Theory and Practice of MPI Programming

In-Ho Lee

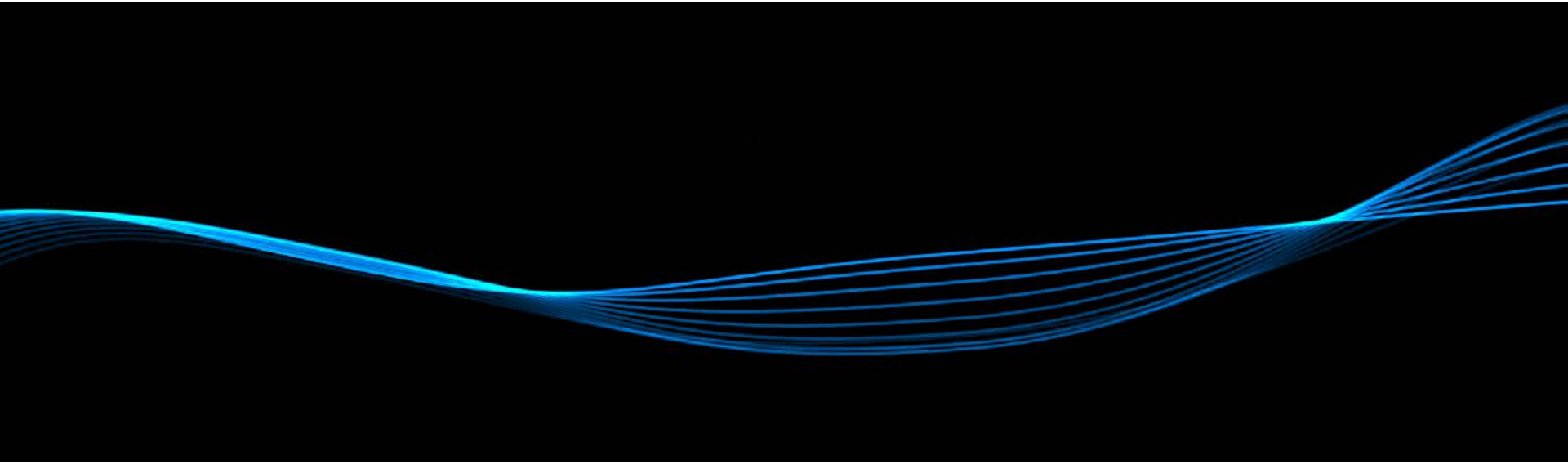
Korea Research Institute of Standards and Science, Daejeon 34113, Republic of Korea

KIAS, June 26--June 30, 2023

The 14th KIAS CAC Summer School

June 26
10:00—12:20
June 27
9:30—10:30

자연과학을 전공하는 대학원생 또는 연구자를 대상으로, 병렬 컴퓨팅, GPU 사용 기법, 그리고 인공지능 방법들에 대한 강의와 실제 연구에의 적용을 위한 발표를 제공하고자 합니다. 참가자들은 또한 실제 클러스터 컴퓨터와 GPU 를 네트워크를 이용하여 직접 접속하고, 자신의 프로그램을 실행시켜 보게 됩니다. 이를 통하여 자신의 문제를 어떻게 적용할 것인지에 대한 도움을 얻을 수 있는 여름학교를 가지고자 합니다.



The 14th KIAS CAC Winter School

education demand survey

공유메모리 다중처리

Fortran 90 — OpenMP

C — OpenMP

Python — multiprocessing

MPI

많은 경우, “포트란 + MPI”, “C + MPI”, “C++ +MPI” 형태의 컴퓨터 프로그램이 필요하다.
상황에 따라서, “파이썬 + MPI” 도 필요하다.

Goal

- Parallel computing
- MPI based parallel computing
- Practical applications

효율적이고 범용적인 병렬 계산의 배경과 기본 원리를 이해한다.
실습을 통해서 병렬 문제 양식을 알아낸다. 문제형식 구별 능력을 함양한다.
다양한 병렬 컴퓨팅 적용 사례들을 습득한다.
최종적으로 각자 연구에 적극 적용한다.

기본적인 MPI 프로그램 방식은 크게 몇 가지로 분류된다. 자신이 풀고자 하는 현재의 문제가, 병렬계산 관점에서, 어떠한 부류에 속하는지를 스스로 먼저 파악 할 수 있어야 한다.

- Coarse-grained
- Fine-grained

Contents

Brief introduction to MPI

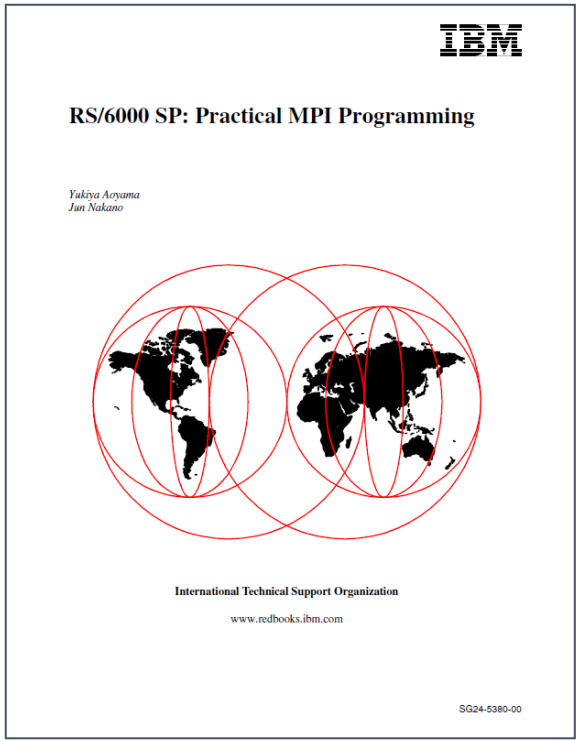
MPI basics

MPI practice

It’s neither complicated nor bloated.

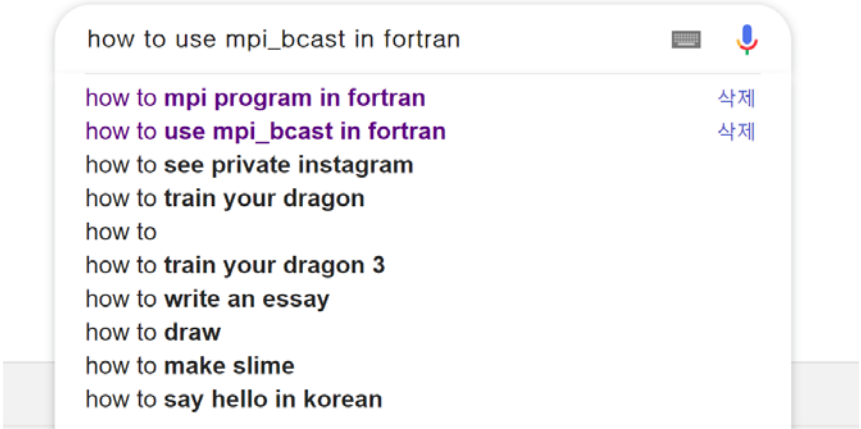
Reference

가장 중요한 참고서적

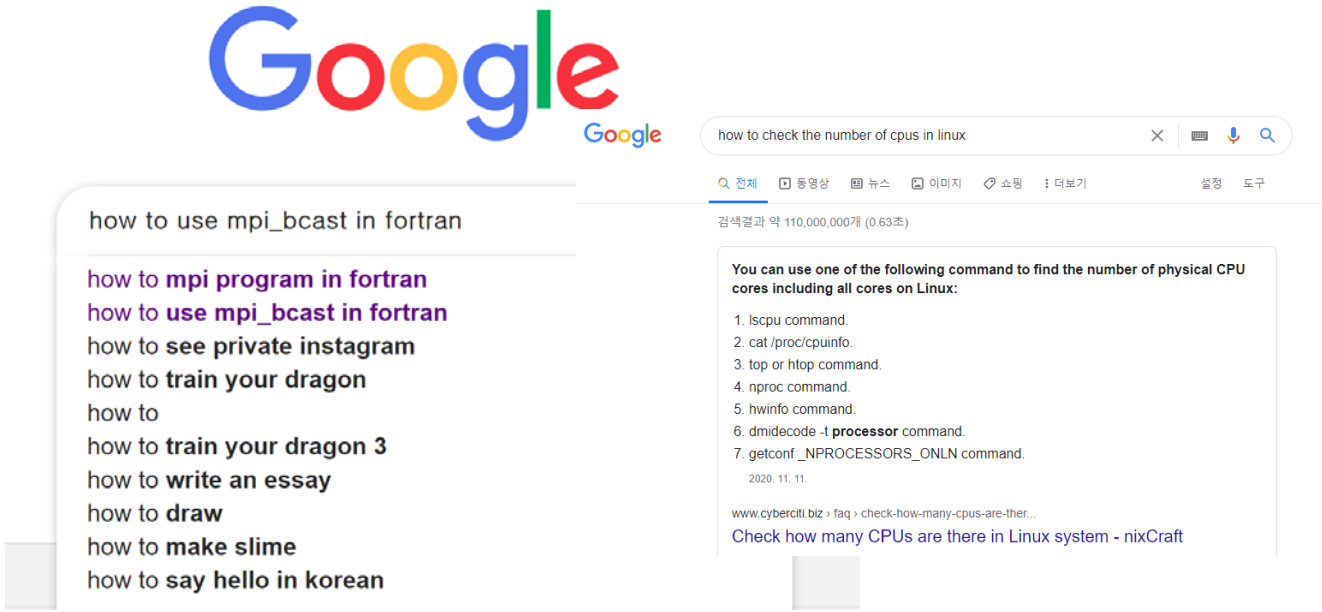


Y. Aoyama and J. Nakano, 1999

ChatGPT



http://en.wikipedia.org/wiki/Parallel_computing



많은 경우, 코드 병렬화에 방법은 인터넷에 공개되어 있다.
하지만, 인터넷에 공개된 해답들을 완전히 자기 것으로 만들기 위해서는 기본 지식과 응용 연습이 필요하다.

stackoverflow

24 million questions/35 million answers in 2023

github.com

PuTTy + Xming

윈도우즈로부터 리눅스 컴퓨터에 접속해서 일을 한다.
파일전송, 리눅스에서 계산 그리고 그림 그리기를 수행할 때 필요한 소프트웨어가 있다.

PuTTy 단순 연결: 원격 접속으로 문자 위주의 리눅스 명령어 사용 가능함. 파일 편집이 가능함.
PuTTy 복잡 연결: PuTTy [무료 소프트웨어] + Xming [무료 소프트웨어] 함께 사용하는 방법: X11 전달 사용 가능함. "그림 그리기"
보다 더 일반적인 연결, 적극 권장함. 그림을 띠워서 볼 수 있음.

다시 말하면 PuTTy 프로그램만 단독으로 사용 할 수도 있습니다. [윈도우 → 원격 리눅스 시스템 접속 가능하다.]
마찬가지로 PuTTy는 Xming 과 함께 사용할 수도 있습니다.
[윈도우즈에 Xming을 실행한 상태에서 PuTTy를 이용한 원격 리눅스 접속이 가능하다.]

윈도우에 아래와 같은 두 개 프로그램의 설치가 필요합니다.

PuTTy 설치 → Xming 설치 → Xming 실행 → PuTTy 실행 → PuTTy 미세조정 [두 가지 모두 무료 소프트웨어]

윈도우즈에서 Xming 실행 함 → 윈도우즈가 X11 그림을 받을 준비함. PC가 전달받을 준비 완료. 아이콘 트레이에 Xming 활성화를 확인한다.
[시작 → 모든 프로그램 → Xming, Xming과 XLaunch를 확인할 수 있다.
Xming은 실제 Xming을 실행시키는 프로그램이고, XLaunch는 화면 설정을 해주는 것이다.
사용자가 만들어 놓은 설정을 저장할 수 있다.]

ls

ls –ltra

예를 들어 프로그램 실행 전후,
파일들의 생성 순서가 중요할 경우

mkdir abc
cd abc
ls

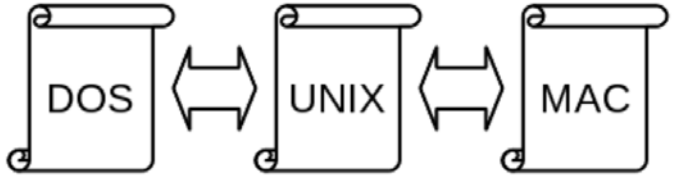
mv aa aa1

cp aa bb

Editors
vi
emacs
pico

:set showmatch
:set paste
:syntax off

Dos2Unix / Unix2Dos - Text file format converters



Convert text files with DOS or Mac line breaks to Unix line breaks and vice versa.

