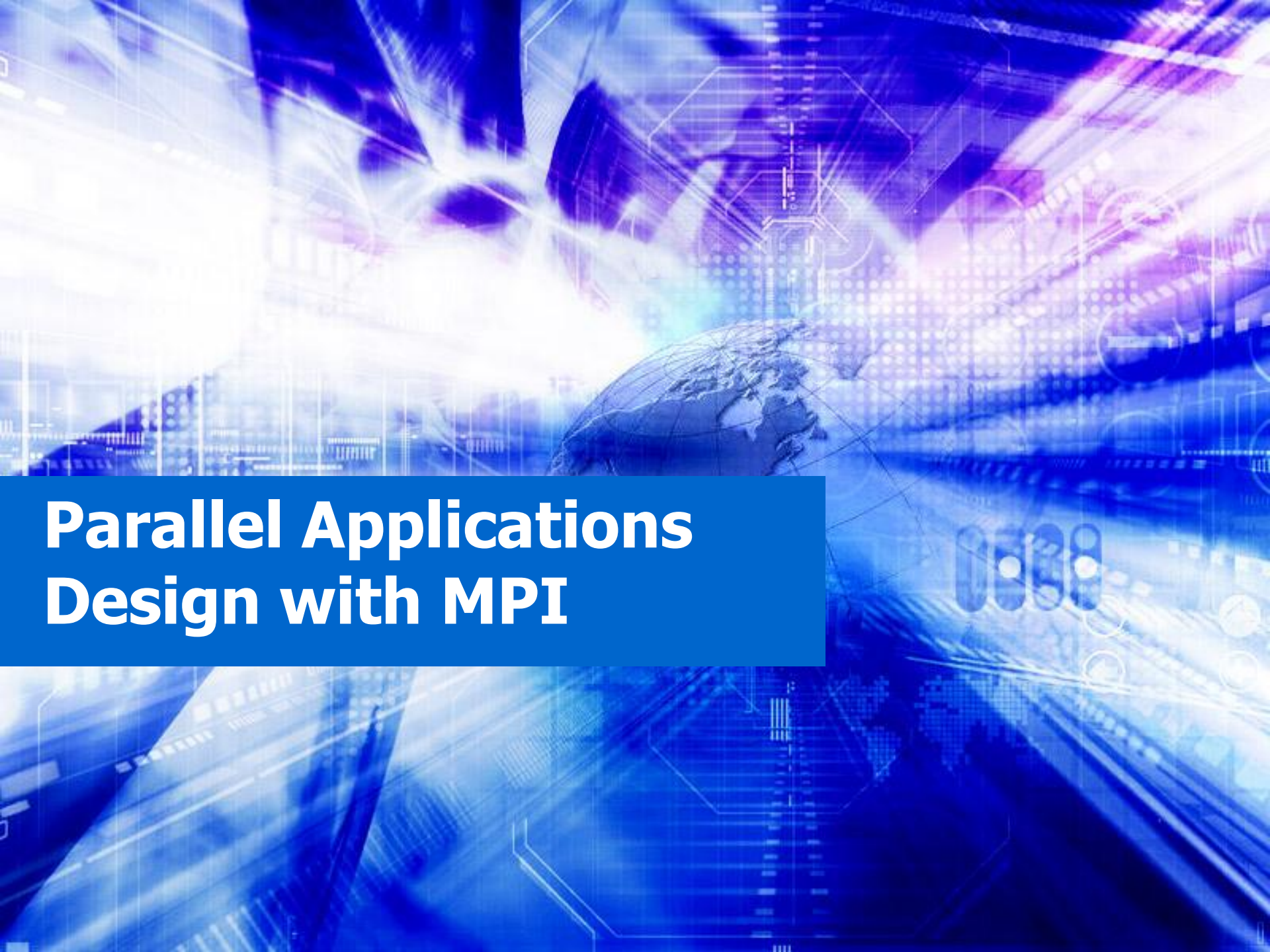
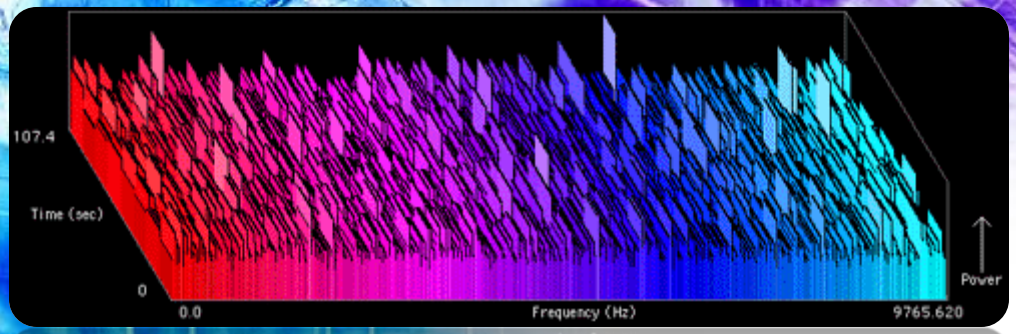
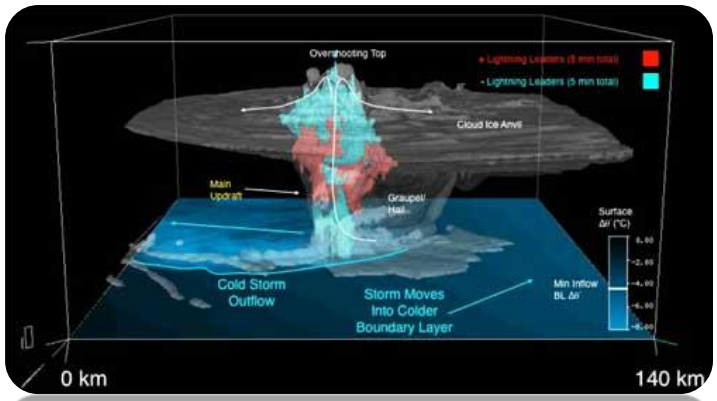
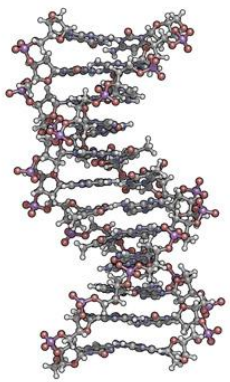

