Introduction to Distributed Memory Computing using Message Passing Interface (MPI)

Alper Kinaci, PhD
Sr. Computational Specialist
Research Computing Services
akinaci@northwestern.edu

Northwestern
INFORMATION TECHNOLOGY

Message Passing Interface (MPI)

- A standard library interface for coding on distributed memory systems
- Only message passing library that can be considered a standard
- It is supported on virtually all HPC platforms
- C, C++, Fortran and Python bindings
- Different implementations exist: OpenMPI, MPICH, Intel MPI etc.
- The programmer is responsible for correctly identifying parallelism and implementing parallel algorithms

Layout of an MPI Code

FORTRAN

```fortran
PROGRAM mpilayout
USE MPI ! f90 designation
! include "mpif.h" ! f77 designation
integer ierr


CALL MPI_INIT(ierr)


CALL MPI_FINALIZE(ierr)


END PROGRAM mpilayout
```

NON-MPI

MPI CALL

NON-MPI

C

```c
#include <mpi.h>

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


MPI_Init(&argc, &argv);


MPI_Finalize();


}
```

Hello World

FORTRAN

```fortran
program hello
implicit none

write(*,*) "Hello, world"

end program hello
```

MPI parallelized

```fortran
program hello_mpi
use mpi
implicit none

integer ierr

call MPI_INIT(ierr)
write(*,*)  "Hello, world"
call MPI_Finalize(ierr)

end program hello_mpi
```

C

```c
#include <stdlib.h>
#include <stdio.h>
int main(int argc, char **argv)
{
printf("Hello, world\n");
}
```

MPI parallelized

```c
#include <stdlib.h>
#include <mpi.h>
#include <stdio.h>
int main(int argc, char **argv)
{

MPI_Init(&argc, &argv);
printf("Hello, world\n");
MPI_Finalize();

}
```

Compilation/Running on Quest

- Check available MPI modules

> module avail mpi

- Load MPI module

> module load mpi/openmpi-1.8.6-gcc-4.8.3

- Compile using MPI

> mpif90 <source.f90> -o <binary.exe>

- Run the MPI code

> mpirun -np 2 <binary.exe>

Compilation on Quest

- Compiling with MPI

Fortran

```
> mpif90 <source.f90> -o <binary.exe>
```

C

```
> mpicc <source.c> -o <binary.exe>
```

C++

```
> mpic++ <source.cpp> -o <binary.exe>
```

A Simple MPI Code

- This code illustrates how to initialize & finalize the MPI environment and gain access to environmental variables.

FORTRAN

```fortran
program mpi_example1
use mpi
implicit none
integer ierr, np, rank, inum

call MPI_INIT(ierr)
call MPI_COMM_SIZE(MPI_COMM_WORLD, np, ierr)
call MPI_COMM_RANK(MPI_COMM_WORLD, rank, ierr)

if (rank == 0) then
write(*,*) 'I am master with rank ',rank
else
write(*,*) 'My rank is: ',rank
endif

call MPI_Finalize(ierr)

end program mpi_example1
```

C

```c
#include <mpi.h>
#include <stdio.h>
int main(int argc, char **argv){
int np, rank;

MPI_Init(&argc, &argv);
MPI_Comm_size(MPI_COMM_WORLD, &np);
MPI_Comm_rank(MPI_COMM_WORLD, &rank);

if (rank == 0){
printf("I am master with rank: %d\n",rank);
}
else {
printf("My rank is: %d\n",rank);
}

MPI_Finalize();
}
```

Initializing MPI

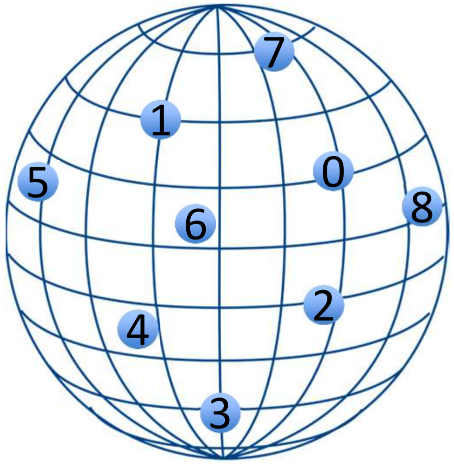
- MPI_Init
  - C: MPI_Init(&argc,&argv)
  - Fortran: MPI_INIT(ierr)
  - Initializes MPI environment
  - Must be called only once before any other MPI commands