1) Ctrl + c
2) 우클릭 → 복사
1) Ctrl + v
2) 우클릭 →붙여넣기

1) Meta + c
2) 우클릭 → 복사
1) Meta + v
2) 우클릭 → 붙여넣기

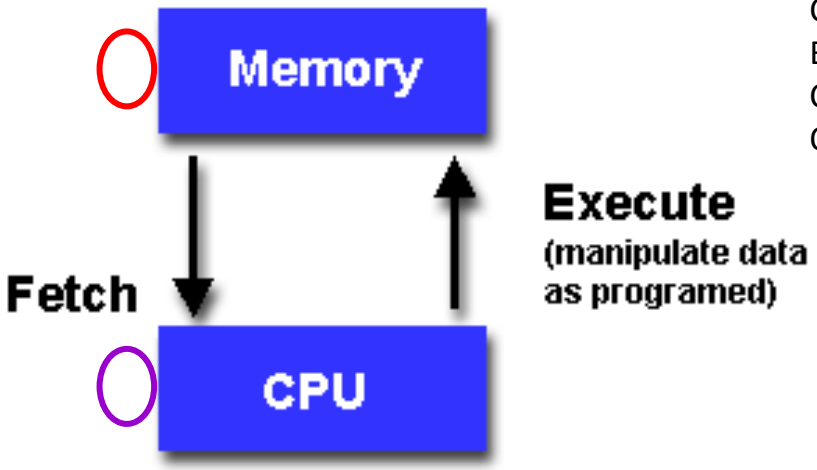
https://www.cyberciti.biz/faq/howto-unix-linux-convert-dos-newlines-cr-lf-unix-text-format/

텍스트 복사할 때 제대로 문자들 배열이 되도록 해주는 옵션

Supercomputer?
Why Parallel Computing?

- Large-scale calculations
- Fast calculations (optimization?)
- www.top500.org
- Save time - ***wall clock time*** (1 day *vs* 1 week)
- speed of light (30 cm/nanosecond)
- Performance/price

von Neumann computer




```
$ lscpu
Architecture:          x86_64
CPU op-mode(s):        32-bit, 64-bit
Byte Order:            Little Endian
CPU(s):                40
On-line CPU(s) list:   0-39
```

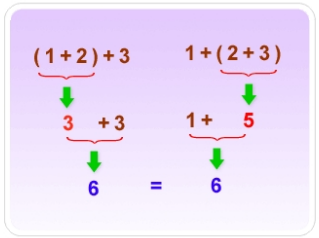
CPU gets instructions and/or data from memory,
decodes the instructions and then *sequentially* performs them.

Serial ←→ Parallel

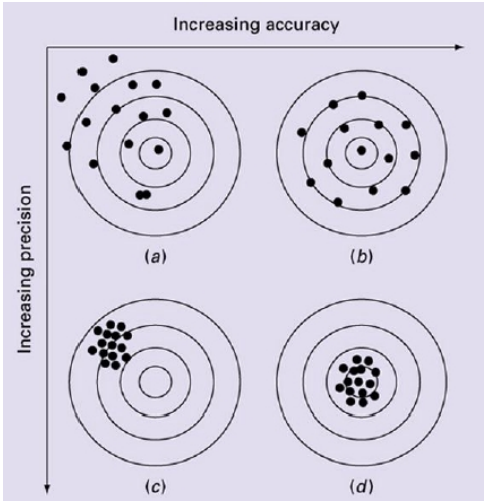
associative property of addition
associative property of multiplication
commutative distributive

SP : 10^{-38} 10^{39}
7 decimal precision
DP : 10^{-308} 10^{308}
15 decimal precision

(1+2)+3 = 1+(2+3)



Truncation Error

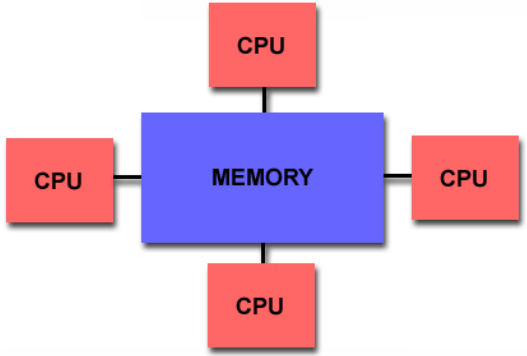


계산하는 순서가 다를 수 있다.
심지어 계산하는 CPU가 서로 다르다.

- Accuracy: how closely a computed or measured value agrees with the true values.
- Precision: how closely individual computed or measured values agree with each other.

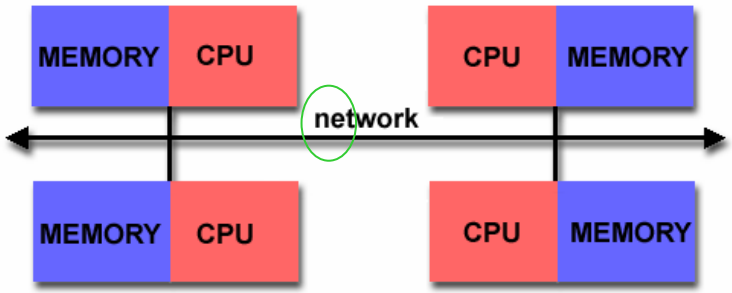
CPU & Memory

Shared Memory



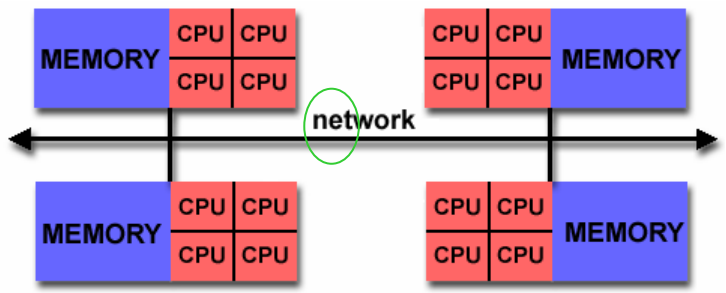
IBM SP Power3 Nodes, Regatta (Power4) servers,
SGI Power Challenge Series, Sun Enterprise Series

Distributed Memory



IBM SP (pre-Power3 nodes,
Clusters of uniprocessor nodes

Hybrid Distributed-Shared Memory



Clusters of SMPs,
IBM SP Series,
SGI Origin Series

Levels of parallelism

Data parallel (*fine-grained* parallel)

OpenMP and HPF Directive-based data-parallel

Parallel execution of DO loops in FORTRAN

Task parallel (*coarse-grained* parallel)

MPI, PVM (message passing)

Inter-process communication

Python → multiprocessing

Very general model

Hardware platforms

Great control over data location and flow in a program

Higher performance level (scalability)

Programmer has to work hard to implement!

Some General Parallel Terminology

- **Observed Speedup** (WC-serial/WC-parallel) **Scalability**
- **Granularity** (coarse, fine)

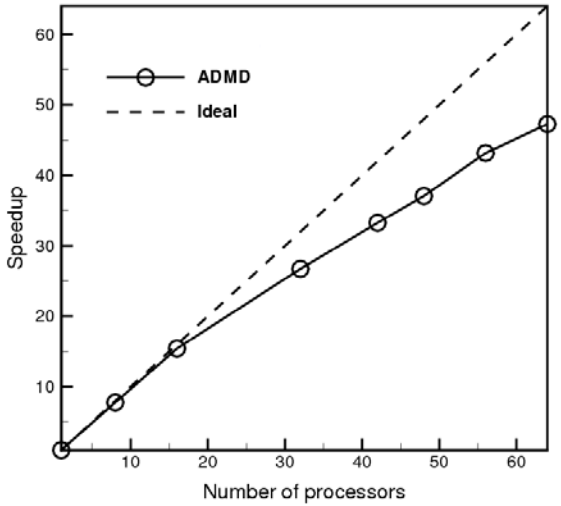
  Computation / Communication Ratio
- **Synchronization**
- **Communications**

The purpose of parallelization is to reduce the time spent for computation. Ideally, the parallel program is *p times faster than the sequential program*, where *p is the number of processes involved in the parallel execution, but this* is not always achievable.

speedup $S_p = \frac{T_1}{T_p}$

***p* is the number of processors.**
***T_p* is the execution time of the algorithm with *p* processors.**

<u>Not CPU time</u>, but wall-clock time reference

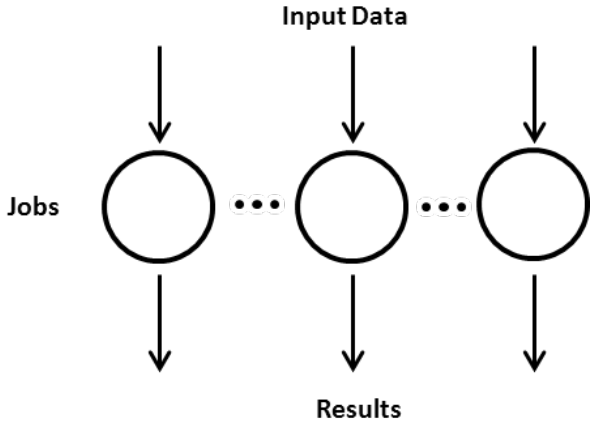


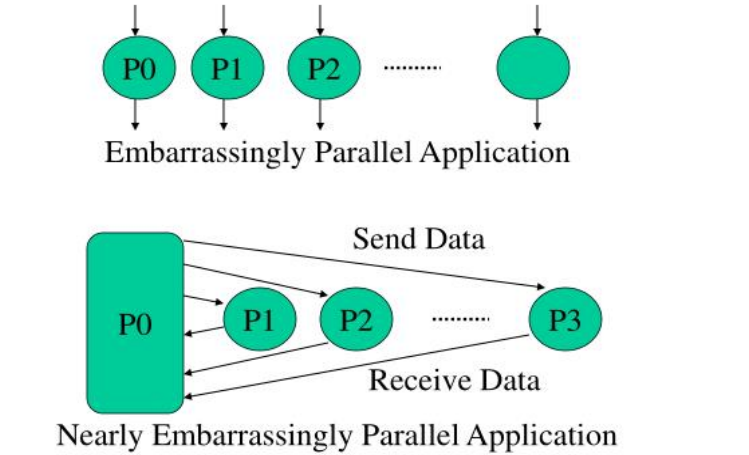
http://en.wikipedia.org/wiki/Speedup

Embarrassingly parallel

- little or no communication of results between tasks
  * Rendering of computer graphics
  * Genetic algorithms, Particle swarm optimization

여러 개의 폴더를 만들어서 처리하는 방법
Master-slave mode로 처리하는 방법





Embarrassingly parallel

덩어리 단위로 계산해야 할 경우, 하나의 덩어리 계산이 굉장한 컴퓨터 자원이 필요할 경우:
스크립트를 만들어서 배치로 일들을 처리한다.
계산마다 폴더를 만들어서 독립적인 일 처리를 수행한다.

Open MPI

Language	Command
C	`$ mpicc <src_file> -o <name_of_executable>`
C++	`$ mpicxx <src_file> -o <name_of_executable>`
Fortran	`$ mpifort <src_file> -o <name_of_executable>`

Intel MPI

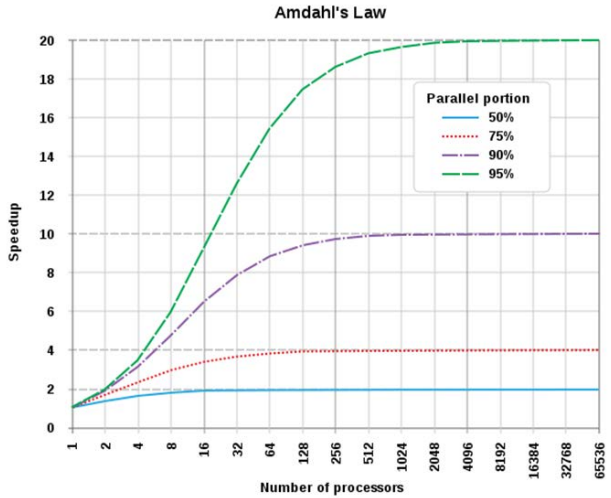
Compiler Driver	C	C++	Fortran
GCC	`$ mpicc <src_file> -o <name>`	`$ mpicpc <src_file> -o <name>`	`$ mpifort <src_file> -o <name>`
Intel	`$ mpiicc <src_file> -o <name>`	`$ mpiicpc <src_file> -o <name>`	`$ mpiifort <src_file> -o <name>`

```
[cac001@Lex04:~]$ ls /usr/bin/mp*
/usr/bin/mpic++          /usr/bin/mpic++.openmpi   /usr/bin/mpif77          /usr/bin/mpifort.openmpi
/usr/bin/mpicc           /usr/bin/mpicxx           /usr/bin/mpif77.openmpi  /usr/bin/mpirun
/usr/bin/mpiCC           /usr/bin/mpicxx.openmpi   /usr/bin/mpif90          /usr/bin/mpirun.openmpi
/usr/bin/mpicc.openmpi   /usr/bin/mpiexec          /usr/bin/mpif90.openmpi
/usr/bin/mpiCC.openmpi   /usr/bin/mpiexec.openmpi  /usr/bin/mpifort
```

Amdahl's law

The speedup is limited by the serial part of the program. For example, if 95% of the program can be parallelized, the theoretical maximum speedup using parallel computing would be 20 times.

