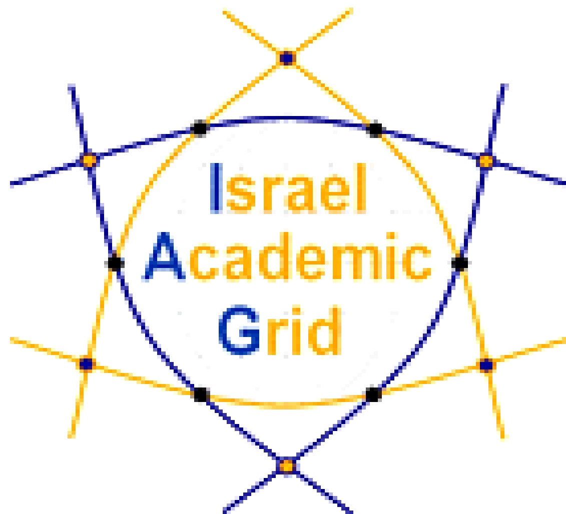
Enabling Grids for E-science in Europe

Israeli Grid Workshop, Ra'anana, Israel, Sep 2005

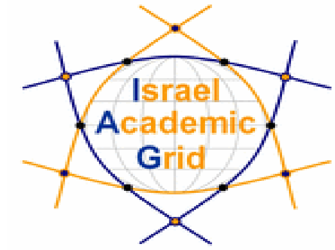
Message Passing Interface

Vered Kunik - Israeli Grid NA3 Team



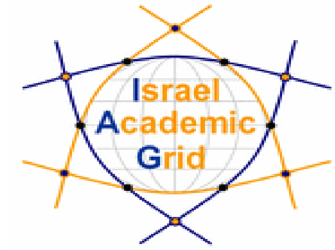
EGEE is a project funded by the European Union under contract IST-2003-508833

Outline



- Introduction
- Parallel Computing
- Parallel Programming models
- Designing parallel programs
- MPI
- Using MPI on the Grid

Introduction



- Traditionally, programs were written for serial computers

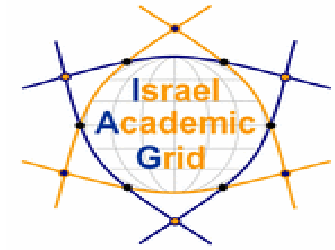
processing time is limited by hardware

- **The need for speed:** Nowadays, we are required to solve memory intensive problems with greater speed

requires a strategy for performing large, complex tasks faster = parallelism

parallel computing

Parallel Computing



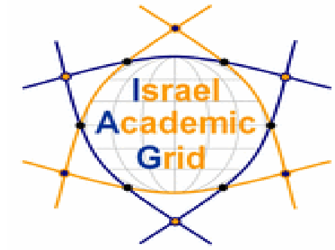
Aspects of parallel computing:

- Parallel computer architectures
- Parallel programming models

What is a parallel computer ?

- A single computer with multiple processors
- An arbitrary number of computers connected by a network (e.g., a cluster)
- A combination of both.

Parallel Computing – cont'd

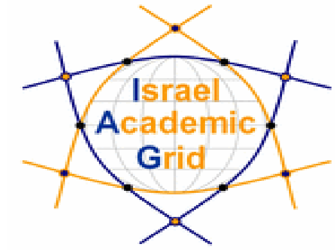


- All parallel computers use multiple processors

→ **inter-processor communication**

- Inter-processor communication is required to:
  - **Exchange information and data** between processors
  - **Synchronize** processors activities
- The means processors communicate depends on **memory architecture**:
  - <u>Shared memory</u> parallel computers
  - <u>Distributed memory</u> parallel computers

