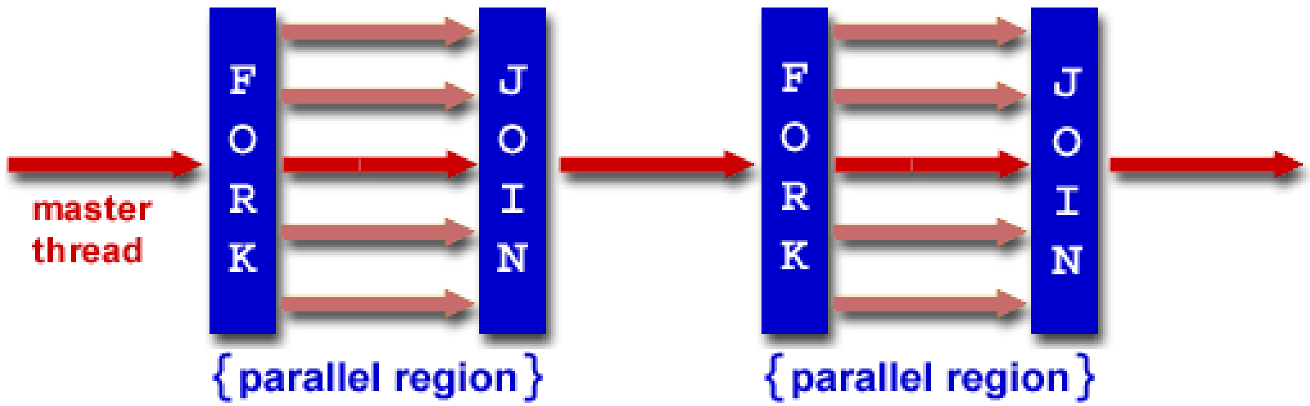
(II) Parallel Programming Models: Shared Memory

- Processing elements share memory (either directly or indirectly)
- Communication among processing elements can be achieved by *carefully*
- *reading and writing in main memory*
- Data and load distribution can be hidden from the programmer
- Messages can be implemented in memory as well (MPI)
- *Programming Model. OpenMP: Directives and Assertions*



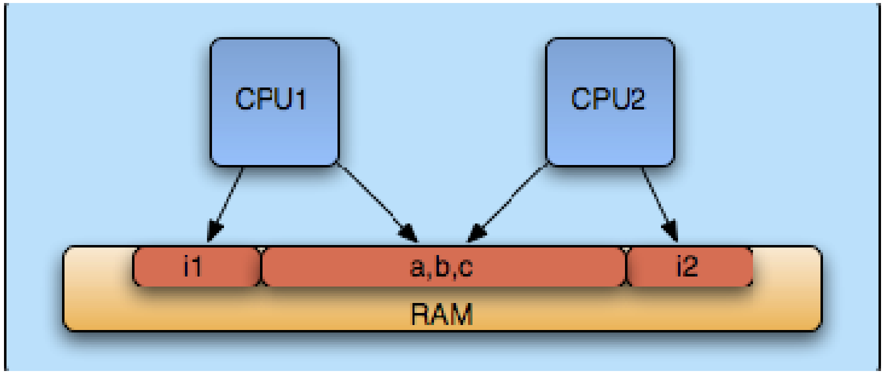
Thread parallel programming model (OpenMP)

- OpenMP is based on a fork - join model
  - Master – worker threads
- Use of directives and pragmas within source code



Memory issues

- Threads have access to the same address
- space
- Programmer needs to define
  - local data
  - shared data



Threads and thread teams

HP-SEE
High-Performance Computing Infrastructure for South East Europe's Research Communities

- A threads is a process – an instance of a program +its data
- Each thread can follow its own flow of control through a program
- Threads can share data with other threads, but also have private data.
- Threads communicate with each other via the shared data.
- A *thread team is a set of threads which cooperate on a task.*
- *The master thread is responsible for coordinating the team.*

Parallel region

- The parallel region is the basic parallel construct in OpenMP
- A parallel region dafines a section of a program
- Program begins execution on a single thread (the master thread).
- When the first parallel region is encountered,the master thread creates a team of threads
- Every thread executes the statements which are inside the parallel region.
- At the end of the parallel region, the master thread waits for the other threads to finish, and continues executing the next statements.