병렬화 방법의 중요성.
병렬화를 시키는 방법들을 고려함.



Amdahl's law can be formulated the following way:

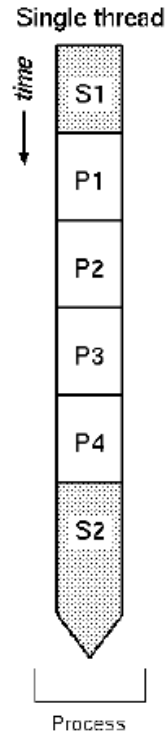
$$S_{\text{latency}}(s) = \frac{1}{(1-p) + \frac{p}{s}}$$

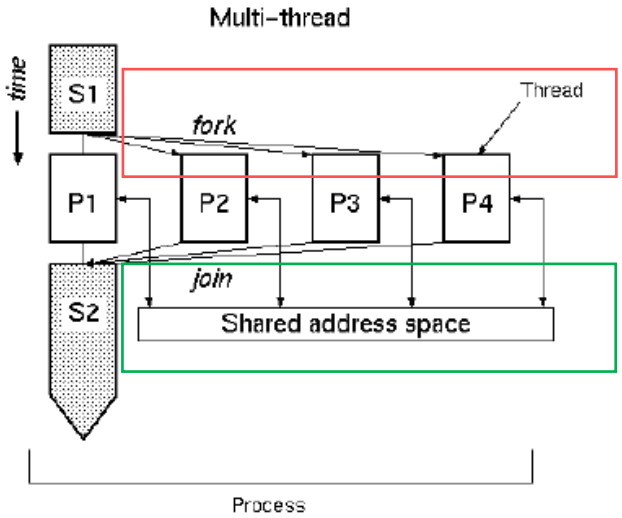
where

- S_{latency} is the theoretical speedup of the execution of the whole task;
- s is the speedup of the part of the task that benefits from improved system resources;
- p is the proportion of execution time that the part benefiting from improved resources originally occupied.

https://en.wikipedia.org/wiki/Amdahl%27s_law

SMP (Symmetric Multi-Processor)

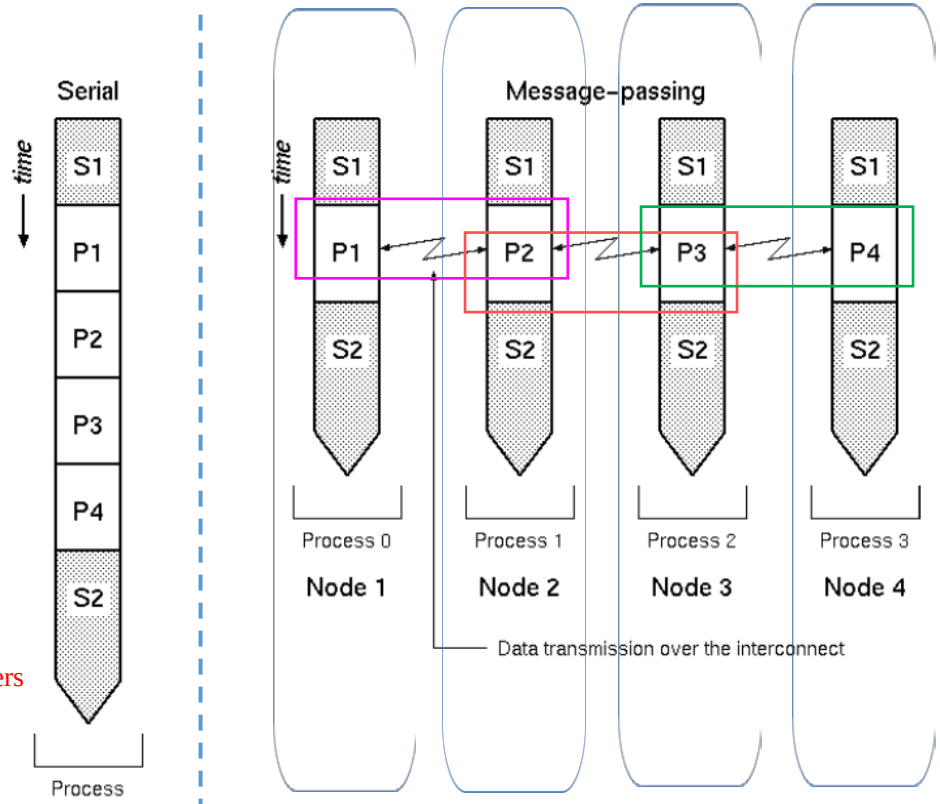




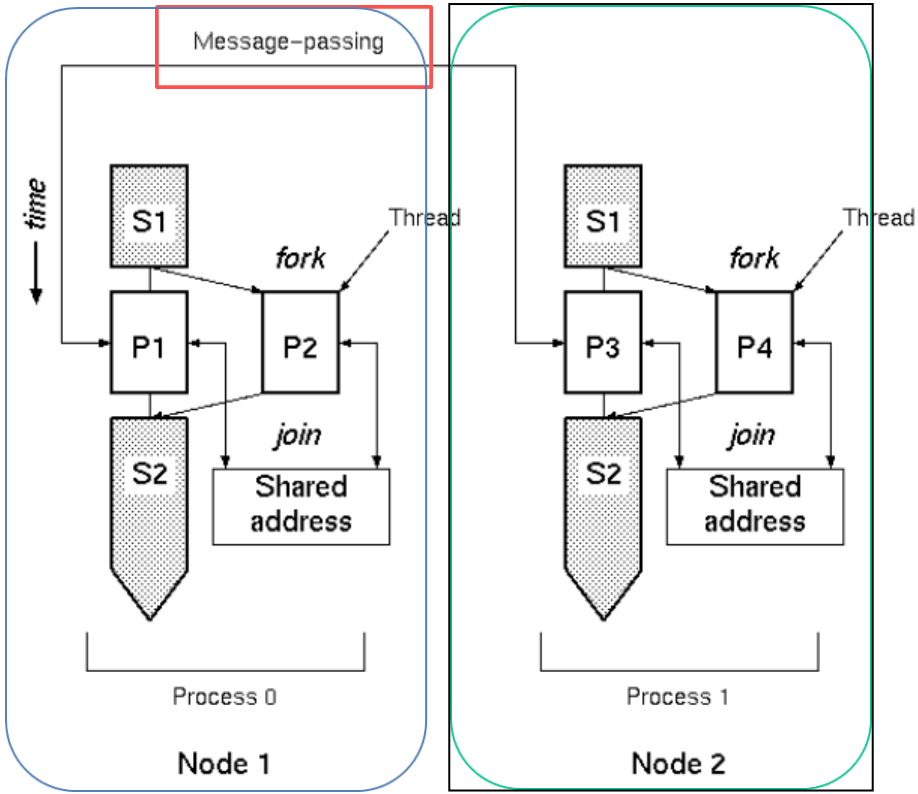
* Cray YMP C/J 90
* SGI Origin 200/2000
* Sun E10000
* SMP Intel-based computers (Dell, Unisys)

Python → multiprocessing

Message-Passing

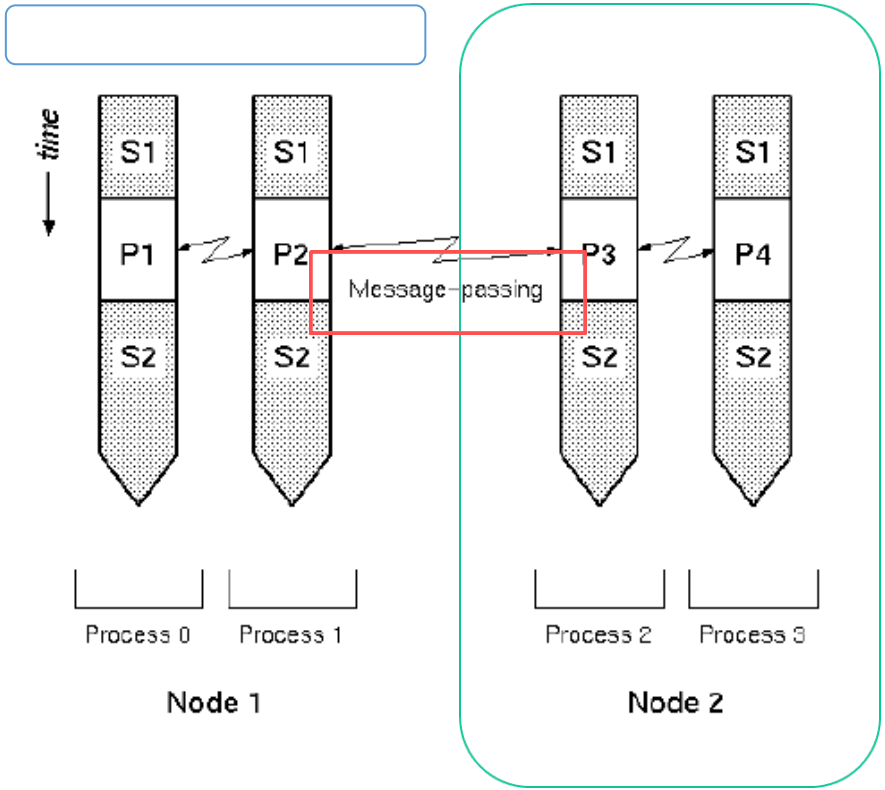


Cray T3E
Unix (Linux) clusters



Y. Aoyama and J. Nakano

Multiple Single-Thread Processes Per Node



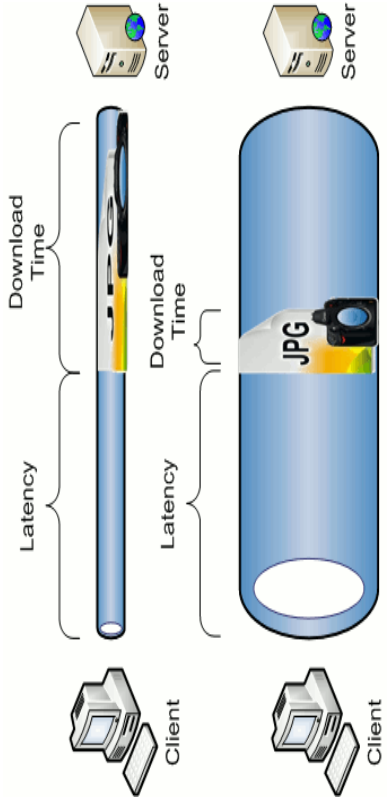
MPI (message passing interface)

- MPI 자체는 병렬 라이브러리들에 대한 표준규약이다. (125개의 서브 프로그램들로 구성되어 있다.) MPI는 약 40개 기관이 참여하는 MPI forum에서 관리되고 있으며, 1992년 MPI 1.0 을 시작으로 현재 MPI 4.0까지 버전 업데이트된 상태이다.
- Various architectures
- Free implementations: MPICH, LAM, CHIMP, openmpi

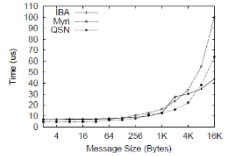
FLUENT (Fluid Dynamics and Heat transfer)
GASP (Gas Dynamics)
AMBER (Molecular dynamics)
NAMD (Molecular dynamics)
GROMACS (Molecular dynamics)
VASP (Electronic structure)
….

mpiexec --version

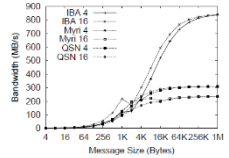
Latency and Bandwidth



- ***latency*** is the time it takes to send a minimal (0 byte) message from point A to point B. ~120, ~120, ~7, ~0.5 microsecond



- ***bandwidth*** is the amount of data that can be communicated per unit time. ~100 Mbps, ~1 Gbps, ~1.98 Gbps, ~1 Gb/sec



큰 스루풋, 작은 레이턴시

cf. CPU time

Fast ethernet, Gigbit ethernet, Myrinet, Quadrics 2, InfiniBand

Computation *vs* communication

연결을 위해서 기본적으로 걸리는 시간 (잠복, 잠재)

Communication time ← latency, throughput

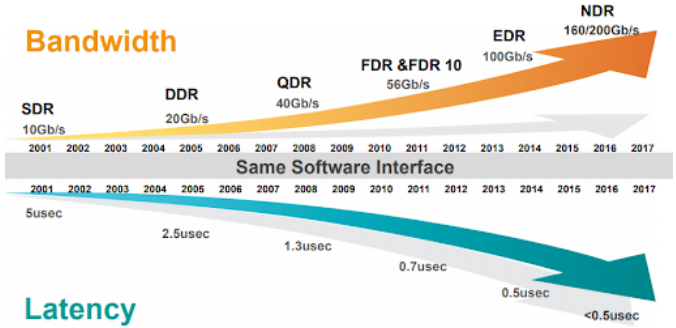
연결이 된 후 할당된 데이터 전송에 걸리는 시간 : 단위시간당 처리량

Latency를 줄이기 위해서는 한 번의 연결에 집중적으로 통신하는 것이 좋다.