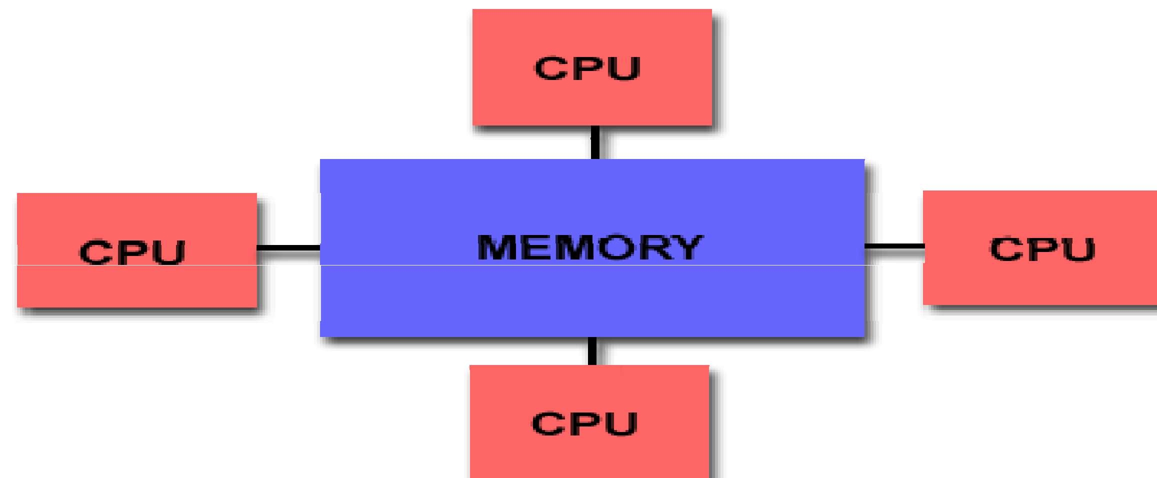
Basics of Parallel computers – cont’d



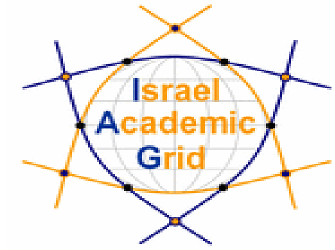
Shared memory parallel computers (UMA / SMP, NUMA)

- Multiple processor operate independently but share access to a global memory address space via a high-speed memory bus

⟹ Changes in a memory location effected by one processor are visible to all other processors



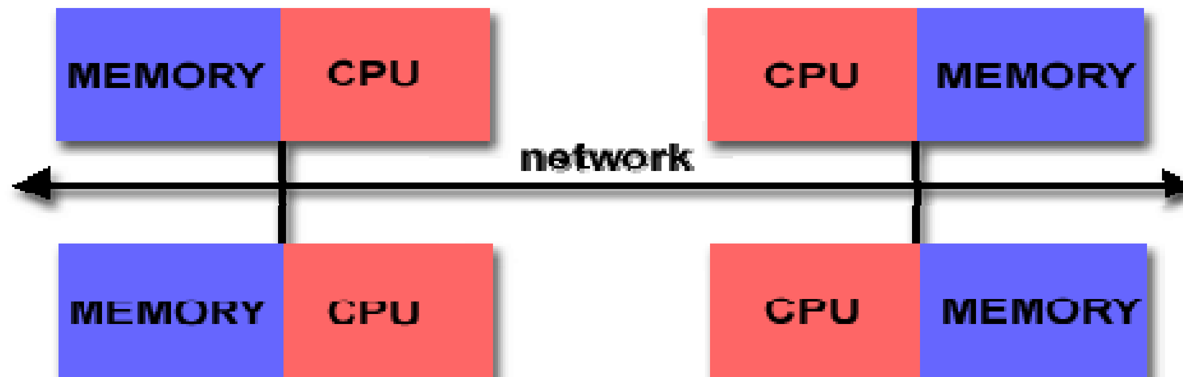
Basics of Parallel computers – cont'd



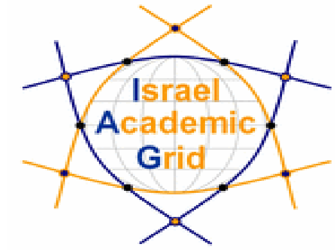
Distributed memory parallel computers

- A collection of independent processors connected via a communication network
- Each processors has its own local memory

→ requires a communication network to connect inter-processor memory

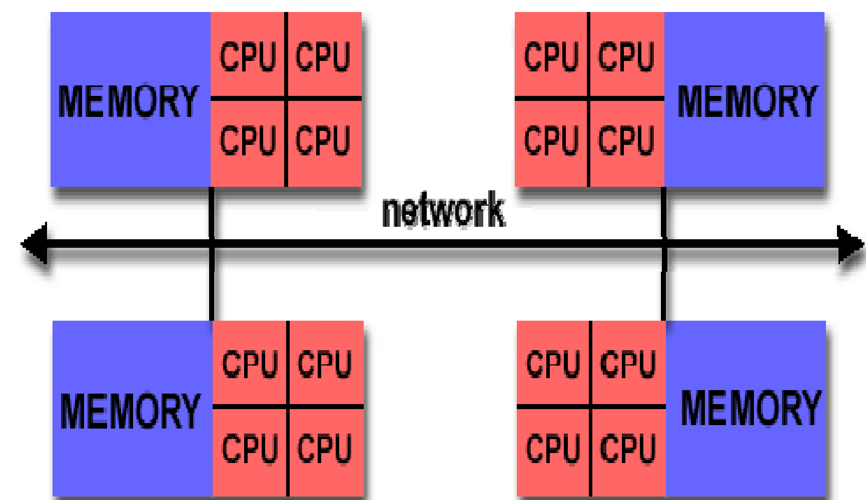


Basics of Parallel computers – cont’d

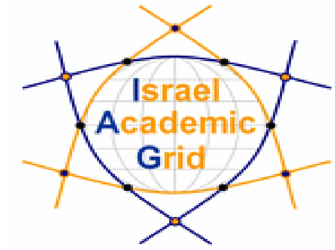


Hybrid Distributed-Shared Memory

- A shared memory component: a cache coherent SMP
- Processors on a given SMP can address that machine's memory as global
- A distributed memory component: networking of multiple SMPs
- SMPs know only about their own memory (not the memory on another SMP)