C	FORTRAN
argc : pointer to the number of arguments (IN)	ierr : error return (OUT)
argv : pointer to the argument vector (IN)	

MPI Environment

C	MPI_Comm_size(MPI_COMM_WORLD, &size)
Fortran	MPI_COMM_SIZE (MPI_COMM_WORLD, size, ierr)

- Communicator: a structure in which we identify a group of processes
- Size: number of processes in a communicator
- MPI_COMM_WORLD: constant which includes the whole activated communicator processes in an instance



MPI_COMM_WORLD

MPI Environment

C	MPI_Comm_rank(MPI_COMM_WORLD, &rank)
Fortran	MPI_COMM_RANK(MPI_COMM_WORLD, rank, ierr)

- Rank: Within the communicator, each processes is assigned to an integer from 0 to size-1

C	MPI_Finalize()
Fortran	MPI_FINALIZE(ierr)

- Terminates MPI environment
- Last MPI routine to be called

Language Differences

Syntax for commands are not unified among different languages

1. Fortran has all uppercase (e.g., MPI_INIT) MPI commands, whereas C MPI routines have upper and lowercase (e.g., MPI_Init).
2. Error codes are returned in a separate argument for Fortran as opposed to the return value for C functions.
3. The arguments of some C MPI routines have specific data types such as MPI_Comm and MPI_Status whereas Fortran has integers.
4. The *include* files are different: in C, mpi.h, in Fortan, mpif.h.
5. The arguments to MPI_Init are different, so that a C program can take advantage of command-line arguments.
6. In Fortran MPI routines are subroutines and are invoked with call statement.

MPI Communication

- As understood from the name, MPI processes communicate by passing messages

Point-to-Point Communication

MPI Communication

Message
(data+envelope)

- A message contains the data and its envelope

DATA	ENVELOPE
Buffer, initial address	Communicator
Count	Source
Data type	Destination
	Tag

MPI Communication

Point-to-Point Communication

Message is passed from one process to another

- Blocking (MPI_Send, MPI_Receive)
- Non-blocking (MPI_Isend, MPI_Irecv)
- Send&Receive (MPI_Sendrecv)

Collective Communication

Message passes to all processes in a communicator

Point-to-Point Communication

- Blocking versus non-blocking:
  - Blocking routines do not return until it is safe to use the routine’s buffer (i.e. variables)
    - safe: the buffer has been copied to system or receiver buffer
  - Non-blocking routines do not wait for communication to complete
  - Non-blocking routines allow the overlap of computation and communication in order to gain performance

Point-to-Point Communication

- Deadlock: Message passing cannot be completed.

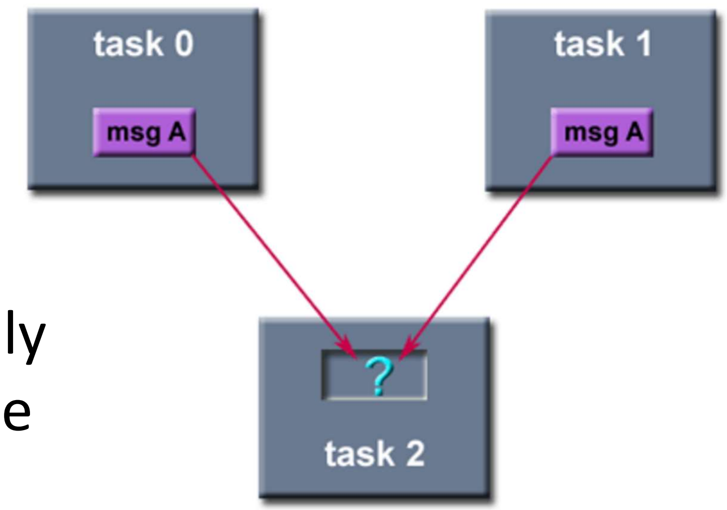
```
if (rank == 0) {
   MPI_Send(..., 1, tag, MPI_COMM_WORLD);
   MPI_Recv(..., 1, tag, MPI_COMM_WORLD, &status);
} else if (rank == 1) {
   MPI_Send(..., 0, tag, MPI_COMM_WORLD);
   MPI_Recv(..., 0, tag, MPI_COMM_WORLD, &status);
}
```

```
if (rank == 0) {
   MPI_Send(..., 1, tag, MPI_COMM_WORLD);
   MPI_Recv(..., 1, tag, MPI_COMM_WORLD, &status);
} else if (rank == 1) {
   MPI_Recv(..., 0, tag, MPI_COMM_WORLD, &status);
   MPI_Send(..., 0, tag, MPI_COMM_WORLD);
}
```