데이터 통신에 걸리는 시간 *vs* CPU를 이용한 계산시간

>>

InfiniBand is a computer-networking communications standard used in high-performance computing that features very high throughput and very low latency.



	Fast Ethernet	Gigabit Ethernet	Myrinet	Quadrics 2
latency	120 microsecond	120 microsecond	7 microsecond	0.5 microsecond
bandwidth	100 Mbps	1 Gbps	1.98 Gbps	1 Gbyte/second

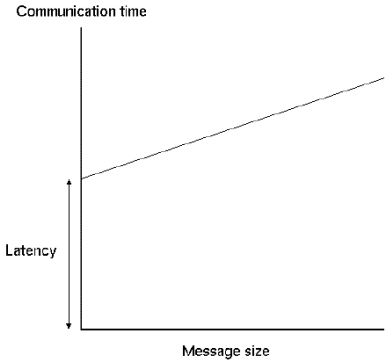


Figure 32. The Communication Time

	SDR	DDR	QDR	FDR10	FDR	EDR	HDR	NDR	XDR
Signaling rate (Gbit/s)	2.5	5	10	10.3125	14.0625[6]	25	50	100	250
Theoretical effective throughput, Gbs, per 1x[7]	2	4	8	10	13.64	24.24			
Speeds for 4x links (Gbit/s)	8	16	32	40	54.54	96.97			
Speeds for 8x links (Gbit/s)	16	32	64	80	109.08	193.94			
Speeds for 12x links (Gbit/s)	24	48	96	120	163.64	290.91			
Encoding (bits)	8/10	8/10	8/10	64/66	64/66	64/66	64/66		
Adapter latency (microseconds)[8]	5	2.5	1.3	0.7	0.7	0.5			
Year[9]	2001, 2003	2005	2007	2011	2011	2014[7]	2017[7]	after 2020	future

InfiniBand

SPMD

Start parallel job on N processors

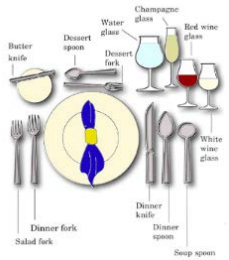
g++ parallel_hello_world.cpp -o parallel_hello_world.exe –fopenmp
icc parallel_hello_world.cpp -o parallel_hello_world.exe -qopenmp

Message passing between processors

Each processor does some calculations (*processor dependent codes are run*)

Message passing between processors

"좌빵우물"



End parallel job

/usr/local/mpich/

bin/ etc/ include/ man/ share/
doc/ examples/ lib/ sbin/ www/

/usr/local/mpich/bin/mpif90 a.f90

mpirun –np 8 a.out

mpif90 hello_world_mpi.f90 -o hello_world_mpi.exe
mpiifort hello_world_mpi.f90 -o hello_world_mpi.exe

mpirun -np 4 ./hello_world_mpi.exe

The 6 basic MPI routines

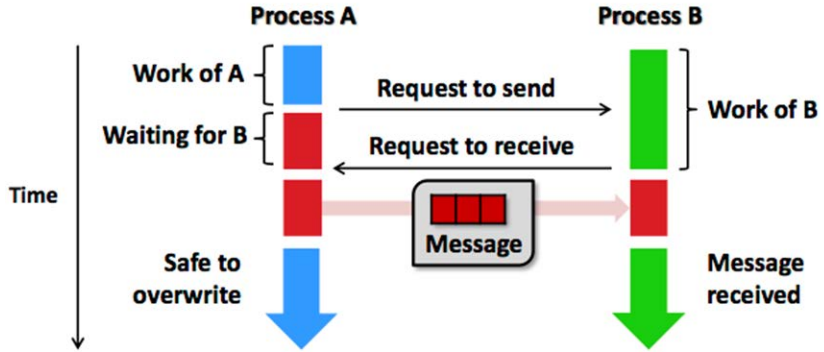
- MPI is used to create parallel programs based on message passing
- Usually the same program is run on multiple processors
- The 6 basic calls in MPI are:
  - MPI_INIT(ierr)
  - MPI_COMM_RANK(MPI_COMM_WORLD, ***myid***, ierr)
  - MPI_COMM_SIZE(MPI_COMM_WORLD, ***numprocs***, ierr)
  - MPI_Send(**buffer**, ***count***, MPI_INTEGER, *destination*, tag, MPI_COMM_WORLD, ierr)
  - MPI_Recv(**buffer**, ***count***, MPI_INTEGER, *source*, tag, MPI_COMM_WORLD, status, ierr)
  - call MPI_FINALIZE(ierr)

```
[cac001@Lex04:~]$ ls /usr/bin/mp*
/usr/bin/mpic++          /usr/bin/mpic++.openmpi   /usr/bin/mpif77          /usr/bin/mpifort.openmpi
/usr/bin/mpicc           /usr/bin/mpicxx           /usr/bin/mpif77.openmpi  /usr/bin/mpirun
/usr/bin/mpiCC           /usr/bin/mpicxx.openmpi   /usr/bin/mpif90          /usr/bin/mpirun.openmpi
/usr/bin/mpicc.openmpi   /usr/bin/mpiexec          /usr/bin/mpif90.openmpi
/usr/bin/mpiCC.openmpi   /usr/bin/mpiexec.openmpi  /usr/bin/mpifort
```

use mpi_f08

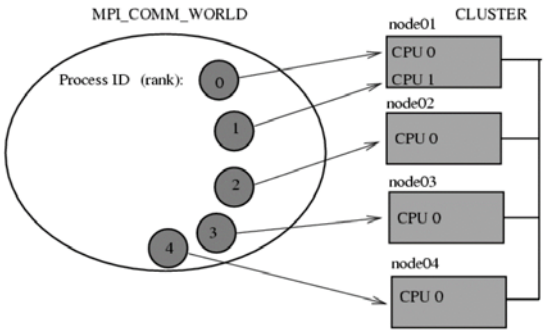
mpiifort, mpiicc, mpiicpc

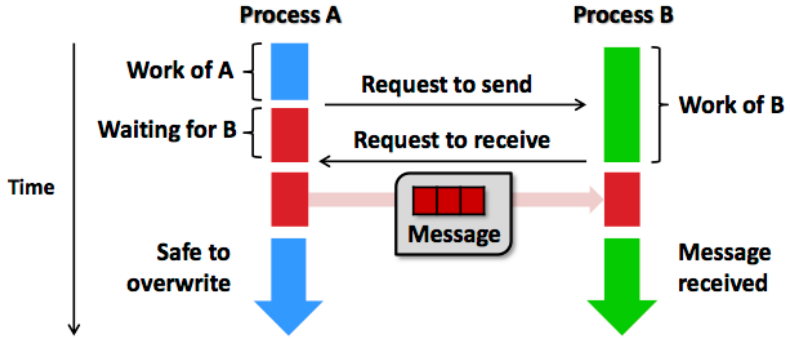
C Binding	
Format:	`rc = MPI_Xxxxx(parameter, ... )`
Example:	`rc = MPI_Bsend(&buf,count,type,dest,tag,comm)`
Error code:	Returned as "rc". MPI_SUCCESS if successful
Fortran Binding	
Format:	`CALL MPI_XXXXX(parameter,..., ierr)` `call mpi_xxxxx(parameter,..., ierr)`
Example:	`CALL MPI_BSEND(buf,count,type,dest,tag,comm,ierr)`
Error code:	Returned as "ierr" parameter. MPI_SUCCESS if successful



Communicators

- Communicators
  - A parameter for most MPI calls
  - A collection of processors working on some part of a parallel job
  - MPI_COMM_WORLD is defined in the MPI include file as all of the processors in your job
  - Can create subsets of MPI_COMM_WORLD
  - Processors within a communicator are assigned numbers 0 to n-1





Communicators

Communicator is a group of processors that can communicate with each other.
There can be many communicators.
MPI_COMM_WORLD is a **pre-defined communicator** encompassing all of the processes.

MPI_INIT: MPI 환경 초기화하기 : *유저 수준에서 바꿀 것이 사실상 없음.*
MPI_COMM_SIZE: 사용 중인 processor 숫자 반환 : 유저 수준에서 바꿀 것이 사실상 없음.
MPI_COMM_RANK: 현 CPU의 번호 (*rank*라도 함. processor 갯수가 nproc일 때, 가능한 rank 값은0,1,2,3....nproc-1이다.) : *유저 수준에서 바꿀 것이 사실상 없음.*
두 개의 processor 간 통신: rank값들을 사용하여서 현재 processor 번호를 확인하고 준비된 데이터를 원하는 processor로 전송한다. 마찬 가지로 현재의 processor번호를 확인하고 전송되어 올 데이터를 받는다. 물론, 우리는 어떤 processor로 부터 데이터가 오는지 그리고 어떤 processor가 이 데이터를 받아야 하는지 다 알고 있다.
MPI_SEND: 원하는 processor에게 데이터 전송시 사용 : 유저의 구체적인 목적이 적용됨 (원하는 데이터 형, 사이즈,...)
MPI_RECV: 원하는 processor로부터 데이터 전송받을 때 사용 : 유저의 구체적인 목적이 적용됨 (원하는 데이터 형, 사이즈,...)

MPI_FINALIZE: MPI 환경 종료하기: *유저 수준에서 바꿀 것이 사실상 없음.*

Include files

- The MPI include file
  - C: mpi.h
  - Fortran: mpif.h (a f90 module is a good place for this)
- Defines many constants used within MPI programs
- In C, defines the interfaces for the functions
- Compilers know where to find the include files

use mpi_f08

```
#include “mpi.h”
#include <stdio.h>
#include <math.h>
```

```
program main
implicit none
include ‘mpif.h’
```

/usr/local/mpich/include

basex11.h	mpeexten.h	mpi_fortdefs.h
f90base/	mpef.h	mpidefs.h
f90choice/	mpetools.h	mpif.h
mpe.h	mpi.h	mpio.h
mpe_graphics.h	mpi2c++/	mpiof.h
mpe_log.h	mpi_errno.h	protofix.h

```
include "mpif.h“     ! Constants used by MPI, e.g. constant integer value MPI_COMM_WORLD
integer istatus(MPI_STATUS_SIZE)      !    MPI_STATUS_SIZE는 위에서 선언한 include문으로 불러들인 내용에서 이미 정의된 것들이다.


integer nproc, me, ierr, idestination, isource, n_array_length, itag


  call MPI_Init(ierr)
  call MPI_Comm_size(MPI_COMM_WORLD, nproc, ierr)
  call MPI_Comm_rank(MPI_COMM_WORLD, me, ierr)
```

```
itag=19
idestination=1
call MPI_Send(real_array_user,n_array_length,MPI_REAL8,idestination,itag,MPI_COMM_WORLD,ierr)
```

real*8 형태의 데이터가 가야할 곳 지정해주어야 한다. 물론, 이 데이터의 크기도 보내는 곳에서 지정해줘야 한다. 특정 노드에서 정보를 보내기 때문에 위 함수는 특정 노드에서 불려져야 한다.

```
itag=19
isource=0
call MPI_Recv(real_array_user,n_array_length,MPI_REAL8,isource,itag,MPI_COMM_WORLD,istatus,ierr)
```

데이터를 받는 쪽에서는 그 형태와 크기를 알고 있어야 하며, 어디에서부터 출발했는지를 알아야 한다.

user-defined tags provided in MPI for user convenience in organizing application

Standard, Blocking Send

- MPI_Send: Sends data to another processor
- Use MPI_Receive to "get" the data
- C
  - MPI_Send(&buffer, count, datatype, *destination*, tag, communicator);
- Fortran
  - Call MPI_Send(buffer, count, datatype, *destination*, tag, communicator, ierr)
- Call blocks until message on the way