Parallel programming models

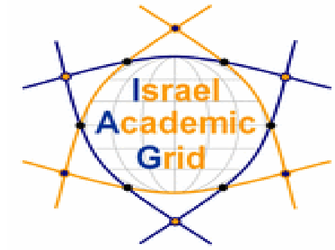


- Parallel programming models exist as an abstraction above hardware and memory architectures

→ **can be implemented on any underlying hardware**

- The most common **parallel programming models** are:
  - **Shared Memory:** Threads
  - **Data parallel**
  - **Message passing**: MPI, PVM, MPL
  - **"High-level" programming models**: **SPMD**, **MPMD**
- These models are NOT specific to a particular type of machine or memory architecture

Parallel programming models – cont'd

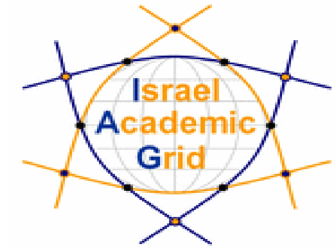


- There is no "best" model

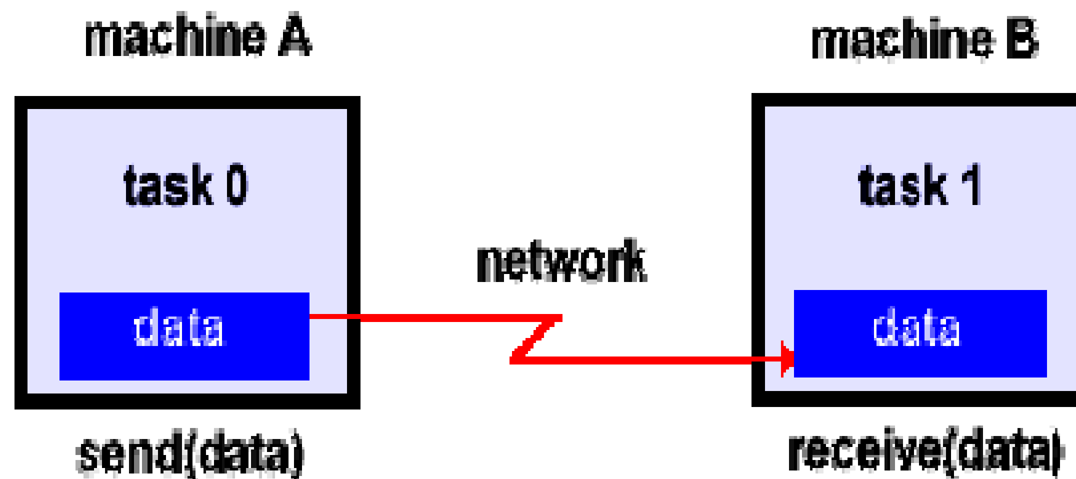
→ which model to use is often a combination of what is available and personal choice

- **Message Passing**
  - A set of processes that use their own local memory during computation
  - Multiple processes can reside on the same physical machine as well across an arbitrary number of machines

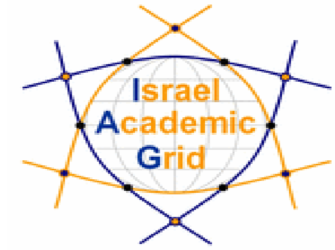
Parallel programming models – cont'd



- **Message passing – cont'd**
  - Processes exchange data through communications by sending and receiving messages
  - Data transfer requires cooperative operations to be performed by each process (i.e., a "matched" send-receive)



Designing parallel programs



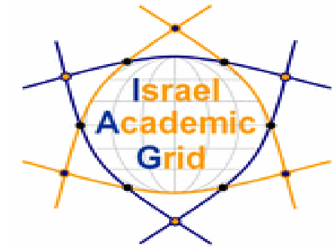
What does parallel programming mean ?