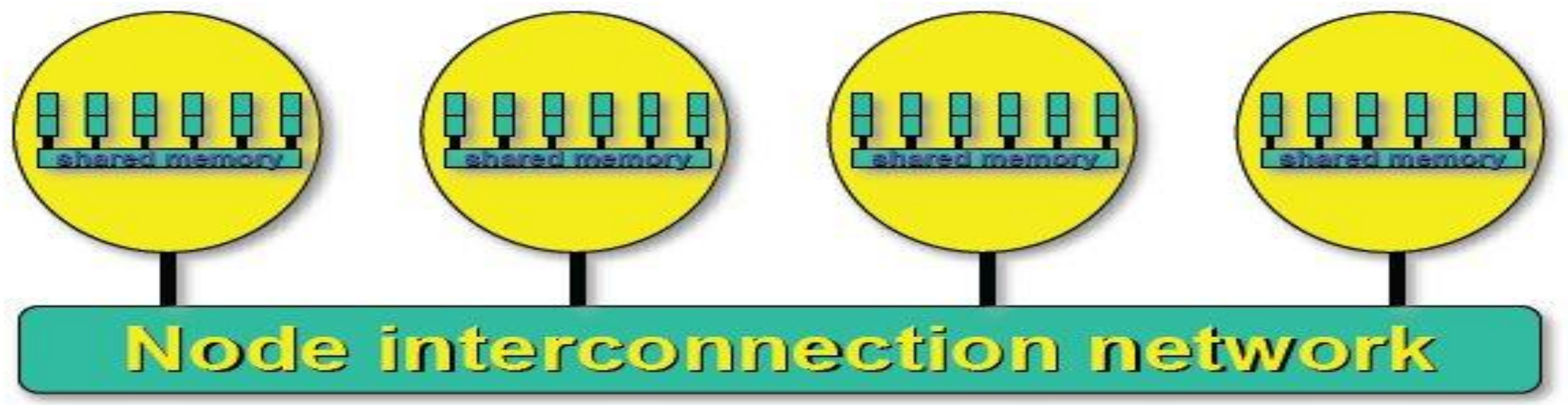
OpenMP Example: HelloWorld

HP-SEE
High-Performance Computing Infrastructure for South East Europe's Research Communities

```
#include <iostream>
#include (omp.h>
using homespace std;
main()
{
#pragma omp parallel
Printf(”hello from thread %d\n”,omp_get_thread_num());
}
```

(III) Programming models on hybrid architectures

- Pure MPI: Remember SMP supports MPI as well. Only MPI processes across the machine
- Hybrid MPI/OpenMP: OpenMP inside SMP nodes and MPI across the node interconnection network