Parallel Applications
Design with MPI

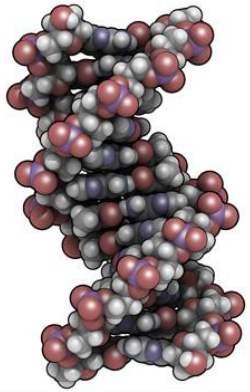


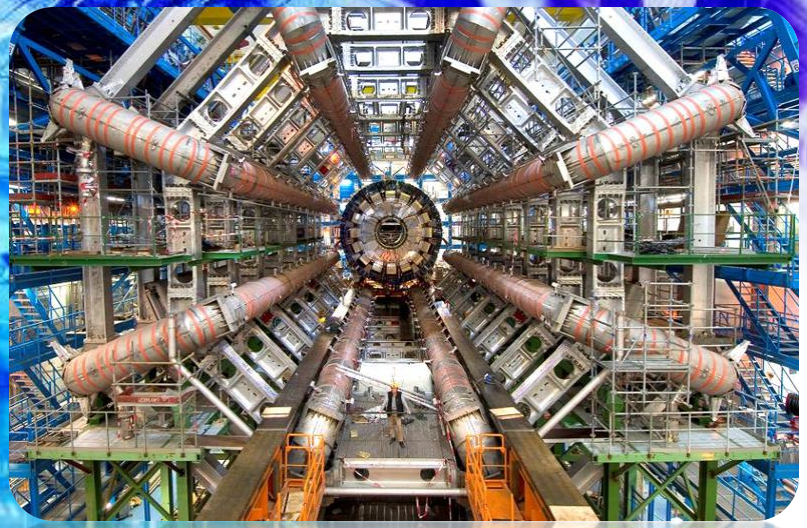
Science Research Challanges

- **Killer applications**
  - Challenging use of computer power and storage
  - Who might be interested in those applications?
  - Simulation and analysis in modern science









Example: Large Hadron Collider as (CERN)

- **LHC Computing Grid**
  - Worldwide collaboration of > 170 computing centers in 34 countries
  - Recording rate (raw) 1 GByte/sec
  - Sum between 5 up to 8 PetaByte/year (10^{15} B/year)
  - Estimated to be 200,000 faster than some today's fastest CPUs
  - How can we get this working?

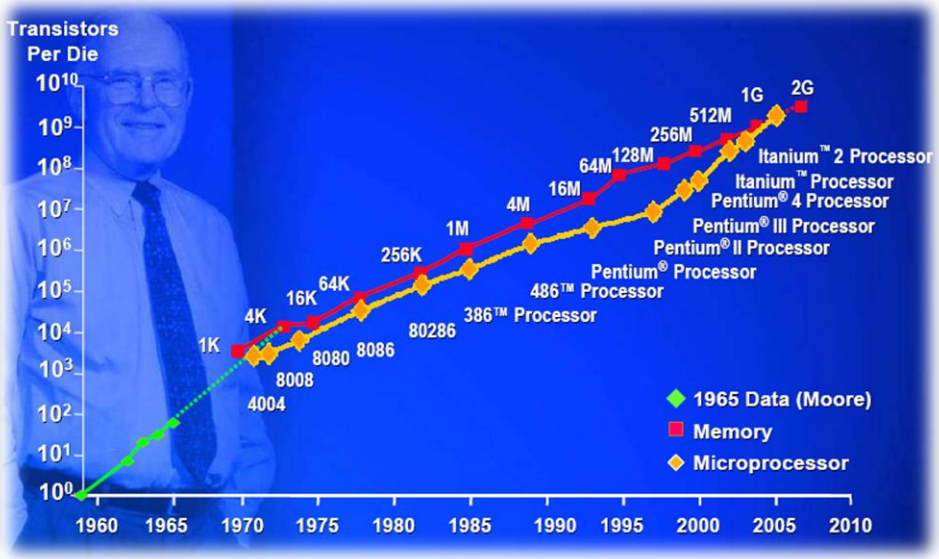
The need for HPC

- **Moore's law**

  - In 1965 stated that transistor density double at each 1.5 years

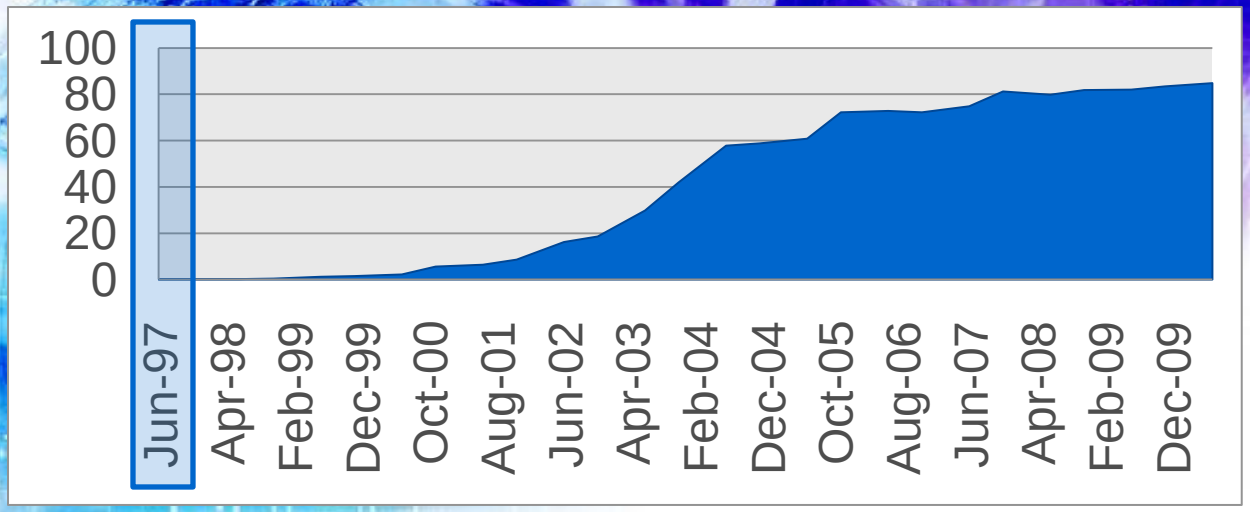
  - A wrong assumption is that if transistor density double than computer gets twice as fast every 18/24 months

  - This is wrong: the proof is that today highest clocks don't reach 20GHz



Can we use a single processor?