- Decomposing the problem into pieces that multiple processors can perform

Developing parallel programs:

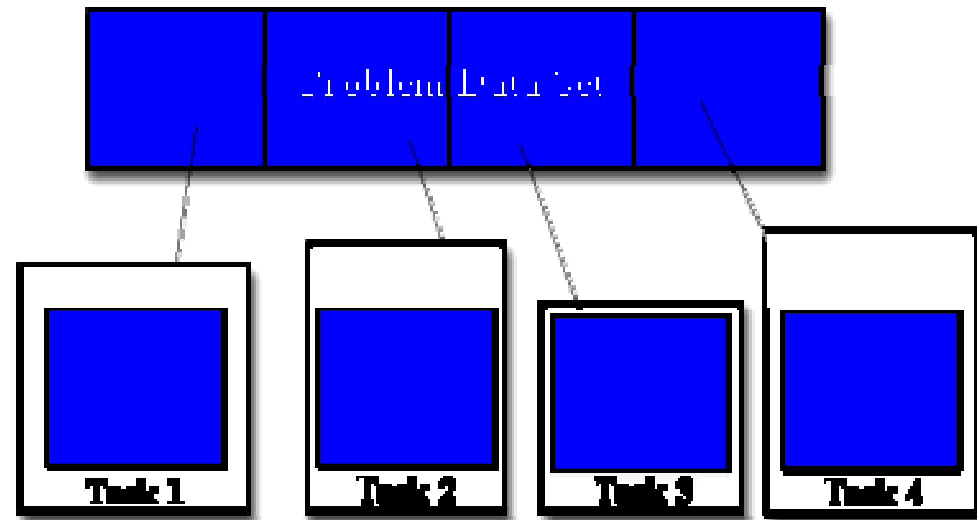
- Understanding the problem and program
- **Decomposition** i.e., break the problem into discreet "chunks"
- **Distributing** the “chunks” as processes which the processor worked on simultaneously
- **Coordinating** the work and communication of the processors

Designing parallel programs – cont'd

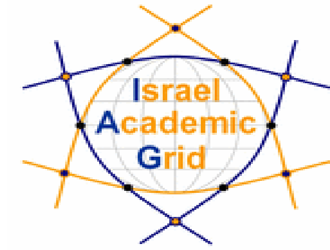


Domain decomposition – "data parallelism"

- The data is divided into pieces of approximately the same size
- Data "slices" are mapped to different processes
- Each process works only on the portion of the data that is assigned to it
- Requires periodic communication between the processes

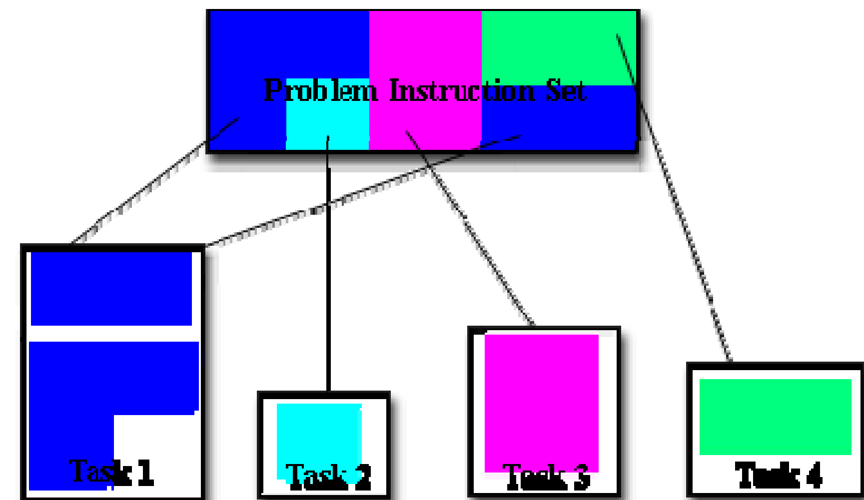


Designing parallel programs – cont'd

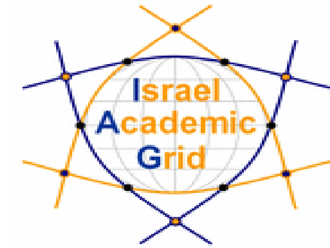


Functional Decomposition – "task parallelism"

- The focus is on the computation that is to be performed rather than on the data
- Each process performs a portion of the overall work
- Processes are assigned to the processors as they become available
- Implemented in a client-server paradigm
- As each process finishes its task, it is assigned a new input