Point-to-Point Communication

- Order and Fairness
  - MPI keeps the order of send (or receive) requests from the same routine
  - MPI does not guarantee fairness

If 2 different tasks send messages that match another tasks receive, only one send will be complete



Point-to-Point Communication

```fortran
program mpi_example2
use mpi
implicit none
integer ierr, np, rank, message

call MPI_INIT(ierr)
call MPI_COMM_SIZE(MPI_COMM_WORLD, np, ierr)
call MPI_COMM_RANK(MPI_COMM_WORLD, rank, ierr)

if (rank == 0) then
    message = 48151623

    call MPI_SEND(message, 1, MPI_INTEGER, 1, 0,&
                  MPI_COMM_WORLD, ierr)

    write(*,*) "process ", rank, " sends ", message
else if (rank == 1) then
    call MPI_RECV(message, 1, MPI_INTEGER, 0, 0,&
                  MPI_COMM_WORLD, MPI_STATUS_IGNORE, ierr)

    write(*,*) "process ",rank," receives ", message

!else if (rank == 2) then
    !write(*,*) "process ",rank," receives ", message

endif

call MPI_Finalize(ierr)
end program mpi_example2
```

A blocking communication between 2 processes

Point-to-Point Communication

- MPI_SEND(buf,count,datatype,dest,tag,comm,ierr)
- MPI_Send(&buf,count,datatype,dest,tag,comm)

Data　　Envelope

Variable		Definition
buf	IN	Address of send buffer
count	IN	Number of elements in buf
datatype	IN	MPI data type of send buffer
dest	IN	Rank of destination
tag	IN	Message tag
comm	IN	Communicator
ierr	OUT	Return code

Mpi_example2

→ message
→ 1
→ MPI_INTEGER
→ 1
→ 0
→ MPI_COMM_WORLD
→ ierr

Point-to-Point Communication

- MPI_RECV(buf,count,datatype,source,tag,comm,ierr)
- MPI_Recv(&buf,count,datatype, source,tag,comm,&status)

Data Envelope

Variable		Definition
buf	OUT	Address of receive buffer
count	IN	Number of elements in buffer
datatype	IN	MPI data type of receive buffer
source	IN	Rank of source
tag	IN	Message tag
comm	IN	Communicator
ierr	OUT	Return code
status	OUT	Status object (sender rank, tag, length)

Mpi_example2

→ message

→ 1

→ MPI_INTEGER

→ 1

→ 0

→ MPI_COMM_WORLD

→ ierr

MPI Data Types

Generally MPI data types and language types have one-to-one correspondence however in C, some MPI variables have their own data types such as MPI_Comm, MPI_Status etc.