- **Short answer is: no**
  - Curiously Moore's law is still true
  - Transistor densisty indeed doubles but higher clock rate leads to unmanageable heat and power consumption
  - Super computers are too expensive for medium size problems
  - The solution is work with multiple processors at the same time
  - If super computers are too expensive why not create a machine clustering desktop solutions



Clusters Computing

- **Cluster of workstations**
  - Commodity PCs interconnected through a network
  - More affordable than supercomputers
  - Adaptable to any scale of usage
  - Fast aquired popularity among researchers

- **Timeline**
  - 1993: Beowulf project
  - 1997: Berkley NoW is first cluster on top500
  - 2010: 80 % of machines in top500 are clusters
  - Source: **www.top500.org**

Programming on a Cluster

- **Applications must be rewritten**
  - Proccess communication changes
    - Instead of memory must use the network
- **Possible solutions**
  - Ad Hoc
    - Work only for the platform it was designed for
  - PVM
    - Research project for heterogeneous network computing
  - MPI
    - It's a standard, independent of implementation
    - Have more than three free implementations
    - Here we are going to talk about MPI

MPI – Message Passing Interface

- **MPI in a nutshell**
  - It is a library specification
  - Works natively with C and Fortran
  - Not a specific implementation or product
  - Scalable
    - Must handle multiple machines
  - Portable
    - Sockets API change from one OS to another
    - Handles Big-endian/little-endian architectures
  - Efficient
    - Optimized communication algorithms
    - Allow communication and computation overlap

MPI – Message Passing Interface

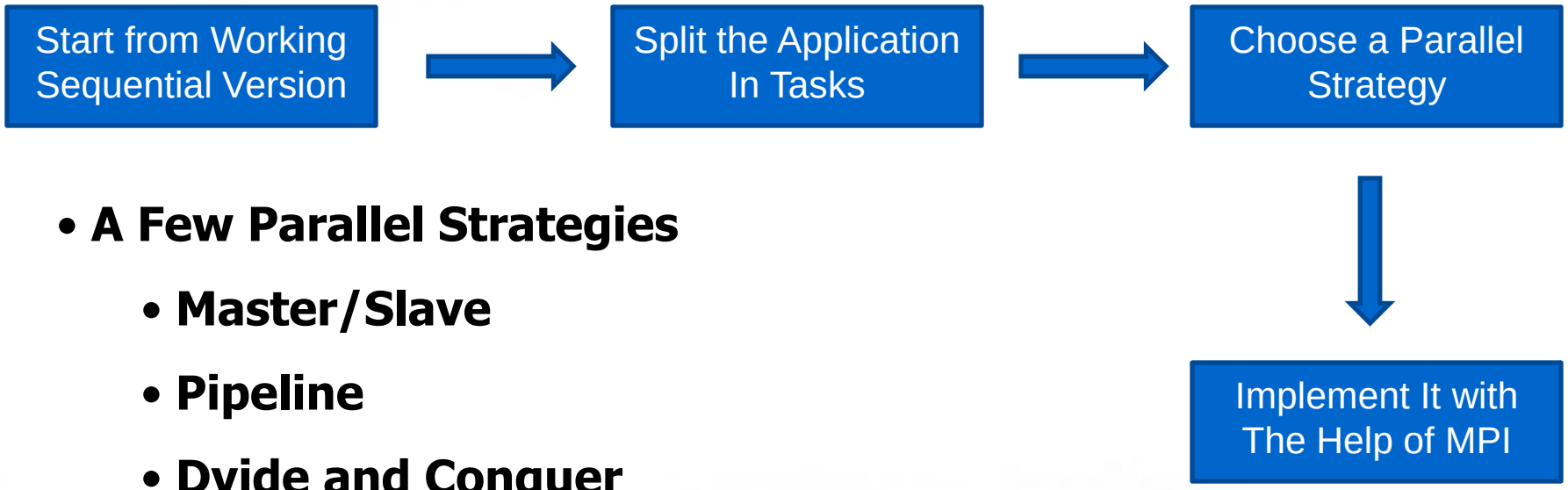
- **MPI References**
  - Books
    - Using MPI: Portable Parallel Programming with the Message Passing Interface, by Gropp, Lusk, and Skejellum, MIT Press, 1994.
    - MPI: The Complete Reference, by Snir, Otto, Huss-Lederman, Walker, and Dongarra, MIT Press, 1996.
    - Parallel Programming with MPI, by Peter Pacheco, Morgan Kaufmann, 1997.
  - The standard:
    - at http://www.mpi-forum.org

MPI Programming

- **MPI**
  - Use of a single program, on multiple data
  - What does it do?
    - way of identifying process
    - Low-level independent API
    - Optimized communication
    - Allow communication and computation overlap
  - What does it do not?
    - gain performance of application for free
    - application must be adapted

MPI Programming

- **Possible Programming Workflow**



- **A Few Parallel Strategies**
  - **Master/Slave**
  - **Pipeline**
  - **Dvide and Conquer**

Parallel Strategies

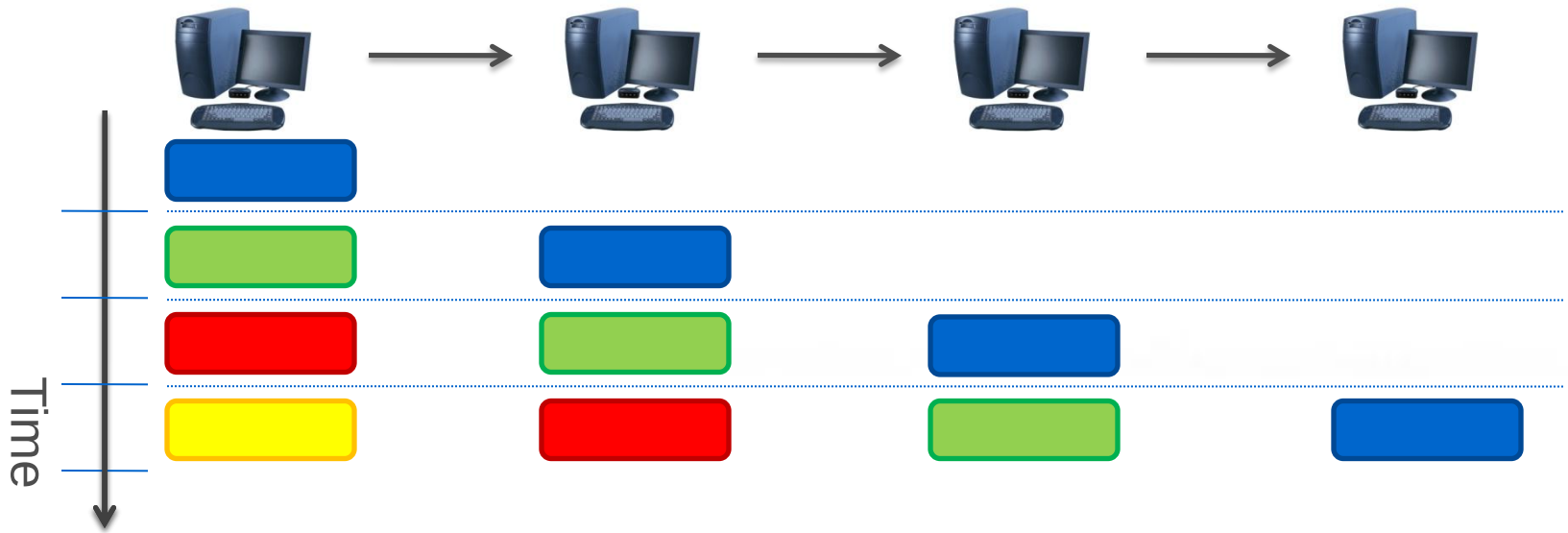
- **Master/Slave**
  - **Master is one process that centrilizes all tasks**
  - **Slaves starve for work**