Basic C Datatypes in MPI

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_INT	unsigned int
MPI_UNSIGNED_LONG	unsigned long int
MPI_FLOAT	float
MPI_DOUBLE	double
MPI_LONG_DOUBLE	long double
MPI_BYTE	
MPI_PACKED	

MPI_COMPLEX

MPI_DOUBLE_COMPLEX

Language	Script Name	Underlying Compiler
C	mpicc	gcc
	mpigcc	gcc
	mpiicc	icc
	mpipgcc	pgcc
C++	mpiCC	g++
	mpig++	g++
	mpiicpc	icpc
	mpipgCC	pgCC
Fortran	mpif77	g77
	mpigfortran	gfortran
	mpiifort	ifort
	mpipgf77	pgf77
	mpipgf90	pgf90

ifort -fast
ifort -CB
ifort -check all
ifort -check all -warn interface
ifort -c -check all -warn interface
ifort -c -CB -check all -warn unused
ifort -CB -check all -warn interface -assume realloc_lhs

Compiler	Option	Example
Intel	-V	ifort -V
PGI	-V	pgf90 -V
GNU	-v --version	g++ --version

-fast ifort
-C
-CB
-check all

C/MPI version of “Hello, World”

```
#include <stdio.h>
#include <mpi.h>
int main(argc, argv)
int argc;
char *argv[ ];
{
        int myid, numprocs;


        MPI_Init (&argc, &argv) ;
        MPI_Comm_size(MPI_COMM_WORLD, &numprocs) ;
        MPI_Comm_rank(MPI_COMM_WORLD, &myid) ;
        printf(“Hello from %d\n”, myid) ;
        printf(“Numprocs is %d\n”, numprocs) ;

        MPI_finalize():
}
```

Fortran/MPI version of “Hello, World”

```
program  hello
include ‘mpif.h’
integer myid, ierr, numprocs

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

write (*,*) “Hello from “, myid
write (*,*) “Numprocs is”, numprocs
call MPI_FINALIZE(ierr)
stop
end
```

Minimal MPI program

```
#include <mpi.h>                    /* the mpi include file
*/
                                        /* Initialize MPI
*/
ierr=MPI_Init(&argc, &argv);
                           /* How many total PEs are there
*/
ierr=MPI_Finalize();
 ierr=MPI_Comm_size(MPI_COMM_WORLD, &nPEs);
                    /* What node am I (what is my rank?) */
 ierr=MPI_Comm_rank(MPI_COMM_WORLD, &iam);
 ...
```

In C, MPI routines are functions and return an error value.

Minimal MPI program

- Every MPI program needs these…
  - Fortran version

```
 include 'mpif.h'  ! MPI include file
c     Initialize MPI
      call MPI_Init(ierr)
c     Find total number of PEs
      call MPI_Comm_size(MPI_COMM_WORLD, nPEs, ierr)
c     Find the rank of this node
      call MPI_Comm_rank(MPI_COMM_WORLD, iam, ierr)
      ...
      call MPI_Finalize(ierr)
```

In Fortran, MPI routines are subroutines, and last parameter is an error-handling value.

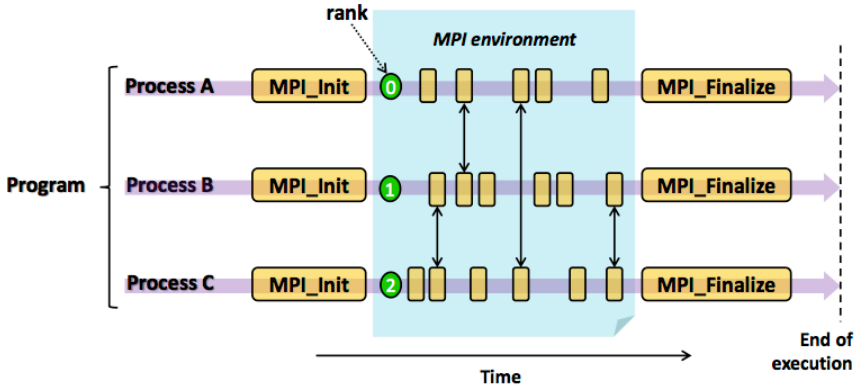
MPI “Hello, World”

- A parallel hello world program
  - Initialize MPI
  - Have each node print out its node number
  - Quit MPI

myid=0 : special

1 node calculation : 0
2 node calculation : 0, 1
3 node calculation,.....,. : 0, 1, 2

At least one node is involved.



Advantages of Message Passing

- Performance:
  - This is the most compelling reason why MP will remain a permanent part of parallel computing environment
  - As modern CPUs become faster, management of their caches and the memory hierarchy is the key to getting most out of them
  - MP allows a way for the programmer to explicitly associate specific data with processes and allows the compiler and cache management hardware to function fully
  - Memory bound applications can exhibit superlinear speedup when run on multiple PEs compare to single PE of MP machines

Background on MPI

- MPI - Message Passing Interface
  - Library standard defined by committee of vendors, implementers, and parallel programmer
  - Used to create parallel SPMD programs based on message passing
- Available on almost all parallel machines in C and Fortran
- About 125 routines including advanced routines
- 6 basic routines

Communicator
Point-to-point basics
Collective basics
Derived data types

Key concepts of MPI

- Used to create parallel SPMD programs based on message passing
- Normally the same program is running on several different nodes
- Nodes communicate using message passing

Data types

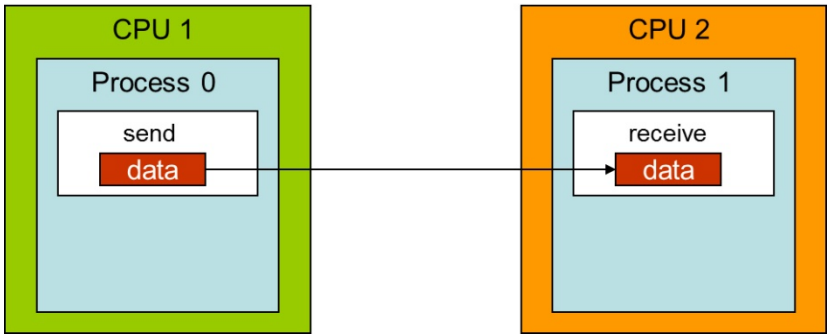
- Data types
  - When sending a message, it is given a data type
  - Predefined types correspond to "normal" types
    - MPI_REAL , MPI_FLOAT -- Fortran and C *real*
    - MPI_DOUBLE_PRECISION, (*mpi_real8*), MPI_DOUBLE -- Fortran and C *double*
    - MPI_INTEGER and MPI_INT -- Fortran and C *integer*
  - Can create user-defined types

Basic communications in MPI

- Data values are transferred from one processor to another
  - One process sends the data
  - Another receives the data
- Standard, Blocking
  - Call does not return until the message buffer is free to be reused
- Standard, Nonblocking
  - Call indicates a start of send or received, and another call is made to determine if finished

Basic MPI send and receive

- A parallel program to send & receive data
  - Initialize MPI
  - Have processor 0 send an integer to processor 1
  - Have processor 1 receive an integer from processor 0
  - Both processors print the data
  - Quit MPI



Simple send & receive Program

```
      program send_recv
      include "mpif.h"
! This is MPI send - recv program
      integer myid, ierr, numprocs
      integer itag, isource, idestination, kount
      integer buffer
      integer istatus(MPI_STATUS_SIZE)

      call MPI_INIT( ierr )
      call MPI_COMM_RANK( MPI_COMM_WORLD, myid, ierr )
      call MPI_COMM_SIZE( MPI_COMM_WORLD, numprocs, ierr )
      itag=1234
      isource=0
      idestination=1
      kount=1
     if(myid .eq. isource)then
        buffer=5678
        call MPI_Send(buffer, kount, MPI_INTEGER,idestination, itag, MPI_COMM_WORLD, ierr)
        write(*,*)"processor ",myid," sent ",buffer
      endif
      if(myid .eq. idestination)then
        call MPI_Recv(buffer, kount, MPI_INTEGER,isource, itag, MPI_COMM_WORLD, istatus, ierr)
        write(*,*)"processor ",myid," got ",buffer
      endif


      call MPI_FINALIZE(ierr)
      end
```

MPI_ANY_TAG
MPI_ANY_SOURCE

Simple send & receive Program

```
      PROGRAM simple_send_and_receive
      INCLUDE 'mpif.h'
      INTEGER myrank, ierr, istatus(MPI_STATUS_SIZE)
      REAL a(100)
C Initialize MPI:
      call MPI_INIT(ierr)
C Get my rank:
      call MPI_COMM_RANK(MPI_COMM_WORLD, myrank, ierr)
C Process 0 sends, process 1 receives:
      if( myrank .eq. 0 )then
      call MPI_SEND( a, 100, MPI_REAL, 1, 17, MPI_COMM_WORLD, ierr)
      else if ( myrank .eq. 1 )then
call MPI_RECV( a, 100, MPI_REAL, 0, 17, MPI_COMM_WORLD, istatus, ierr)
                              endif
C Terminate MPI:
      call MPI_FINALIZE(ierr)
      END
```

istatus(MPI_SOURCE), istatus(MPI_TAG), and istatus(MPI_ERROR) contain, respectively, the source, tag and error code of the received message.

```
call mpi_probe(0,mytag,MPI_COM_WORLD,istatus,ierr)
call mpi_get_count(istatus,MPI_REAL,icount,ierr)
write(*,*)"getting ", icount," values"
call mpi_recv(iray,icount,MPI_REAL,0,mytag,MPI_COMM_WORLD,istatus,ierr)
```

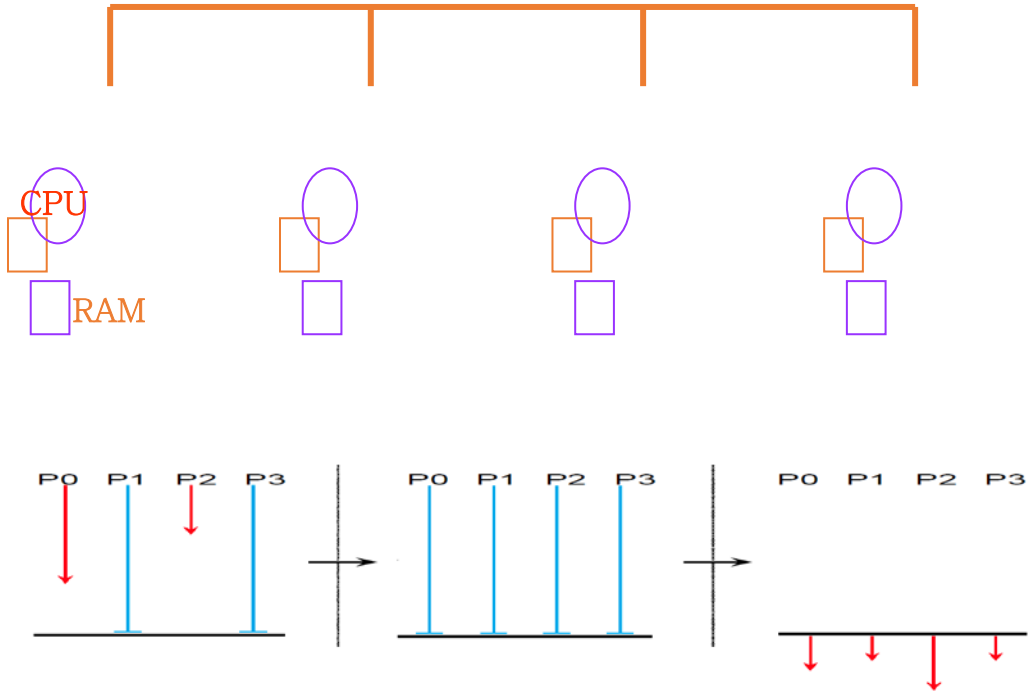
MPI_ANY_TAG
MPI_ANY_SOURCE

Simple send & receive Program

```
/* simple send and receive */
#include <stdio.h>
#include <mpi.h>

void main (int argc, char **argv) {
int myrank;
MPI_Status status;
double a[100];

MPI_Init(&argc, &argv);                                    /* Initialize MPI */
MPI_Comm_rank(MPI_COMM_WORLD, &myrank);                    /* Get rank */
if( myrank == 0 )                                          /* Send a message */
MPI_Send( a, 100, MPI_DOUBLE, 1, 17, MPI_COMM_WORLD );
else if( myrank == 1 )                                     /* Receive a message */
MPI_Recv( a, 100, MPI_DOUBLE, 0, 17, MPI_COMM_WORLD, &status );
MPI_Finalize();                                            /* Terminate MPI */
}
```

Messages can be screened at the receiving end by specifying a specific tag, or not screened by specifying MPI_ANY_TAG as the tag in a receive.