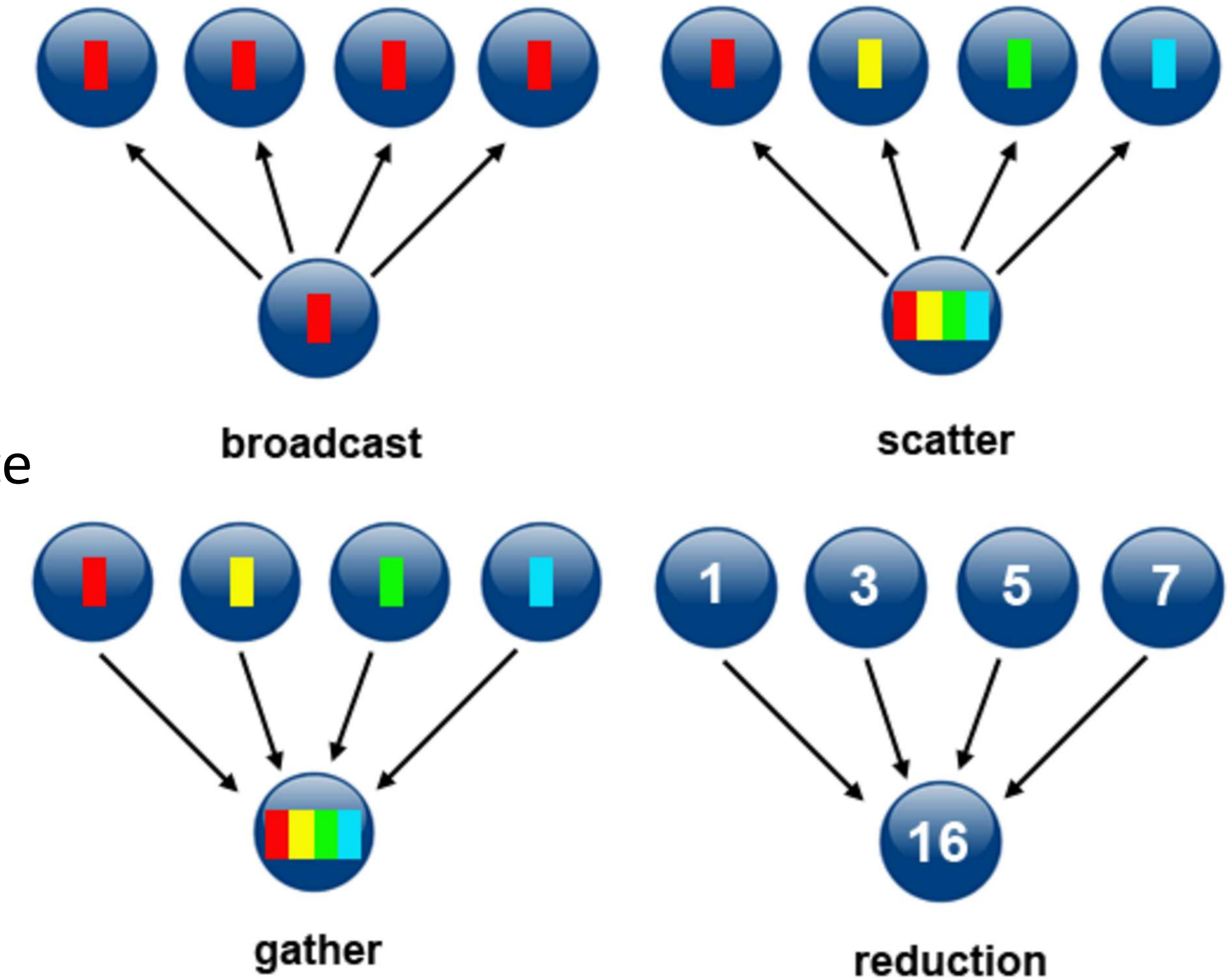
Data type	C/C++
MPI_CHAR	char
MPI_SHORT	short int
MPI_INT	int
MPI_LONG	long int
MPI_FLOAT	float
MPI_DOUBLE	double
MPI_LONG_DOUBLE	long double

Data type	Fortran
MPI_CHARACTER	character
MPI_INTEGER	integer
MPI_REAL	Real*4
MPI_REAL8	Real*8
MPI_COMPLEX	Complex
MPI_LOGICAL	logical