Parallel Strategies

- **Master/Slave**
  - **Master is often the bottleneck**
  - **Scalability is limited due to centralization**
  - **Possible to use replication to improve performance**
  - **It is adatable to heterogenous platforms**

Parallel Strategies

- **Pipeline**
  - **Each process plays a specific role, pipeline stages**
  - **Data follows in a single direction**
  - **Parallelism is achieved when the pipeline is full**

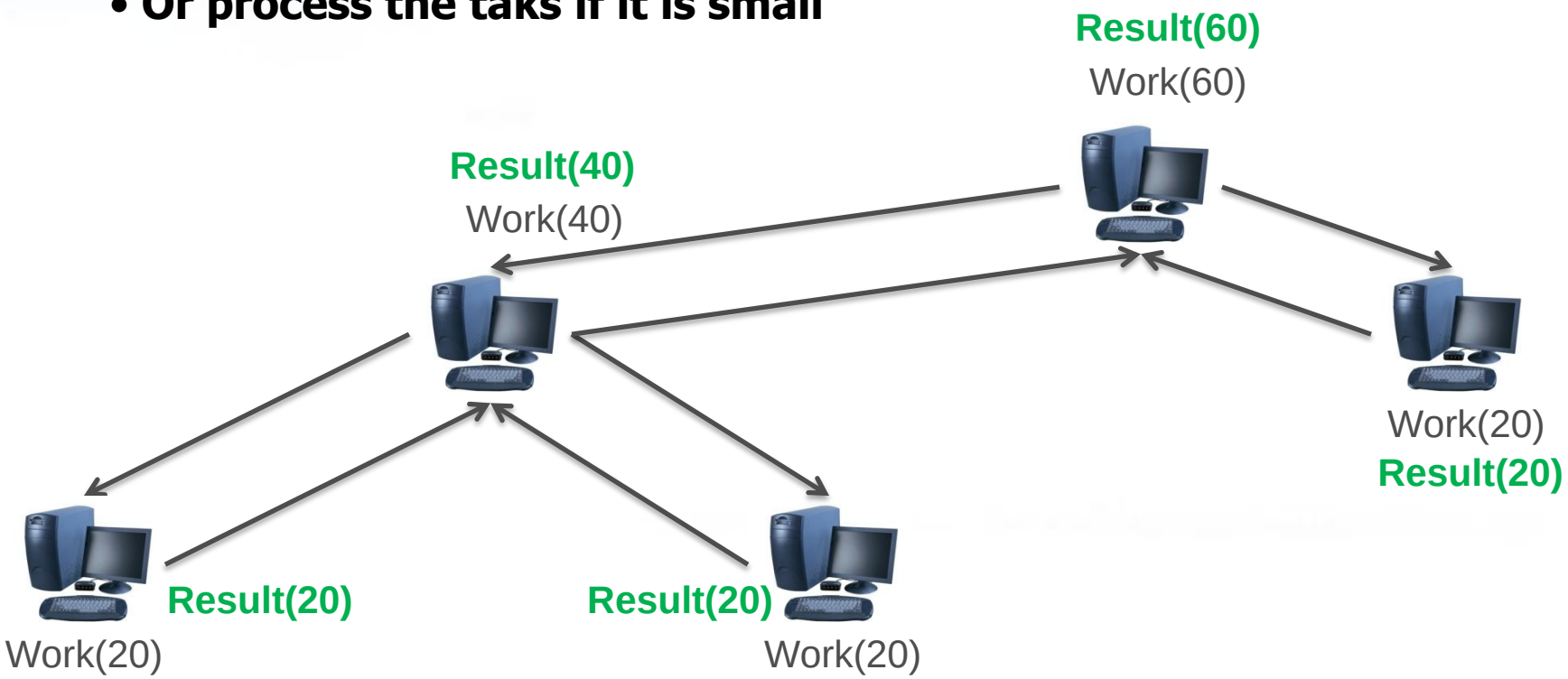


Parallel Strategies

- Pipeline
  - Scalabillity is limited by the number of stages
  - Synchronization may lead to bubbles
    - Slow sender
    - Fast receiver
  - Difficult to use on heterogenous platforms

Parallel Strategies

- **Divide and Conquer**
  - **Recursevely partion task on roughly equal sized tasks**
  - **Or process the taks if it is small**



Parallel Strategies

- **Divide and Conquer**
  - **More scalable**
  - **Possible to use replicated branches**
  - **In practice is difficult to split tasks**
  - **Suitable for branch and bound algorithms**

MPI Programming

- **Installing**
  - Some common MPI implementations, all free:
    - OpenMPI

      http://www.open-mpi.org/

    - MPICH-2

      http://www.mcs.anl.gov/research/projects/mpich2/

    - LAM/MPI

      http://www.lam-mpi.org/

MPI Programming

- **Installing**
  - I'm using MPICH-2
  - Installed in Ubuntu 10.04 Lucid Lynx with

```
$ sudo apt-get install mpich2
```

  - Should work for most Debian based distributions
  - Must create a local configuration file

```
$ echo "MPD_SECRET_WORD=ChangeMe" > ~/.mpd.conf
```

MPI Programming

- **Test program**

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

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

/* Initialize MPI */
MPI_Init(&argc, &argv);

printf("Test Program\n");

/* Finalize MPI */
return MPI_Finalize();
}
```

MPI Programming

- **Compiling**

– Compiled with gcc, but a mpicc script is provided to invoke gcc with specific MPI options enabled

```
$ mpicc mpi_program.c -o my_mpi_executable
```

– Executed with a specital script

```
$ mpirun -np 1 my_mpi_executable
$ mpirun -np 2 my_mpi_executable
$ mpirun -np 3 my_mpi_executable
```

MPI Programming

- **Running**
  - Compiled with gcc, but a mpicc wrapper is provided to invoke gcc with specific mpi options

```
$ mpicc mpi_program.c -o my_mpi_executable
```

  - For a complete list of parameters try

```
$ man mpicc
```

  - Executed with a special wrapper

```
$ mpirun -np 2 my_mpi_executable
```

MPI Programming

- **Exercise 1 – Hello World**

  - **Compile and run the simplest MPI program that only prints the "Hello World" string and after exits**

  - **Try vary the –np <nproc> parameter and observe the differences**

MPI Programming

- **How many proccess are running?**