A set of computing nodes connected via networks



MPI_Send

call MPI_Send(buffer, icount, datatype, *destination*, itag, communicator, ierr)

- buffer: The data
- icount : Length of source array (in elements, 1 for scalars)
- datatype : Type of data, for example :

  MPI_DOUBLE_PRECISION, MPI_INT, etc
- *destination* : Processor number of destination processor in communicator
- itag : Message type (arbitrary integer)
- communicator : Your set of processors
- ierr : Error return (Fortran only)

Standard, Blocking Receive

- Call blocks until message is in buffer
- C
  - MPI_Recv(&buffer,count, datatype, *source*, tag, communicator, &istatus);
- Fortran
  - call MPI_ RECV(buffer, count, datatype, *source*, tag, communicator, istatus, ierr)
- Status - contains information about incoming message
  - C
    - MPI_Status istatus;
  - Fortran
    - Integer istatus(MPI_STATUS_SIZE)

MPI_Recv

call MPI_Recv(buffer, icount, datatype, *isource*, itag, communicator, *istatus*, ierr)

- buffer: The data
- icount : Max. number of elements that can be received
- datatype : Type of data, for example :

  MPI_DOUBLE_PRECISION, MPI_INT, etc
- *isource* : Processor number of source processor in communicator
- itag : Message type (arbitrary integer)
- communicator : Your set of processors
- *istatus*: Information about message
- ierr : Error return (Fortran only)

call MPI_RECV(ind,nid,MPI_INTEGER,MPI_ANY_SOURCE,itag,MPI_COMM_WORLD, *istatus*, ierr)
man=*istatus*(MPI_SOURCE)

istatus(MPI_SOURCE), *istatus*(MPI_TAG), and *istatus*(MPI_ERROR) contain, respectively, the source, tag, and error code of the received message.

```
program hello
  implicit none
  include "mpif.h"
  character(LEN=12) :: inmsg, message
  integer i, ierr, me, nproc, itag,kount
  integer istatus(MPI_STATUS_SIZE)

  call MPI_Init(ierr)
  call MPI_Comm_size(MPI_COMM_WORLD,nproc,ierr)
  call MPI_Comm_rank(MPI_COMM_WORLD,me,ierr)
  itag = 100 ; kount=12

  if (me == 0) then
    message = "Hello, world"
    do i = 1,nproc-1
      call MPI_Send(message,kount,MPI_CHARACTER,i,itag,MPI_COMM_WORLD,ierr)
    end do
    write(*,*) "process", me, ":", message
  else
    call MPI_Recv(inmsg,kount,MPI_CHARACTER,0,itag,MPI_COMM_WORLD,istatus,ierr)
    write(*,*) "process", me, ":", inmsg

  end if
  call MPI_Finalize(ierr)
end program hello
```

Simple send & receive Program

```
#include <stdio.h>
#include "mpi.h"
int main(argc,argv)
int argc;
char *argv[];
{
    int myid;
    int tag,source,destination,count;
    int buffer;
    MPI_Status status;
     MPI_Init(&argc,&argv);
    MPI_Comm_rank(MPI_COMM_WORLD,&myid);
tag=1234;
   source=0;
   destination=1;
   count=1;
   if(myid == source){
     buffer=5678;
    MPI_Send(&buffer,count,MPI_INT,destination,tag,MPI_COMM_WORLD);
     printf("processor %d  sent %d\n",myid,buffer);
   }
   if(myid == destination){
MPI_Recv(&buffer,count,MPI_INT,source,tag,MPI_COMM_WORLD,&status);
      printf("processor %d  got %d\n",myid,buffer);
   }
   MPI_Finalize();
 }
```

status.MPI_SOURCE, status.MPI_TAG, and status.MPI_ERROR contain the source, tag, and error code respectively of the received message

```
      subroutine Get_data(a, b, n, myid, nproc)
      IMPLICIT NONE
      real*8 a, b
      integer   n, myid,  nproc
C
      INCLUDE 'mpif.h'
      integer source, dest, tag, istatus(MPI_STATUS_SIZE), ierr
      data source /0/
C
      if (myid == 0) then
      print *, 'Enter a, b and n'
      read *, a, b, n
C
       do dest = 1 , nproc-1
        tag = 0
        call MPI_SEND(a, 1, MPI_REAL8 , dest, tag, MPI_COMM_WORLD, ierr )
        tag = 1
        call MPI_SEND(b, 1, MPI_REAL8 , dest, tag, MPI_COMM_WORLD, ierr )
        tag = 2
        call MPI_SEND(n, 1, MPI_INTEGER, dest, tag, MPI_COMM_WORLD, ierr )
       end do
                  else
        tag = 0
        call MPI_RECV(a, 1, MPI_REAL8 , source, tag, MPI_COMM_WORLD, istatus, ierr )
        tag = 1
        call MPI_RECV(b, 1, MPI_REAL8 , source, tag, MPI_COMM_WORLD, istatus, ierr )
        tag = 2
        call MPI_RECV(n, 1, MPI_INTEGER, source, tag, MPI_COMM_WORLD, istatus, ierr )
                  end if
       return
       end
```

istatus(MPI_SOURCE), istatus(MPI_TAG), istatus(MPI_ERROR) contain respectively the source, the tag, and the error code of the received message.

```
int recvd_tag, recvd_from, recvd_count;
MPI_Status status; MPI_Recv(..., MPI_ANY_SOURCE, MPI_ANY_TAG, ..., &status)
recvd_tag= status.MPI_TAG; recvd_from= status.MPI_SOURCE;
MPI_Get_count( &status, datatype, &recvd_count);
```

MPI User Guide in Fortran, P. S. Pacheco *et al.*

```
      PROGRAM simple_send_and_receive
      INCLUDE 'mpif.h'
      INTEGER myid, ierr, istatus(MPI_STATUS_SIZE)
      REAL  a(100)
C Initialize MPI:
      call MPI_INIT(ierr)
C Get my rank:
      call MPI_COMM_RANK(MPI_COMM_WORLD, myid, ierr)
C Process 0 sends, process 1 receives:
       if( myid .eq. 0 )then
       call MPI_SEND( a, 100, MPI_REAL, 1, 17, MPI_COMM_WORLD, ierr)
       else if ( myid .eq. 1 )then
       call MPI_RECV( a, 100, MPI_REAL, 0, 17, MPI_COMM_WORLD, istatus, ierr )
       endif
C Terminate MPI:
       call MPI_FINALIZE(ierr)
       END
```

Process 0 sends a message to process 1, and process 1 receives it.

istatus(MPI_SOURCE), istatus(MPI_TAG), and istatus(MPI_ERROR) contain, respectively, the source, tag, and error code of the received message.

```
include 'mpif.h'
integer myid, numprocs, ierr, ntemp
integer istatus (MPI_STATUS_SIZE)
real*8 sum
call MPI_INIT(ierr)
call MPI_COMM_RANK(MPI_COMM_WORLD, myid, ierr)
call MPI_COMM_SIZE(MPI_COMM_WORLD, numprocs, ierr)
sum=myid
if(myid == 0)then
do i=1,numprocs-1
call MPI_RECV(ntemp,1,MPI_INTEGER, i,i, MPI_COMM_WORLD, istatus, ierr)
sum=sum+ntemp
enddo
average=sum/numprocs ; write(6,*) 'the average is ', average
else
call MPI_SEND(myid,1,MPI_INTEGER,0,myid,MPI_COMM_WORLD, ierr)
endif
call MPI_FINALIZE(ierr)
end
```

istatus - contains information about incoming message

```
subroutine get_input(aa,bb,nn)
include “mpif.h”
real*8 aa,bb
integer nn,myid
integer ierr

call MPI_COMM_RANK(MPI_COMM_WORLD,myid,ierr)

if(myid == 0)then
print *, ‘Enter aa, bb, nn’
read *, aa,bb,nn
endif

call MPI_BCAST(aa,1, MPI_REAL8,  0, MPI_COMM_WORLD, ierr)
call MPI_BCAST(bb,1, MPI_REAL8, 0, MPI_COMM_WORLD,  ierr)
call MPI_BCAST(nn,1, MPI_INTEGER, 0, MPI_COMM_WORLD,  ierr)

end subroutine get_input
```

```
implicit none
integer n, np, i, j, ierr, master,num
real*8 h, result, a, b, integral, pi, my_a, my_range, MPI_WTime,start_time,end_time, my_result
include "mpif.h"
integer Iam, source, dest, tag, istatus(MPI_STATUS_SIZE)
data master/0/
call MPI_Init(ierr)  ;   call MPI_Comm_rank(MPI_COMM_WORLD, Iam, ierr)
call MPI_Comm_size(MPI_COMM_WORLD, np, ierr)
  pi = acos(-1.0d0) ;   a = 0.0d0  ; b = pi*1.d0/2.d0  ;  dest = 0   ; tag = 123
  if(Iam  ==  master) then
print *,'The requested number of processors =',p ;  print *,'Enter total number of increments across all processors'
   read(*,*) n
   start_time = MPI_Wtime()
  endif
   call MPI_Bcast(n, 1, MPI_INTEGER, 0, MPI_COMM_WORLD, ierr)
  h = (b-a)/n   ;   num = n/np ;  my_range = (b-a)/np ;  my_a = a+ Iam*my_range
  my_result = integral(my_a, num, h)
  write(*,"('Process ',i2,' has the partial result of',f10.6)")   Iam,my_result
  call MPI_Reduce(my_result,result, 1, MPI_REAL8, MPI_SUM, dest, MPI_COMM_WORLD, ierr)
  if(Iam  == master) then
    print *,'The result =',result
   end_time = MPI_Wtime() ;  print *, 'elapsed time is ',end_time-start_time,' seconds'
  endif
  call MPI_Finalize(ierr)  ; stop ;  end
```

Status

- In C
  - status is a structure of type MPI_Status which contains three fields MPI_SOURCE, MPI_TAG, and MPI_ERROR
  - status.MPI_SOURCE, status.MPI_TAG, and status.MPI_ERROR contain the source, tag, and error code respectively of the received message
- In Fortran
  - status is an array of INTEGERS of length MPI_STATUS_SIZE, and the 3 constants MPI_SOURCE, MPI_TAG, MPI_ERROR are the indices of the entries that store the source, tag, & error
  - status(MPI_SOURCE), status(MPI_TAG), and status(MPI_ERROR) contain respectively the source, the tag, and the error code of the received message.

status.MPI_SOURCE, status.MPI_TAG, and status.MPI_ERROR contain the source, tag, and error code respectively of the received message

```
! Program hello.ex1.f
! Parallel version using MPI calls
! Modified from basic version so that workers send back
! a message to the master, who prints out a message for each worker
program hello
implicit none
integer, parameter:: DOUBLE=kind(1.0d0), SINGLE=kind(1.0)
include "mpif.h"
character(LEN=12) :: inmsg,message
integer :: i,ierr,me,nproc,itag,iwrank
integer, dimension(MPI_STATUS_SIZE) :: istatus
!
call MPI_Init(ierr)
call MPI_Comm_size(MPI_COMM_WORLD,nproc,ierr)
call MPI_Comm_rank(MPI_COMM_WORLD,me,ierr)
itag = 100
!
if (me == 0) then
message = "Hello, world"
do i = 1,nproc-1
call MPI_Send(message,12,MPI_CHARACTER,i,itag,MPI_COMM_WORLD,ierr)
end do
write(*,*) "process", me, ":", message
do i = 1,nproc-1
call MPI_Recv(iwrank,1,MPI_INTEGER,MPI_ANY_SOURCE,itag,MPI_COMM_WORLD, istatus, ierr)
write(*,*) "process", iwrank, ":Hello, back"
end do
else
call MPI_Recv(inmsg,12,MPI_CHARACTER,0,itag,MPI_COMM_WORLD,istatus,ierr)
call MPI_Send(me,1,MPI_INTEGER,0,itag,MPI_COMM_WORLD,ierr)
end if
call MPI_Finalize(ierr)
end program hello
```


http://www.dartmouth.edu/~rc/classes/

```
Program Example1
implicit none
integer n, p, i, j,num
real h, result, a, b, integral, pi
real my_a,my_range
pi = acos(-1.0)                    ! = 3.141592...
a = 0.0                            ! lower limit of integration
b = pi*1./2.                       ! upper limit of integration
p = 4 !! number of processes (partitions)
n = 100000 !! total number of increments
h = (b-a)/n                        ! length of increment
num= n/p                           ! number of calculations done by each process
result = 0.0                       ! stores answer to the integral
do i=0,p-1                         ! sum of integrals over all processes
my_range = (b-a)/p
my_a = a + i*my_range
result = result + integral(my_a,num,h)
enddo
print *,'The result =',result
stop
end

real function integral(a,n,h)
implicit none
integer n, i, j
real h, h2, aij, a
real fct, x
fct(x) = cos(x)                    ! kernel of the integral
integral = 0.0                     ! initialize integral
h2 = h/2.
do j=0,n-1                         ! sum over all "j" integrals
aij = a+j*h                        ! lower limit of "j" integral
integral = integral + fct(aij+h2)*h
enddo
return
end
```