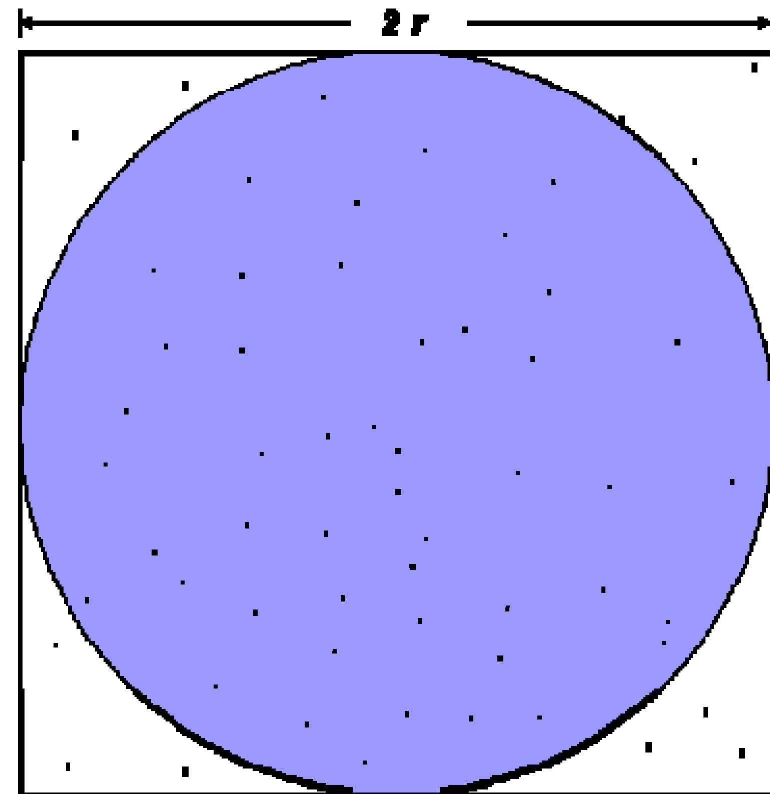


Designing parallel programs – cont'd

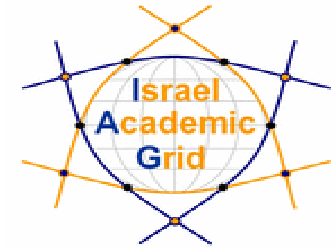


An Example: π calculation

- Inscribe a circle in a square
- Randomly generate points in the square (the more points = better approximation)
- Determine the number of points in the square that are also in the circle
- Let k be the number of points in the circle divided by the number of points in the square
- $\pi \sim 4$ k



Designing parallel programs – cont'd

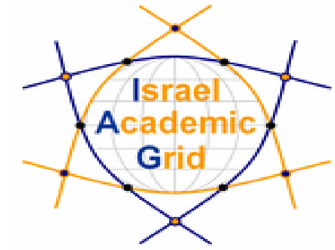


Serial solution:

most of running time

```
npoints = 10000;
 circle_count = 0;
 do j = 1,npoints
     // generate 2 random numbers between 0 and 1
     xcoordinate = random1;
     ycoordinate = random2;
     if (xcoordinate, ycoordinate) inside circle then
       circle_count = circle_count + 1;
  end do
 PI = 4.0*circle_count/npoints;
```

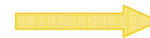
Designing parallel programs – cont'd



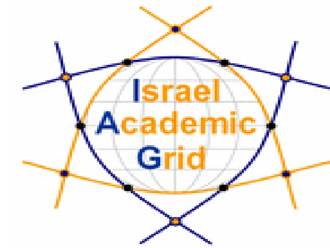
Parallel solution

- Break the loop into portions that can be executed by various processes

- **In order to approximate π** :
  - Each process executes its portion of the loop a number of times
  - Each process can do its work without requiring any information from the other tasks there are no data dependencies)
  - Uses the SPMD model: one task acts as master and collects the results



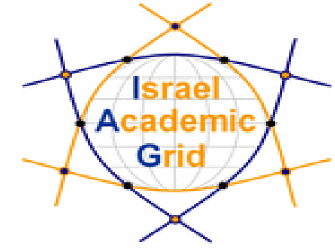
Designing parallel programs – cont'd



Changes for parallelism

```
npoints = 10000;
circle_count = 0;
p = number of processes;
num = npoints/p;
find out if I am MASTER or WORKER
do j = 1 , num
    // generate 2 random numbers between 0 and 1
     xcoordinate = random1;
     ycoordinate = random2;
     if (xcoordinate, ycoordinate) inside circle then
              circle_count = circle_count + 1;
end do
if I am MASTER
     receive from WORKERS their circle_counts;
     compute PI     // use MASTER and WORKER calculations
else if I am WORKER
     send to MASTER circle_count;
endif
```

MPI



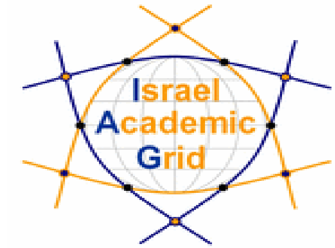
MPI = Message Passing Interface (C, C++, Fortran)

- A standard for message-passing libraries for parallel computing
- Employs the **SPMD** programming model: multiple instances of the same program run concurrently in separate address spaces communicating via messages

Why MPI ?