```
int MPI_Comm_size(MPI_Comm comm, int *psize)
```

  - **comm**
    - Group of process to communicate
    - For grouping all process use **MPI_COMM_WORLD**
  - **psize**
    - Passed as reference will return the total amoung of proccess in this communicator

MPI Programming

- **Exercise 2 – Number of Proccess**

  - **Create program that prints the total number of available process on the screen**

  - **Vary the "–np <param>" to verify that your program is working**

MPI Programming

- **Assigning Process Roles**

```
int MPI_Comm_rank(MPI_Comm comm, int *rank)
```

  - **comm**
    - Group of process to communicate
    - To group all available process use **MPI_COMM_WORLD**
  - **rank**
    - Passed as reference will return the unique ID of the calling process in this communicator

MPI Programming

- Exercise 3 – Who am I?
  - If I am process 0
    - Prints: “hello world”
  - else
    - Prints: “I’m process <ID>”
    - Replacing <ID> by the process rank

MPI Programming

- **Running on Grid5000 1/2**

  – Log in grid5000 using the instructions gave to you

```
$ ssh <username>@access.<frontend>.grid5000.fr
```

  –Log in the front end of your choice

```
$ ssh <frontend>
```

  – Make a reservation

```
$ oarsub -l nodes=4,walltime=1 -I
```

MPI Programming

- **Running on Grid5000 2/2**
  - When connected to a cluster in interactive mode a file with the list o available machines is generated

```
$ cat $OAR_FILENODES
```

  - Compile your code again

```
$ mpicc –o mybin myprogram.c
```

  - Run MPI application

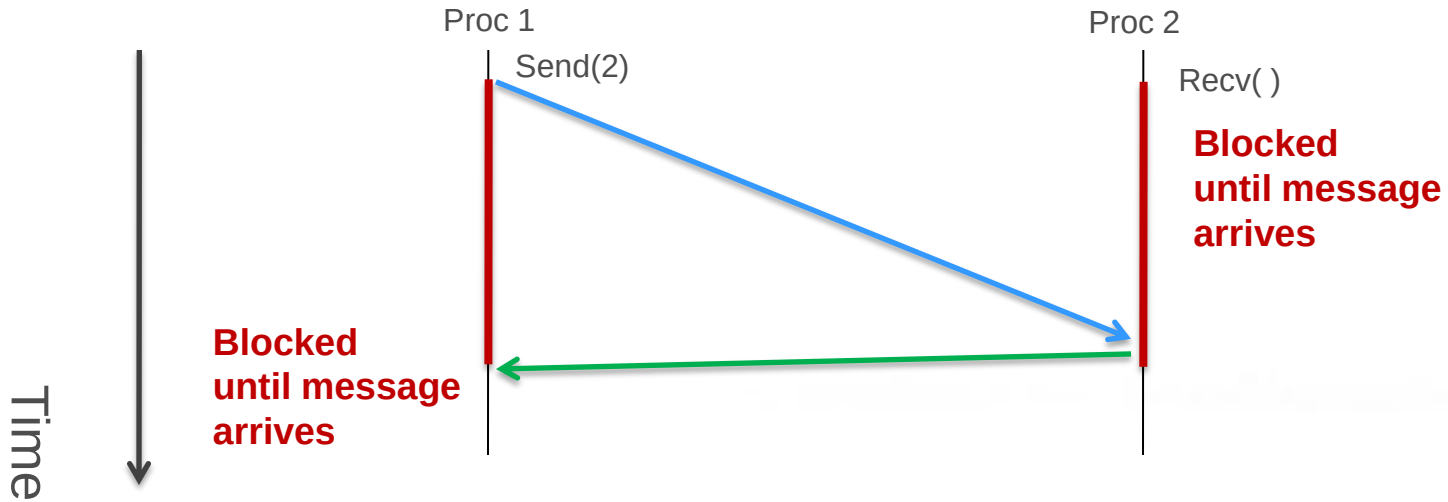
```
$ mpirun --mca plm_rsh_agent oarsh -machinefile $OAR_FILENODES -np <nproc> mybin
```

MPI Programming

- **Exercise 4 – Running on a cluster**
  - **Use your Grid5000 account to run the "Who am I?" program in a real cluster**
  - **Use the gethostname function to print the host name where the process is running**

MPI One-to-one Communication

- **Synchronous/Blocking**
  - Process sits waiting for message to arrive
  - Synchronization purpose



MPI Programming

- **Blocking Send 1/2**

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

  - **buf**
    - Pointer to the data to be sent
  - **count**
    - Number of data elements in buf
  - **dtype**
    - Type of elements in buf

MPI Programming

- **Blocking Send 2/2**

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

  - **dest**
    - Rank of destination process
  - **tag**
    - Tag another integer to identify the message
  - **comm**
    - Same as before, for all proccess use **MPI_COMM_WORLD**

MPI Programming

- **Blocking Receive 1/2**

```
int MPI_Recv(void *buf, int count, MPI_Datatype dtype,
        int src, int tag, MPI_Comm comm, MPI_Status &status)
```

  - **buf**
    - Pointer where data will be received if succeed
  - **count**
    - Maximum number of elements that buf can handle
  - **dtype**
    - Type of elements in buf

MPI Programming

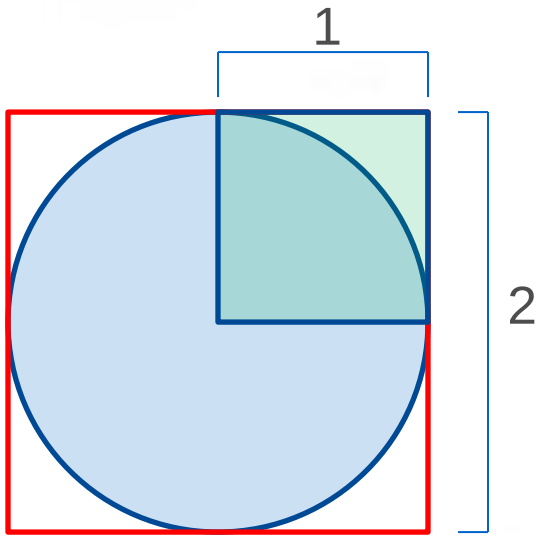
- **Blocking Receive 2/2**