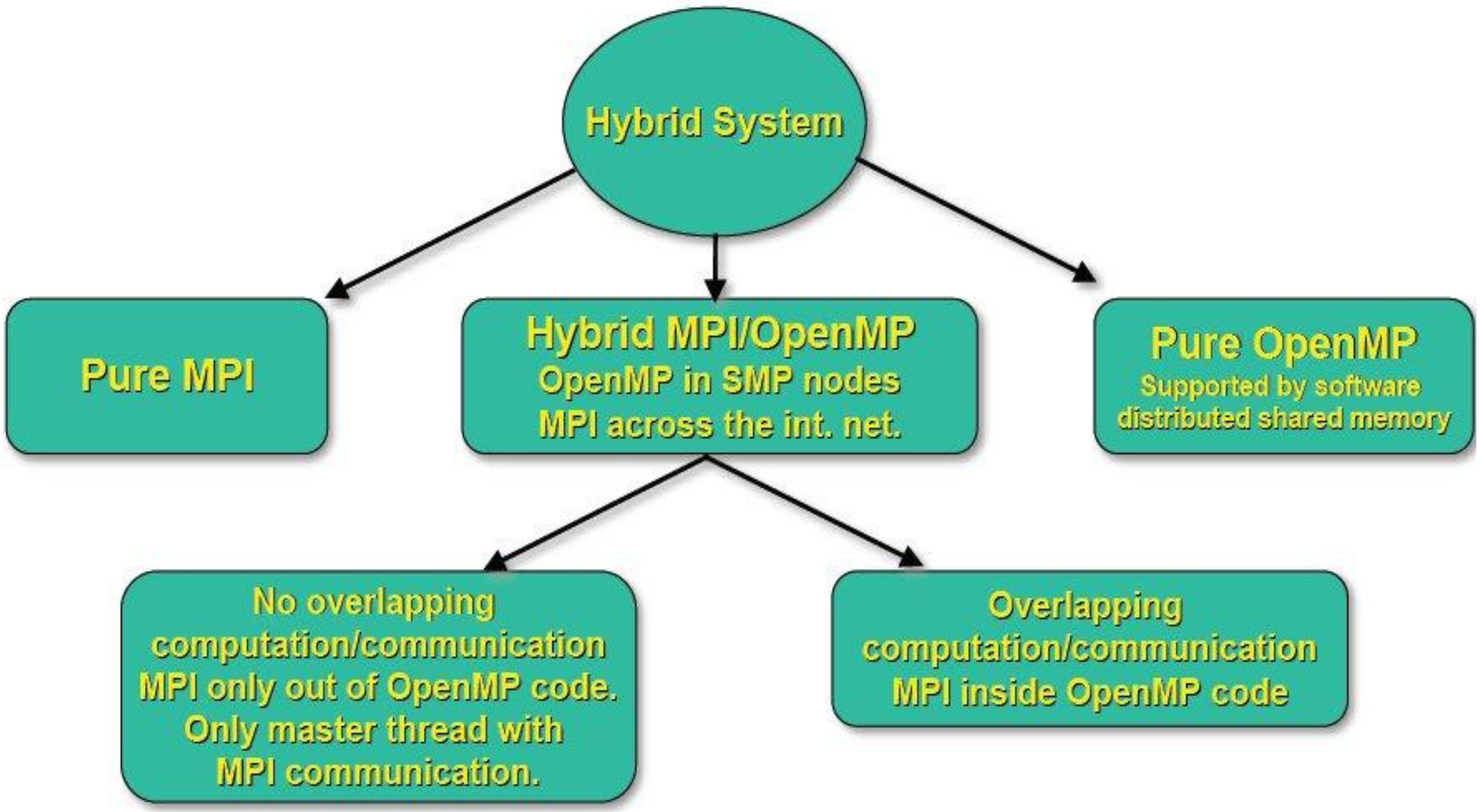
Hybrid Architectures: "Clusters" of SMPs

Hybrid Architectures: Examples

- IBM Blue Gene series
  - 1024 SMP nodes per rack
  - 4 cores per SMP node, 2-4 Gbytes per node
  - Hundreds of racks to reach 3PFlops

- IBM p6 575 (Huygens)
  - 16 dual core procs per node
  - 32 corés on SMP node, 128-256 Gbytes per node
  - 14 SMP nodes per rack, tens of racks

Hybrid systems programming hierarchy



General Consideration

- Compute everything every where
  - Use routines such as *Allreduce*
  - Perhaps the value only really needs to know on the master
- Often easiest to make P a compile-time constant
  - may not seem elegant but make coding much easier
  - Put definition in an include file
  - A clever *Makefile* can reduce the need for recompilation
    - Only recompile routines that define arrays rather than just use them
    - Pass array bounds as arguments to all other routines

Parallelisation and optimisation

- Some parallel approaches may be simple
  - But not necessary optimal for performance
  - Case study example is very simple due may be to 1D decompoition
    - But not particullary efficient for large Parallelism
- Some people write incredibly complicated code
  - Step back and ask: what do I actually want to do?
  - Is there an existing MPI routine or collective communications?
- Keep running your code
  - On a number of input data sets
  - With a range of MPI processes
- If scaling is poor
  - Find out parallel routines are the bottlenecks
  - Much easier with a separate comms library
- If performance is poor
  - Work on the serial code

Conclusion

- Run on a variety of machines
- Keep it simple
- Maintain a serial code
- Don’t assume all bugs are parallel bugs
- Find a debugger you like