- Most portable (not hardware-dependent), functional (~150 functions) and widely available
- Designed for the widest possible range of parallel processors

MPI – cont'd



General MPI program structure:

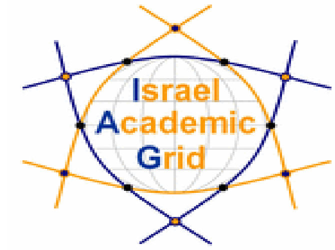
MPI include file — #include "mpi.h"

Initialize MPI Environment

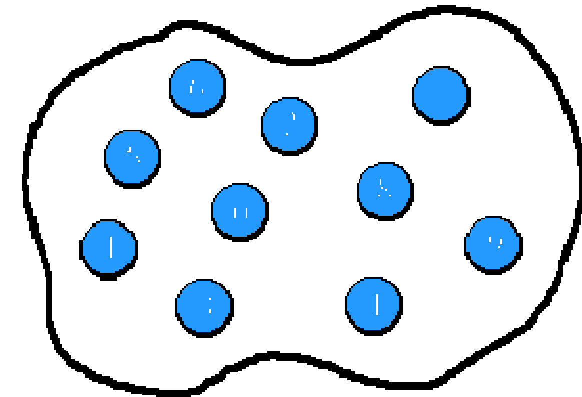
Do work and make message passing calls

Terminate MPI Environment

MPI – cont'd

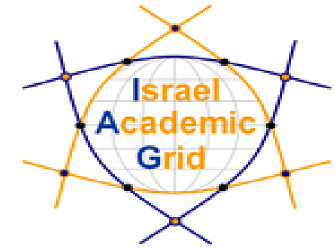


- MPI uses objects called **communicators** and **groups** to define which collection of processes may communicate with each other
- **MPI_COMM_WORLD** = predefined communicator that includes all of your MPI processes
- **Rank**: within a communicator, when the system initializes a process its gives the process its own unique, integer identifier



- Ranks are contiguous and begin at zero

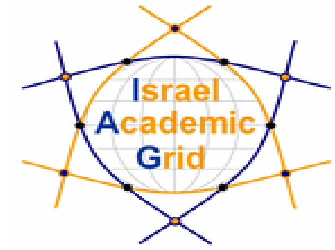
MPI – cont'd



- **6 most used MPI functions:**

MPI_Init	Initializes MPI
MPI_Finalize	Terminates MPI
MPI_Comm_size	Determines the number of processes
MPI_Comm_rank	Determines the label of the calling process
MPI_Send	Sends a message
MPI_Recv	Receives a message

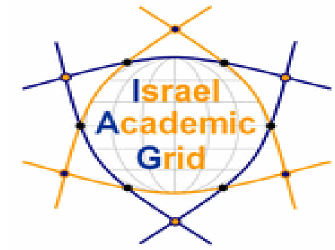
MPI – cont'd



int MPI_Init(int *argc, char **argv)

- Initializes the MPI execution environment
- All MPI programs must call MPI_Init before any other MPI routine is called
- must be called only once in an MPI program
- In C, it may be used to pass the command line arguments to all processes

MPI – cont'd



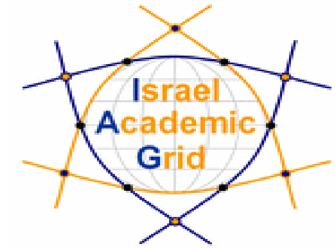
int MPI_Finalize()

- Terminates the MPI execution environment should be the last MPI routine called in every MPI program ('shutdown')

int MPI_Comm_size(MPI_Comm comm, int *size)

- Determines the number of processes in the group associated with a communicator
- Generally used within the communicator MPI_COMM_WORLD

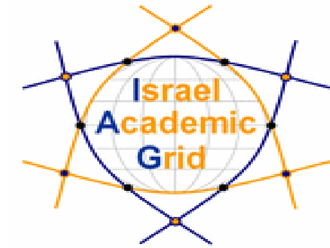
MPI – cont'd



int MPI_Comm_rank(MPI_Comm comm, int *rank)

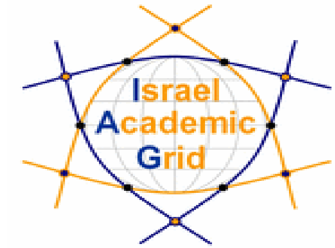
- Determines the rank of the calling process within the communicator
- Each process is assigned a unique integer rank between 0 and (#processes – 1) within MPI_COMM_WORLD
- If a process becomes associated with other communicators, it will have a unique rank within each of these as well

MPI – cont'd



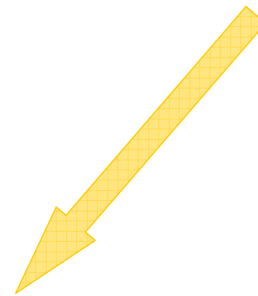
MPI Datatype	C Datatype
MPI_CHAR	Signed char
MPI_SHORT	Signed short int
MPI_INT	Signed int
MPI_LONG	Signed long int
MPI_UNSIGNED_CHAR	Unsigned char
MPI_UNSIGNED_SHORT	Unsigned short int
MPI_UNSIGNED	Unsigned int
MPI_UNSIGNED_LONG	Unsigned long int
MPI_FLOAT	float
MPI_DOUBLE	double
MPI_LONG_DOUBLE	Long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