```
int MPI_Recv(void *buf, int count, MPI_Datatype dtype,
        int src, int tag, MPI_Comm comm, MPI_Status &status)
```

  - **src**
    - Rank of sender process
  - **tag**
    - Message tag
  - **stat**
    - Sending process info, if desired can be ignored using **MPI_STATUS_IGNORE**

MPI Programming

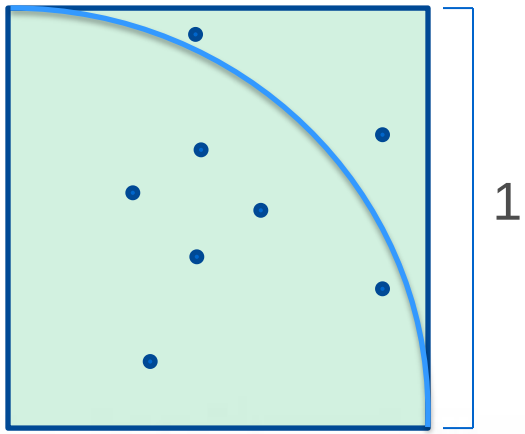
- **Exercise 5 (1/2) – Computing π by Monte Carlo Methods**



Area of Circle =

$\pi r^2 = \pi (1)^2 = \pi$

$A = \frac{\pi}{4}$ $P(I) = A = \frac{\pi}{4}$



P(I) = (Inside/Total)*4

P(I) = (6/8)*4 = 3

MPI Programming

- Exercise 5 (2/2) – Computing π by Monte Carlo Methods
  - Generate two random numbers X, Y in [0,1]
  - If (X*X + Y*Y) <= 1
    - Add 1 to counter
  - At the end use (counter/total)*4 as π approximation
  - More random points generated more close to π

MPI Programming

- **Performance Evaluation**
  - **Elapsed Time**
    - The timer itself
  - **Speedup**
    - How many times my application is faster than the sequential version
  - **Efficiency**
    - Estimate processing power dissipation

MPI Programming

- **Timer**

`double MPI_WTime()`

- **RETURN**
  - The time passed, in seconds, since an arbitrary time in the past

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

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

/* Initialize MPI */
MPI_Init(&argc, &argv);

t1 = MPI_WTime();
compute_pi();
t2 = MPI_WTime();
printf("Elapsed time: %f\n", t2 - t1);

/* Finalize MPI */
return MPI_Finalize();
}
```

MPI Programming

- **Performance Evaluation**
  - **Speedup**
    - Obtained from elapsed time
    - Ratio of elapsed time with one processor and elapsed time with n processors
      - **Speedup(n) = $\frac{\mathbf{T(1)}}{\mathbf{T(n)}}$**
    - T(1) = elapsed time with one processor
    - T(n) = elapsed time with n processors
    - The ideal is: **Speedup(i) = i**
    - Meaning: using **i** processors I get **i** times faster

MPI Programming

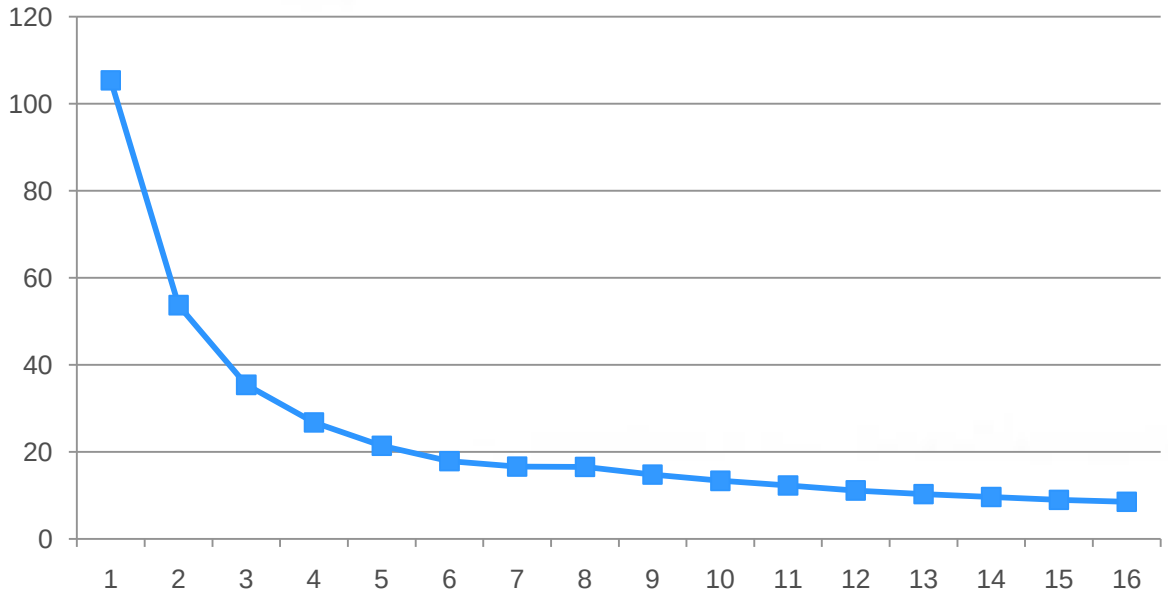
- **Performance Evaluation**
  - **Efficiency**
    - It is obtained from speedup
    - The efficiency shows the percentage of usage by processor
      - **Efficiency(n) = $\frac{\text{Speedup(n)}}{\text{n}}$**
    - Ideal is: **Efficiency(i) = 1**
    - Meaning:
      - Each processor is being used at 100%, no parallelization overhead

MPI Programming

- **Exercise 6 – Performance Evluation of the π Problem**
  - **Compute the elapsed time to compute 10^9 iterations of Pi with:**
    - **1, 2, 4, 8, 10 processors**
    - **Present speedup and efficiency**