Buffer and Deadlock

safe

```
if (rank == 0 )then
call mpi_send(sbuff,kount,MPI_REAL8,  1,  itag, MPI_COMM_WORLD, ierr)
call mpi_recv(rbuff,kount,MPI_REAL8,  1,  itag, MPI_COMM_WORLD, istatus,ierr)
else if (rank ==1 )then
call mpi_recv(rbuff,kount,MPI_REAL8,  0,  itag, MPI_COMM_WORLD, istatus,ierr)
call mpi_send(sbuff, kount, MPI_REAL8,  0,  itag, MPI_COMM_WORLD, ierr)
endif
```

Deadlock; 수렁, 교착

```
if (rank == 0 )then
call mpi_recv(rbuff,kount,MPI_REAL8, 1,  itag, MPI_COMM_WORLD, istatus,ierr)
call mpi_send(sbuff,kount,MPI_REAL8, 1,  itag, MPI_COMM_WORLD, ierr)
else if (rank ==1 )then
call mpi_recv(rbuff,kount,MPI_REAL8,  0,  itag, MPI_COMM_WORLD, istatus, ierr)
call mpi_send(sbuff, kount, MPI_REAL8,  0,  itag, MPI_COMM_WORLD, ierr)
endif
```

Buffering dependent

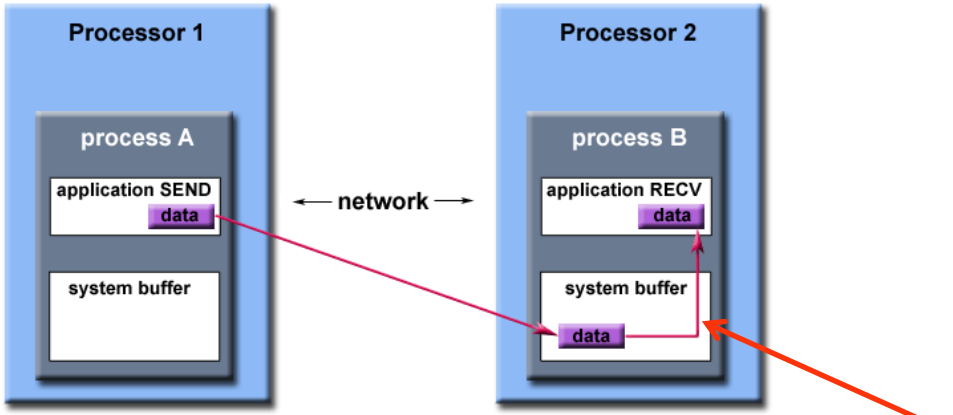
```
if (rank == 0 )then
call mpi_send(sbuff,kount,MPI_REAL8,  1,  itag, MPI_COMM_WORLD, ierr)
call mpi_recv(rbuff,kount,MPI_REAL8,  1,  itag, MPI_COMM_WORLD, istatus,ierr)
else if (rank ==1 )then
call mpi_send(sbuff, kount, MPI_REAL8,  0,  itag, MPI_COMM_WORLD, ierr)
call mpi_recv(rbuff,kount,MPI_REAL8,  0,  itag, MPI_COMM_WORLD, istatus, ierr)
endif
```

kount is large→ deadlock occurs

Non-blocking communications

The programmer should be able to explain why the program does not (or does) deadlock.

Note that increasing array dimensions and message sizes have no effect on the safety of the protocol.



Path of a message buffered at the receiving process

https://computing.llnl.gov/tutorials/mpi/

System buffer size

Deadlock (C)

```
                                                    /* simple deadlock */
#include <stdio.h>
#include <mpi.h>
void main (int argc, char **argv) {
int myrank;
MPI_Status status;
double a[100], b[100];
MPI_Init(&argc, &argv);                                    /* Initialize MPI */
MPI_Comm_rank(MPI_COMM_WORLD, &myrank);          /* Get rank */
if( myrank == 0 ) {
                                        /* Receive, then send a message */
MPI_Recv( b, 100, MPI_DOUBLE,  1, 19,  MPI_COMM_WORLD, &status );
MPI_Send( a, 100, MPI_DOUBLE,  1, 17,  MPI_COMM_WORLD );
}
else if( myrank == 1 ) {
                                         /* Receive, then send a message */
MPI_Recv( b, 100, MPI_DOUBLE,  0, 17,  MPI_COMM_WORLD, &status );
MPI_Send( a, 100, MPI_DOUBLE,  0, 19,  MPI_COMM_WORLD );
}
MPI_Finalize();                              /* Terminate MPI */
}
```

Deadlock (FORTRAN)

```
        PROGRAM simple_deadlock
        INCLUDE 'mpif.h'
        INTEGER myrank, ierr, istatus(MPI_STATUS_SIZE)
        REAL a(100), b(100)
C Initialize MPI:
        call MPI_INIT(ierr)
C Get my rank:
        call MPI_COMM_RANK(MPI_COMM_WORLD, myrank, ierr)
C Process 0 receives and sends; same for process 1
if( myrank .eq. 0 )then
   call MPI_RECV( b, 100, MPI_REAL, 1, 19, MPI_COMM_WORLD, istatus, ierr )
   call MPI_SEND( a, 100, MPI_REAL, 1, 17, MPI_COMM_WORLD, ierr)
else if ( myrank .eq. 1 )then
   call MPI_RECV( b, 100, MPI_REAL, 0, 17, MPI_COMM_WORLD, istatus, ierr )
   call MPI_SEND( a, 100, MPI_REAL, 0, 19, MPI_COMM_WORLD, ierr)
endif
C Terminate MPI:
        call MPI_FINALIZE(ierr)
        END
```

```
      PROGRAM safe_exchange
      INCLUDE 'mpif.h'
      INTEGER myrank, ierr, istatus(MPI_STATUS_SIZE)
      REAL  a(100), b(100)
C Initialize MPI:
      call MPI_INIT(ierr)
C Get my rank:
      call MPI_COMM_RANK(MPI_COMM_WORLD, myrank, ierr)
C Process 0 receives and sends; process 1 sends and receives
      if( myrank  .eq.  0 )then
      call MPI_RECV( b, 100, MPI_REAL,  1,  19,  MPI_COMM_WORLD, istatus, ierr )
      call MPI_SEND( a, 100, MPI_REAL,  1,  17,  MPI_COMM_WORLD, ierr)
      else if ( myrank  .eq.  1 )then
      call MPI_SEND( a, 100, MPI_REAL,  0,  19,  MPI_COMM_WORLD, ierr )
      call MPI_RECV( b, 100, MPI_REAL,  0,  17,  MPI_COMM_WORLD, istatus, ierr)
      endif
C Terminate MPI:
      call MPI_FINALIZE(ierr)
      END
```

```
                                                    /* safe exchange */
#include <stdio.h>
#include <mpi.h>
void main (int argc, char **argv) {
int myrank;
MPI_Status status;
double a[100], b[100];
MPI_Init(&argc, &argv);                    /* Initialize MPI */
MPI_Comm_rank(MPI_COMM_WORLD, &myrank);    /* Get rank */
if( myrank == 0 ) {
                                           /* Receive a message, then send one */
MPI_Recv( b, 100, MPI_DOUBLE, 1, 19, MPI_COMM_WORLD, &status );
MPI_Send( a, 100, MPI_DOUBLE, 1, 17, MPI_COMM_WORLD );
}
else if( myrank == 1 ) {
                                            /* Send a message, then receive one */
MPI_Send( a, 100, MPI_DOUBLE, 0, 19, MPI_COMM_WORLD );
MPI_Recv( b, 100, MPI_DOUBLE, 0, 17, MPI_COMM_WORLD, &status );
}
MPI_Finalize();                                          /* Terminate MPI */
}
```

status.MPI_SOURCE, status.MPI_TAG, and status.MPI_ERROR contain the source, tag, and error code respectively of the received message

status.MPI_SOURCE 자료를 보낸 노드의 아이디를 알아 낼 수 있다.

```
      PROGRAM depends_on_buffering
      INCLUDE 'mpif.h'
      INTEGER myrank, ierr, istatus(MPI_STATUS_SIZE)
      REAL*8  a(100), b(100)
C Initialize MPI:
      call MPI_INIT(ierr)
C Get my rank:
      call MPI_COMM_RANK(MPI_COMM_WORLD, myrank, ierr)
C Process 0 sends and receives; same for process 1
      if( myrank .eq. 0 )then
      call MPI_SEND( a, 100, MPI_REAL8,   1,   17, MPI_COMM_WORLD, ierr)
      call MPI_RECV( b, 100, MPI_REAL8,   1,   19, MPI_COMM_WORLD, istatus, ierr )
      else if ( myrank .eq. 1 )then
      call MPI_SEND( a, 100, MPI_REAL8,   0,   19, MPI_COMM_WORLD, ierr )
      call MPI_RECV( b, 100, MPI_REAL8,   0,   17, MPI_COMM_WORLD, istatus, ierr)
      endif
C Terminate MPI:
      call MPI_FINALIZE(ierr)
      END
```

REAL*8 → DOUBLE PRECISION
MPI_REAL8 → MPI_DOUBLE_PRECISION

Again, process 0 attempts to exchange messages with process 1. This time, both processes send first, then receive. Success depends on the availability of buffering in MPI.

If the message sizes are increased, sooner or later the program will deadlock.

```
      PROGRAM probable_deadlock
      INCLUDE 'mpif.h'
      INTEGER myrank, ierr, istatus(MPI_STATUS_SIZE)
      INTEGER n
      PARAMETER (n=100000000)
      REAL  a(n), b(n)
C Initialize MPI:
      call MPI_INIT(ierr)
C Get my rank:
      call MPI_COMM_RANK(MPI_COMM_WORLD, myrank, ierr)
C Process 0 sends, then receives; same for process 1
      if( myrank.eq.0 )then
   call MPI_SEND( a, n, MPI_REAL, 1, 17,  MPI_COMM_WORLD, ierr)
   call MPI_RECV( b, n, MPI_REAL, 1, 19,  MPI_COMM_WORLD, istatus, ierr )
      else if ( myrank.eq.1 )then
   call MPI_SEND( a, n, MPI_REAL, 0, 19,  MPI_COMM_WORLD, ierr )
   call MPI_RECV( b, n, MPI_REAL, 0, 17,  MPI_COMM_WORLD, istatus, ierr)
      endif
C Terminate MPI:
      call MPI_FINALIZE(ierr)
      END
```

This program will deadlock under the default configuration of nearly all available MPI implementations.