Collective Communication

<u>Types of Collective Communications</u>

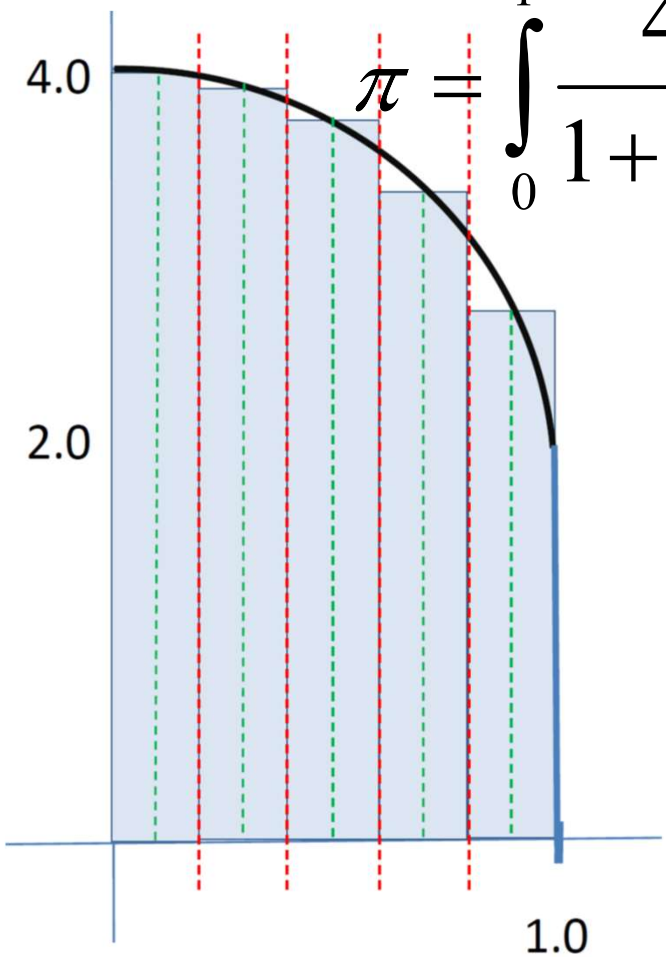
- Synchronization
- Data Movement
- Collective Compute



Collective Communication

MPI_BARRIER	All processes within a communicator will be blocked until all processes within the communicator have entered the call.
MPI_BCAST	Broadcasts a message from one process to members in a communicator.
MPI_REDUCE	Performs a reduction operation to the vector of elements in the sendbuf of the group members and places the result in recvbuf on root.
MPI_GATHER MPI_GATHERV	Collects data from the sendbuf of all processes in comm and place them consecutively to the recvbuf on root based on their process rank.
MPI_SCATTER MPI_SCATTERV	Distribute data in sendbuf on root to recvbuf on all processes in comm.
MPI_ALLREDUCE	Same as MPI_REDUCE, except the result is placed in recvbuf on all members in a communicator.
MPI_ALLGATHER MPI_ALLGATHERV	Same as GATHER/GATHERV, except now data are placed in recvbuf on all processes in comm.
MPI_ALLTOALL	The j-th block of the sendbuf at process i is send to process j and placed in the i-th block of the recvbuf of process j.

Collective Communication



$$\pi = \int_0^1 \frac{4}{1+x^2}dx$$

Calculating Pi (Serial)

```
PROGRAM compute_pi
implicit none
integer i,n
double precision pi,h,x

n=900000000
pi = 0.0D+0
h = 1.0D+0/REAL(n)    !trapezoid base

do i=1,n
     x = h*(REAL(i)-0.5D+0)
     pi = pi+ (4.0D+0/(1.0D+0 + x*x))
end do

pi=pi*h
write(*,*) pi

END PROGRAM compute_pi
```

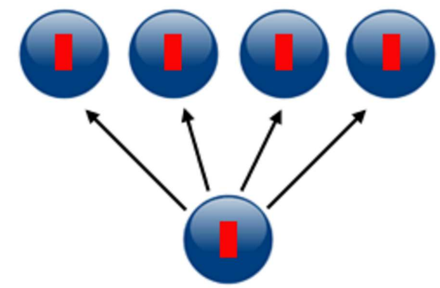
Calculating Pi (MPI)

```
PROGRAM pi_mpi
use mpi
double precision mypi, pi, h, x, f, a
integer n, myid, numprocs, i, ierr

call MPI_INIT( ierr )
call MPI_COMM_RANK( MPI_COMM_WORLD, myid, ierr )
call MPI_COMM_SIZE( MPI_COMM_WORLD, numprocs, ierr )

if (myid .eq. 0) n = 900000000

call MPI_BCAST(n,1,MPI_INTEGER,0,MPI_COMM_WORLD,ierr)

h = 1.0D+0 / REAL (n) ! trapezoid base
mypi = 0.0D+0

do i = myid+1, n, numprocs ! cyclic distribution
    x = h * ( REAL (i) - 0.5D+0)
    mypi = mypi + 4.0D+0 / (1.0D+0 + x * x)
end do
!!!!!collect partial sums!!!!!!!!!!!!!!!
call MPI_REDUCE(mypi, pi, 1, MPI_REAL8, &
    MPI_SUM, 0, MPI_COMM_WORLD,ierr)

if (myid .eq. 0) then
    pi=pi*h
    write(*,*) pi
end if

call MPI_FINALIZE(ierr)
end program pi_mpi
```