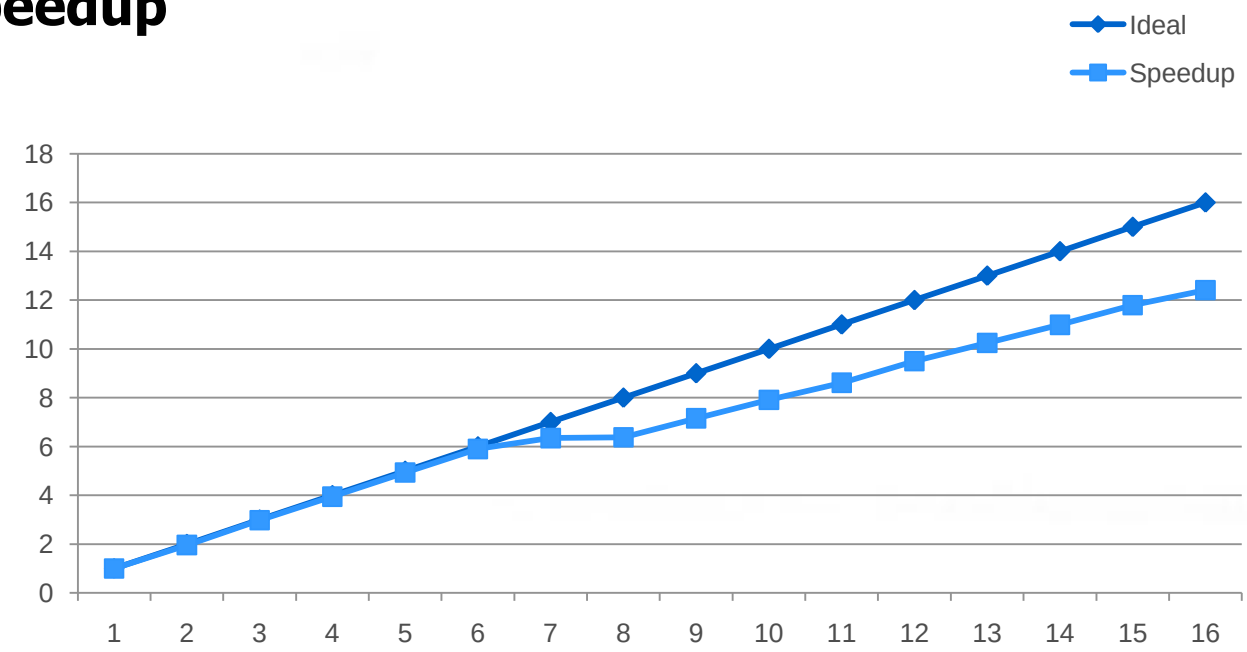
MPI Programming

- **Performance Evluation of the π Problem**
  - **Elapsed Time**



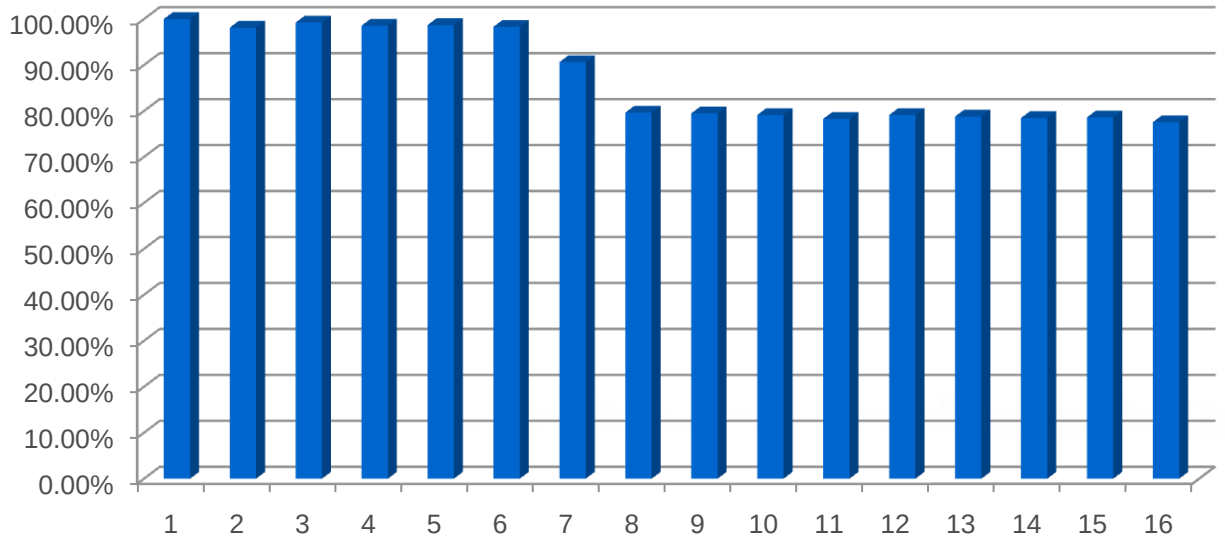
MPI Programming

- **Performance Evluation of the π Problem**
  - **Speedup**



MPI Programming

- **Performance Evluation of the π Problem**
  - **Efficiency**



MPI Programming

- **Deadlock**

  – Two process, or more, are blocked waiting for the other

  –The computation do not advance

MPI Programming

- **Deadlock**

```
...

if(my_Rank == 0){
    MPI_Recv(&tmp, 1, MPI_INT, 1, 122, MPI_COMM_WORLD, MPI_IGNORE_STATUS);
    MPI_Send(&dat, 1, MPI_INT, 1, 122, MPI_COMM_WORLD);
}

if(my_Rank == 1){
    MPI_Recv(&tmp, 1, MPI_INT, 0, 122, MPI_COMM_WORLD, MPI_IGNORE_STATUS);
    MPI_Send(&dat, 1, MPI_INT, 0, 122, MPI_COMM_WORLD);
}

...
```

MPI Programming

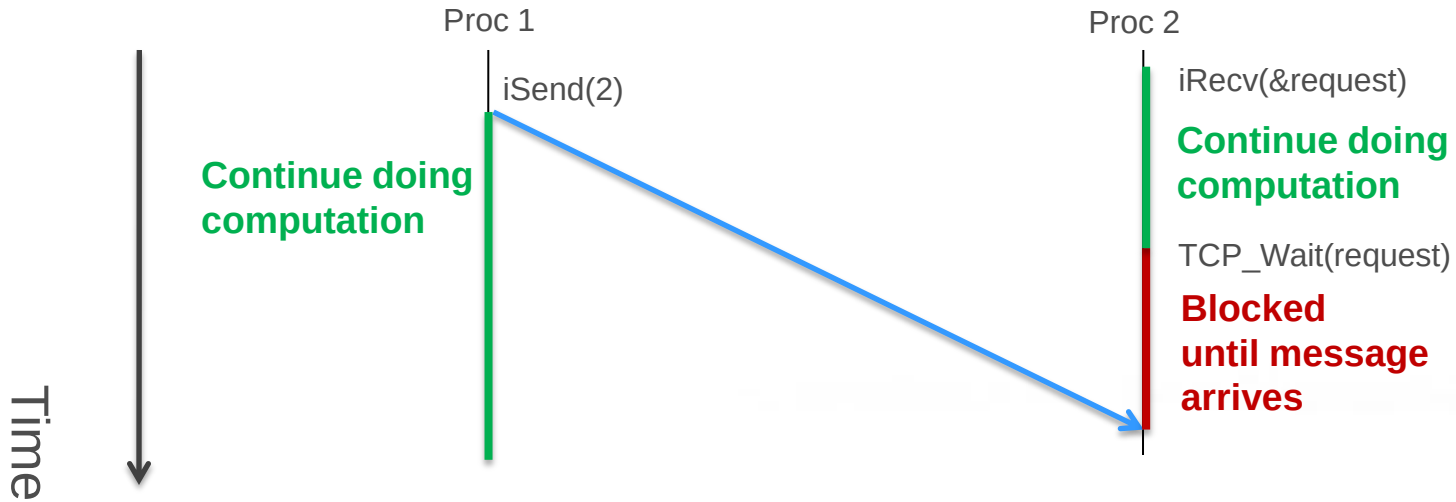
- **Deadlock**
  - **Can be solved proper ordering blocking calls**

```
if(my_Rank == 0){
    MPI_Recv(&tmp, 1, MPI_INT, 1, 122, MPI_COMM_WORLD, MPI_IGNORE_STATUS);
    MPI_Send(&dat, 1, MPI_INT, 1, 122, MPI_COMM_WORLD);
}

if(my_Rank == 1){
    MPI_Send(&dat, 1, MPI_INT, 0, 122, MPI_COMM_WORLD);
    MPI_Recv(&tmp, 1, MPI_INT, 0, 122, MPI_COMM_WORLD, MPI_IGNORE_STATUS);
}
```