MPI – cont'd



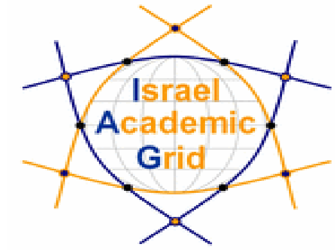
- An Example:

```
#include "mpi.h"
 Main(int argc, char *argv[ ])
 {
    int npes, myrank;
    MPI_Init(&args, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &npes);
    MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
    /*do some work*/
    MPI_Finalize();
 }
```



Created when MPI_Init is called

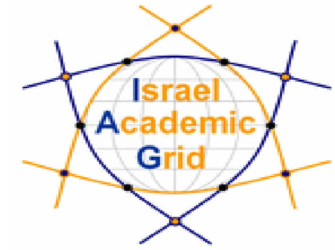
MPI – cont'd



Communication:

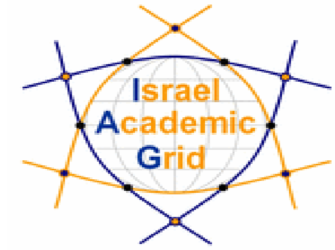
- **Point-to-Point communication**
  - Involves message passing between two, and only two, different MPI tasks
  - One task is performing a send operation and the other task is performing a matching receive operation
  - There are different types of send and receive routines used for different purposes:
    - Synchronous send
    - Blocking send / blocking receive
    - etc`

MPI – cont'd



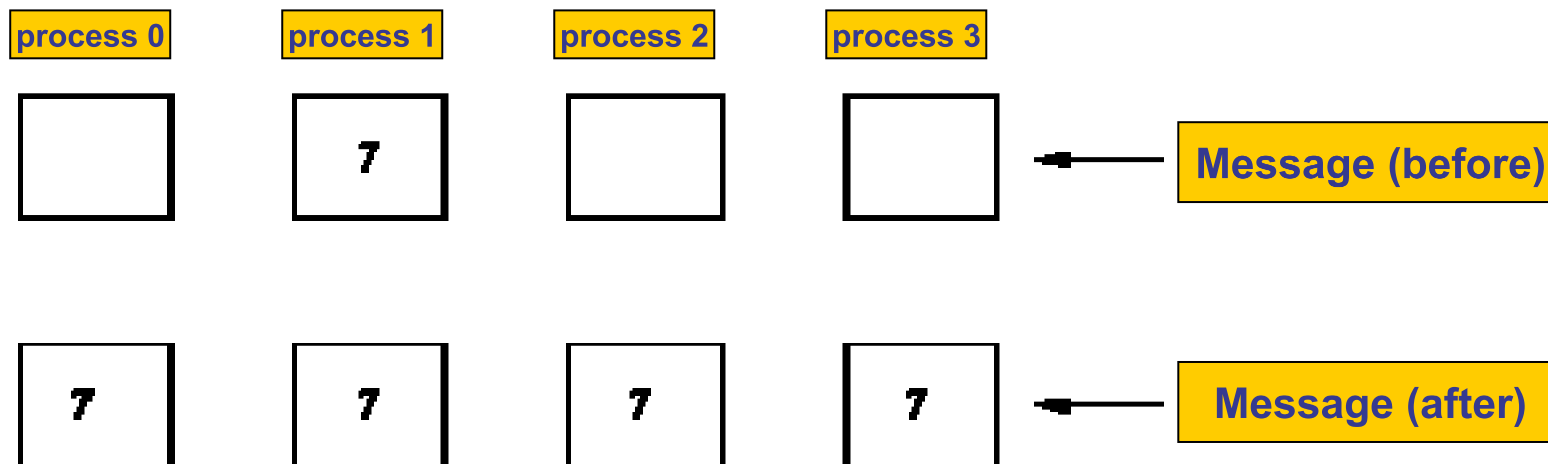
- **Collective communication**
  - Collective communication must involve **all** processes in the scope of a communicator
  - Collective communication routines:
    - **Synchronization** - processes wait until all members of the group have reached the synchronization point
    - **Data Movement** - broadcast, scatter/gather, all to all
    - **Collective Computation** (reductions) - one member of the group collects data from the other members and performs an operation (min, max, etc.) on that data

MPI – cont'd



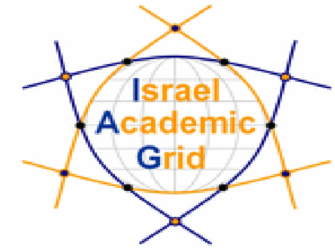
MPI_Bcast()

- Broadcasts (sends) a message from the process with rank "root" to all other processes in the group



MPI_Bcast(&msg,1,MPI_INT,1,MPI_COMM_WORLD)