Collective Communication

MPI_Bcast (&inbuf,count,datatype,root,comm)

MPI_BCAST(inbuf, incnt, intype, root, comm, ierr)



Broadcasts a message from one process to members in a communicator

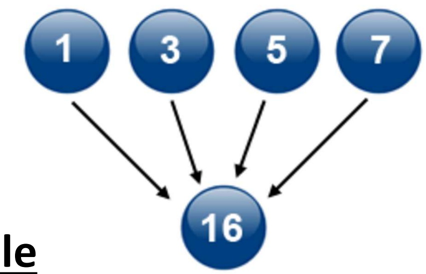
Variable		Definition	Pi example
inbuf	IN/OUT	Address of input buffer	n
incnt	IN	Number of elements in inbuf	1
intype	IN	MPI data type of inbuf	MPI_INTEGER
root	IN	Process id of root process	0
comm	IN	Communicator	MPI_COMM_WORLD

Collective Communication

MPI_Reduce (&inbuf,&outbuf,count,datatype,op,root,comm)

MPI_REDUCE(inbuf, outbuf, count, datatype, op, root, comm, ierr)

Performs a reduction operation to the vector of elements in the sendbuf of the group members and places the result in recvbuf on root



reduction

Variable		Definition		Pi example
inbuf	IN	Address of send buffer	→	mypi
outbuf	OUT	Address of receive buffer	→	pi
count	IN	Number of elements in inbuf	→	1
type	IN	MPI data type of send buffer	→	MPI_REAL8
op	IN	Operation (+,-,x,max,min, …)	→	MPI_SUM
root	IN	Process id of root process	→	0
comm	IN	Communicator	→	MPI_COMM_WORLD

References

- https://computing.llnl.gov/tutorials/mpi/
- https://computing.llnl.gov/tutorials/openMP
- http://sc.tamu.edu/shortcourses/SC-MPI/mpi_shortcourse_v4.pdf

MPI for Python
MPI4PY

Layout of an Python Code with mpi4y

```
#!/usr/bin/env python
"""
Simple MPI layout
"""

from mpi4py import MPI
```

MPI REGION

- MPI module in mpi4py package contains the routines used for parallelization

Running on Quest

- Load MPI module

> module load mpi4py

- Run the MPI code

> mpirun -np 2 python <script.py>

Hello World

```python
#!/usr/bin/env python
"""
Hello World, serial
"""

import sys

sys.stdout.write("Hello, World!"+"\n")
```

MPI parallelized

```python
#!/usr/bin/env python
"""
Hello World, parallel
"""

from mpi4py import MPI
import sys

sys.stdout.write("Hello, World!"+"\n")
```

A Simple Python MPI Code

```
#!/usr/bin/env python
"""
Hello World, parallel
"""

from mpi4py import MPI
import sys

size = MPI.COMM_WORLD.Get_size()
rank = MPI.COMM_WORLD.Get_rank()
name = MPI.Get_processor_name()

sys.stdout.write(
    "Hello, World! I am process %d of %d on %s.\n"
    % (rank, size, name))
```

Point-to-Point Communication

- Usual communication methods are included in Comm class of MPI module
- Comm.send(buf, dest,tag)

$\underbrace{\text{(buf,}}_{\text{Data}}\underbrace{\text{dest,tag)}}_{\text{Envelope}}$

Variable	Definition
buf	Address of send buffer
dest	Rank of destination
tag	Message tag

Point-to-Point Communication

- Comm.recv($\boxed{\text{buf}}$,$\boxed{\text{source,tag,}}$status)

Data Envelope

Variable	**Definition**
buf	Address of receive buffer
source	Rank of source
tag	Message tag
status	Status object (sender rank, tag, length)

Communicating Objects & Arrays

- mpi4py provides two sets of functions for communication:
  - All lower case methods (send, recv etc.) are for communicating generic python data objects. Can be slow.
  - Upper-case initial letter case (Send, Recv etc.) are for communicating array data (such as NumPy arrays) which occupies continuous memory blocks. Fast communication.