MPI
One-to-one Communication

- **Assynchronous/Non-Blocking Receive**
  - Process tries to receive, returns if there is no message
  - Wait for the message is done using Wait



MPI Programming

- **Non-blocking Send and Receive**

```
int MPI_Isend(..., MPI_Request  &req)
```

- **req**
  - Reference to the communication request, holds information to use later

```
int MPI_Irecv(..., MPI_Request  &req, MPI_Status &status)
```

- **req**
  - Reference to the communication request, holds information to use later

MPI Programming

- **MPI_Wait**

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

  - **status**
    - Contains information about the received message can be ignored using **MPI_STATUS_IGNORE**
  - **req**
    - Reference to the communication request, holds information to use later

MPI Programming

- **MPI_Waitany**

```
int MPI_Waitany(int count, MPI_Status *status[], int *index,
MPI_Request  *req[])
```

  - **count**
    - Number process which are going to wait for
  - **status**
    - Array of statusus, to ignore use **MPI_STATUS_IGNORE**
  - **index**
    - Returns the index of the received request
  - **req**
    - Array of requests to wait for

MPI Programming

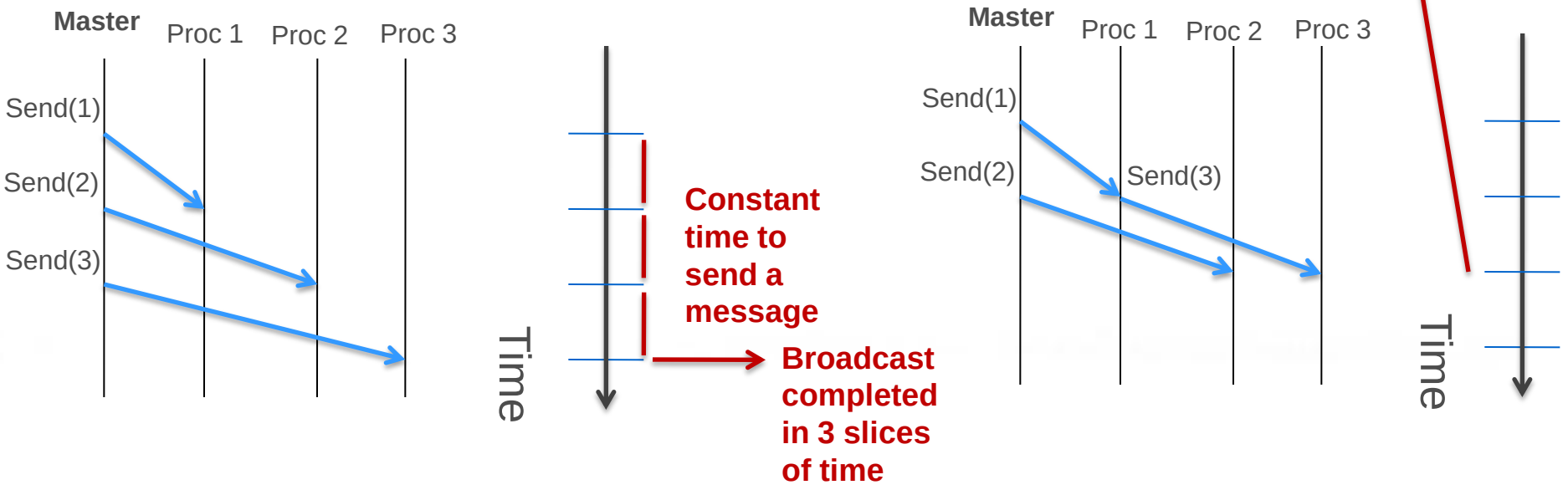
- **Exercise 7 – π Problem with MPI_Waitany**

  - **Use MPI_Isend and MPI_Irecv instead of blocking communication**

  - **Use MPI_Waitany to receive messages efficiently**

MPI Collective Communication

- **Proccess master wants to send a message to everybody**
  - First solution, process master send N-1 messages
  - Optimized collective communication send in parallel



MPI Programming

- **Broadcast**

```
int MPI_Bcast(void* buffer, int count, MPI_Datatype
datatype, int root , MPI_Comm coom)
```

  - **root**
    - If (myrank == root) send the content of buffer else not receive the content through buffer parameter
    - It is the first process to send messages to the others

MPI Programming

- **Exercise 8 –π Problem with MPI_Bcast**

MPI Programming

- **Collective reduce 1/2**

```
int MPI_Reduce(void* *sendbuf, void *recvbuf, int count,
MPI_Datatype  datatype, MPI_Op op, int root, MPI_Comm comm)
```

  - **sendbuf**
    - Data to be sent
  - **recvbuf**
    - Data to be received
  - **count**
    - Number of elements in sendbuf and recvbuf
  - **datatype**
    - Datatype of sendbuf and recvbuf elements

MPI Programming

- **Collective reduce 2/2**

```
int MPI_Reduce(void* *sendbuf, void *recvbuf, int count,
MPI_Datatype  datatype, MPI_Op op, int root, MPI_Comm comm)
```

  - **MPI_Op**
    - The arithmetic operation to execute some possible values: **MPI_MAX**, **MPI_MIN**, **MPI_SUM** , **MPI_PROD**, and so on **STATUS_IGNORE**
  - **root**
    - The same meaning as for MPI_Bcast

MPI Programming

- **Exercise 9 –π Problem with MPI_Reduce**

  - **Use MPI_Isend and MPI_Irecv instead of blocking communication**

  - **Use MPI_Waitany to receive messages in the efficientest order**

MPI Programming

- **Conclusion**
  - MPI is useful and easy to program
  - Well, at least easyer than sockets
  - Have many vendor and free implementations
  - It is optimized for different network architectures
  - What we saw:
    - Some of the MPI one-to-one and collective communication API
  - What we didn’t saw:
    - A lot of stuff, MPI_Scatter, MPI_Barrier, MPI_Get_processor_name, and so on...

MPI Programming

- **Future of MPI**
  - Exascale computing??
  - 10^18 Flops/s