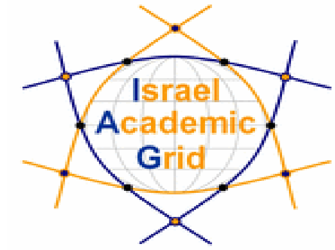
Using MPI on the Grid



- MPICH is a public domain version of the MPI library
- The JDL **JobType** attribute should be set to **MPICH**
- The **NodeNumber** attribute should be used to indicate the required **number of CPU's**
- The following are to the jdl **Requirements** attribute:
  - **Member("MpiCH", other.GlueHostApplicationSoftwareRunTimeEnvironment)**

  ⇒ Indicate that the MPICH runtime environment must be installed on the computing element

Using MPI on the Grid – cont'd



- **other.GlueCEInfoTotalCPUs >= NodeNumber**

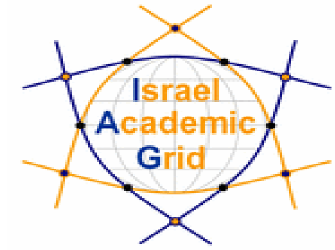
→ number of CPUs must be at least equal to the required number of nodes

- To the **Rank** attribute:
  - **other.GlueCEStateFreeCPUs**

→ The CE with the largest number of free CPUs is chosen

Lets use the Grid to approximate π

Using MPI on the Grid – cont’d



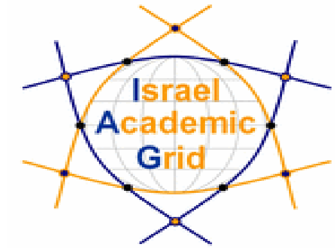

```
Type = "Job";
JobType = "MPICH";

Executable = "MPItest.sh";
Arguments = "cpi 2";
NodeNumber = 2;

StdOutput = "test.out";
StdError = "test.err";

InputSandbox = {"MPItest.sh","cpi"};
OutputSandbox = {"test.err","test.out","executable.out"};

Requirements   = other.GlueCEInfoLRMSType == "PBS" ||
                 other.GlueCEInfoLRMSType == "LSF";
```

# of required CPUs

The executable

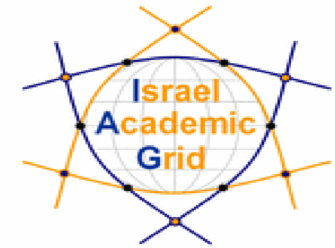
Renders the computation to the WNs allocated for the parallel execution

Using MPI on the Grid – cont'd

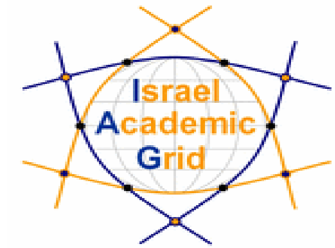



```
#!/bin/sh
# this parameter is the binary to be executed
EXE=$1
# this parameter is the number of CPU's to be reserved for parallel execution
CPU_NEEDED=$2
# prints the name of the master node
echo "Running on: $HOSTNAME"
echo "****************************************"
if [ -f "$PWD/.BrokerInfo" ] ; then
TEST_LSF=`edg-brokerinfo getCE | cut -d/ -f2 | grep lsf`
else
TEST_LSF=`ps -ef | grep sbatchd | grep -v grep`
fi
if [ "x$TEST_LSF" = "x" ] ; then
# prints the name of the file containing the nodes allocated for parallel execution
echo "PBS Nodefile: $PBS_NODEFILE"
# print the names of the nodes allocated for parallel execution
cat $PBS_NODEFILE
echo "****************************************"
HOST_NODEFILE=$PBS_NODEFILE
else
# print the names of the nodes allocated for parallel execution
echo "LSF Hosts: $LSB_HOSTS"
# loops over the nodes allocated for parallel execution
HOST_NODEFILE=`pwd`/lsf_nodefile.$$
for host in ${LSB_HOSTS}
do
echo $host >> ${HOST_NODEFILE}
done
fi
```

Using MPI on the Grid – cont’d




```
echo "****************************************"
# prints the working directory on the master node
echo "Current dir: $PWD"
echo "****************************************"

for i in `cat $HOST_NODEFILE` ; do
echo "Mirroring via SSH to $i"
# creates the working directories on all the nodes allocated for parallel execution
ssh $i mkdir -p `pwd`
# copies the needed files on all the nodes allocated for parallel execution
/usr/bin/scp -rp ./* $i:`pwd`
# checks that all files are present on all the nodes allocated for parallel execution
echo `pwd`
ssh $i ls `pwd`
# sets the permissions of the files
ssh $i chmod 755 `pwd`/$EXE
ssh $i ls -alR `pwd`
echo "@@@@@@@@@@@@@@@@"
done

# execute the parallel job with mpirun
echo "************************************"
echo "Executing $EXE"
chmod 755 $EXE
ls -l
mpirun -np $CPU_NEEDED -machinefile $HOST_NODEFILE `pwd`/$EXE > executable.out
echo "************************************"
```



MPI job submission tutorial