Asynchronous communication

```
      PROGRAM simple_deadlock_avoided
        INCLUDE 'mpif.h'
        INTEGER myrank, ierr, istatus(MPI_STATUS_SIZE)
        INTEGER irequest
        REAL  a(100), b(100)
C Initialize MPI:
        call MPI_INIT(ierr)
C Get my rank:
        call MPI_COMM_RANK(MPI_COMM_WORLD, myrank, ierr)
C Process 0 posts a receive, then sends; same for process 1
        if( myrank .eq. 0 )then
        call MPI_IRECV( b, 100, MPI_REAL, 1, 19,  MPI_COMM_WORLD, irequest, ierr )
        call MPI_SEND( a, 100, MPI_REAL, 1, 17,  MPI_COMM_WORLD, ierr)
        call MPI_WAIT( irequest, istatus, ierr )
        else if ( myrank .eq. 1 )then
        call MPI_IRECV( b, 100, MPI_REAL, 0, 17,  MPI_COMM_WORLD, irequest, ierr )
        call MPI_SEND( a, 100, MPI_REAL, 0, 19,  MPI_COMM_WORLD, ierr)
        call MPI_WAIT( irequest, istatus, ierr )
                        endif
C Terminate MPI:
        call MPI_FINALIZE(ierr)
        END
```

MPI_Wait used to complete communication

This is a race condition, which can be very difficult to debug.

```
int i=123; MPI_Request myRequest; MPI_Isend(&i, 1, MPI_INT, 1,
MY_LITTLE_TAG,MPI_COMM_WORLD, &myRequest);
i=234;
```

```
int i=123; MPI_Request myRequest;
MPI_Isend(&i, 1, MPI_INT, 1, MY_LITTLE_TAG, MPI_COMM_WORLD, &myRequest);
// do some calculations here // Before we re-use variable i, we need to wait until the asynchronous function call is complete
MPI_Status myStatus; MPI_Wait(&myRequest, &myStatus);
i=234;
```

MPI_Wait blocks until the message specified by “request” completes.

```
                                                                    /*   deadlock avoided    */
#include
#include
void main (int argc, char **argv) {
int myrank;
MPI_Request request;
MPI_Status status;
double a[100], b[100];
MPI_Init(&argc, &argv);                                              /*   Initialize MPI   */
MPI_Comm_rank(MPI_COMM_WORLD, &myrank);                              /*   Get rank   */
if( myrank == 0 ) {
                                                                     /*   Post a receive, send a message, then wait   */
MPI_Irecv( b, 100, MPI_DOUBLE, 1, 19,  MPI_COMM_WORLD, &request );
MPI_Send( a, 100, MPI_DOUBLE, 1, 17,  MPI_COMM_WORLD );
MPI_Wait( &request, &status );
}
else if( myrank == 1 ) {
                                                                     /*   Post a receive, send a message, then wait   */
MPI_Irecv( b, 100, MPI_DOUBLE, 0, 17,  MPI_COMM_WORLD, &request );
MPI_Send( a, 100, MPI_DOUBLE, 0, 19,  MPI_COMM_WORLD );
MPI_Wait( &request, &status );
}
MPI_Finalize();                                                      /*   Terminate MPI   */
}
```

It is always safe to order the calls of MPI_(I)SEND and MPI_(I)RECV so that a send subroutine call at one process and a corresponding receive subroutine call at the other process appear in matching order.

```
IF (myrank==0) THEN
  CALL MPI_SEND(sendbuf, ...)
  CALL MPI_RECV(recvbuf, ...)
ELSEIF (myrank==1) THEN
  CALL MPI_RECV(recvbuf, ...)
  CALL MPI_SEND(sendbuf, ...)
ENDIF
```

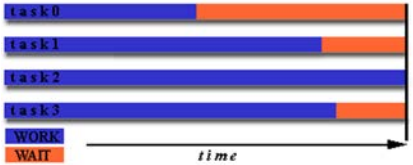
In this case, you can use either blocking or non-blocking subroutines.

Considering the previous options, performance, and the avoidance of deadlocks, it is recommended to use the following code.

```
IF (myrank==0) THEN
  CALL MPI_ISEND(sendbuf, ..., ireq1, ...)
  CALL MPI_IRECV(recvbuf, ..., ireq2, ...)
ELSEIF (myrank==1) THEN
  CALL MPI_ISEND(sendbuf, ..., ireq1, ...)
  CALL MPI_IRECV(recvbuf, ..., ireq2, ...)
ENDIF
CALL MPI_WAIT(ireq1, ...)
CALL MPI_WAIT(ireq2, ...)
```

Master-slave mode

never ending at the same time



master-slave mode, king-soldier mode

```
if(nproc > 1)then
if(myid == 0)then   ! -----[   process id = 0
call feedin1(nconf,iter)
        else   ! -----    process id /= 0
call predict1(nconf)
        endif  ! -----]   process id /= 0
        else
do ip=1,ncrystals
call grbfnn_predict(ip,tmp)
enddo
        endif
```

Master-slave mode

never ending at the same time

```
      subroutine feedin1(nconf0,iter0)
      USE crystal_set, ONLY : maxnatoms,penergy_set,pstress_set,pfxyz_set
      implicit none
      include 'mpif.h'
      integer nconf0,iter0
      integer, parameter :: maxnid=2
      integer ind(maxnid),nid,id,mm,man,kid,itag,koumpi,istatus(MPI_STATUS_SIZE)
      mm=0
      DO id=1,min(nproc-1,nconf0)
      mm=mm+1
!  Generation  [                    ]
      itag=1  ;       call MPI_SEND(mm,1,MPI_INTEGER, mm, itag,MPI_COMM_WORLD,ierr)
      enddo
      do id=1,nconf0
      itag=3 ;       nid=2
      call MPI_RECV(ind,nid,MPI_INTEGER,MPI_ANY_SOURCE,itag,MPI_COMM_WORLD,istatus,ierr)
      kid=ind(1)  ;    man=istatus(MPI_SOURCE)  ;  itag=5
      koumpi=1
      call MPI_RECV(penergy_set(kid),koumpi,MPI_DOUBLE_PRECISION,man,itag,MPI_COMM_WORLD,istatus,ierr)
      koumpi=6
      call MPI_RECV(pstress_set(1,kid),koumpi,MPI_DOUBLE_PRECISION,man,itag,MPI_COMM_WORLD,istatus,ierr)
      koumpi=3*maxnatoms
      call MPI_RECV(pfxyz_set(1,1,kid),koumpi,MPI_DOUBLE_PRECISION,man,itag,MPI_COMM_WORLD,istatus,ierr)
!  Analysis  [                      ]
      if (mm < nconf0)then
      mm=mm+1
!  Generation  [                    ]
      itag=1 ;       call MPI_SEND(mm,1,MPI_INTEGER, man, itag,MPI_COMM_WORLD,ierr)
                    else
      itag=1 ;       call MPI_SEND(0,1,MPI_INTEGER, man, itag,MPI_COMM_WORLD,ierr)
                    endif
      ENDDO   ;       end subroutine feedin1
```

Master-slave mode

never ending at the same time

```
      subroutine predict1(nconf0)
      USE crystal_set, ONLY : maxnatoms,penergy_set,pstress_set,pfxyz_set
      implicit none
      include 'mpif.h'
      integer nconf0
      integer, parameter :: maxnid=2
      integer ind(maxnid),nid,id,itag,koumpi,istatus(MPI_STATUS_SIZE)
      real*8 eslave

      if (myid > nconf0) return
      DO
      itag=1
      call MPI_RECV(id,1,MPI_INTEGER, 0, itag,MPI_COMM_WORLD,istatus,ierr)
      if (id == 0) return
!  local   [
      call grbfnn_predict(id,eslave)
!  local   ]
      itag=3  ;     nid=2  ;       ind(2)=id   ; ind(1)=id
      call MPI_SEND(ind,nid,MPI_INTEGER, 0, itag,MPI_COMM_WORLD,ierr)
      itag=5  ;       koumpi=1
      call MPI_SEND(penergy_set(id),koumpi,MPI_DOUBLE_PRECISION, 0, itag,MPI_COMM_WORLD,ierr)
      koumpi=6
      call MPI_SEND(pstress_set(1,id),koumpi,MPI_DOUBLE_PRECISION, 0, itag,MPI_COMM_WORLD,ierr)
      koumpi=3*maxnatoms
      call MPI_SEND(pfxyz_set(1,1,id),koumpi,MPI_DOUBLE_PRECISION, 0, itag,MPI_COMM_WORLD,ierr)
      ENDO
      end subroutine predict1
```

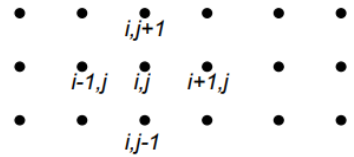
Finite-difference method

2-D Laplace equation

$$\frac{\partial^2}{\partial x^2}u(x,y)+\frac{\partial^2}{\partial y^2}u(x,y)=0$$

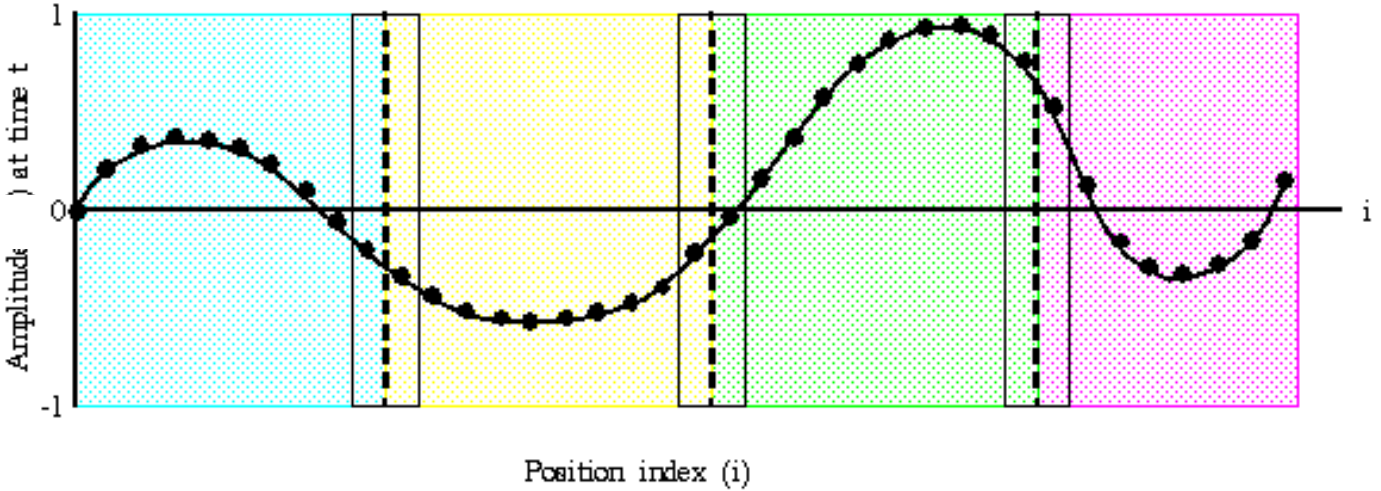
Central discretization leads to

$$\frac{u_{i+1,j}-2u_{i,j}+u_{i-1,j}}{h^2}+\frac{u_{i,j+1}-2u_{i,j}+u_{i,j-1}}{h^2}=0$$



https://www.sharcnet.ca/help/images/4/4f/Parallel_Numerical_Solution_of_PDEs_with_Message_Passing.pdf
http://www2.cs.uh.edu/~gabriel/courses/cosc6374_f07/ParCo_05_laplace.pdf
https://people.sc.fsu.edu/~jburkardt/f_src/heat_mpi/heat_mpi.f90
https://dournac.org/info/parallel_heat2d

Communications?



Message-passing tools for Structured Grid communications (MSG)

MSG-2.0

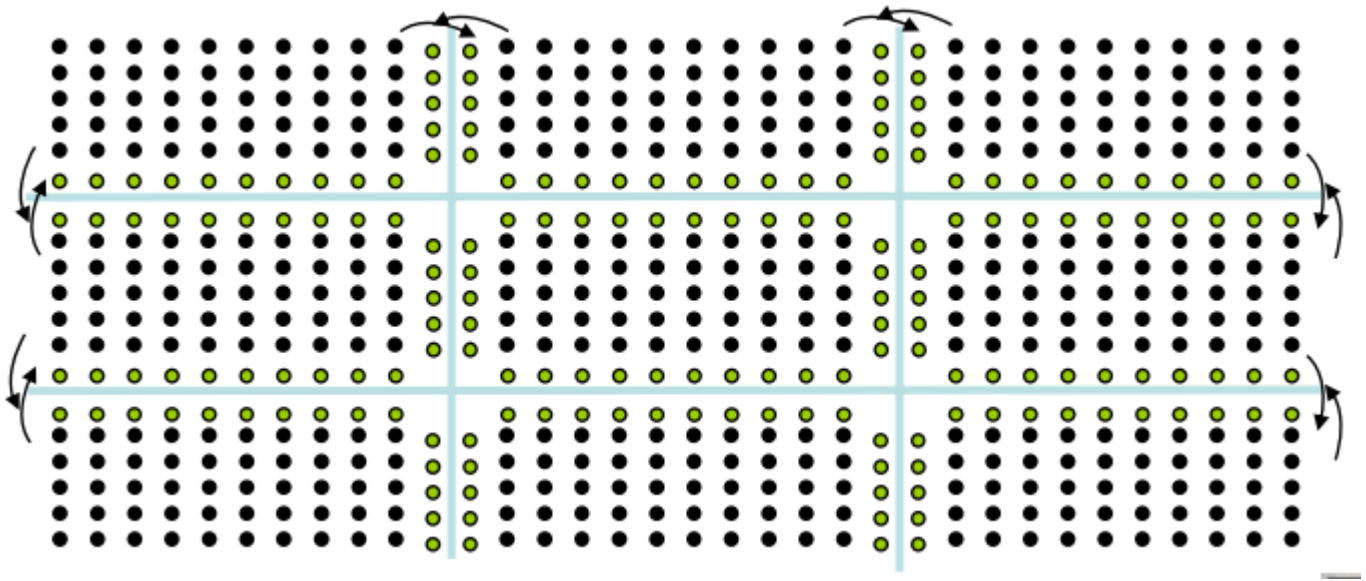
FFT ←→ FD

https://www.mcs.anl.gov/research/projects/mpi/libraries.html

Finite-difference method

Data exchange at process boundaries required

Halo

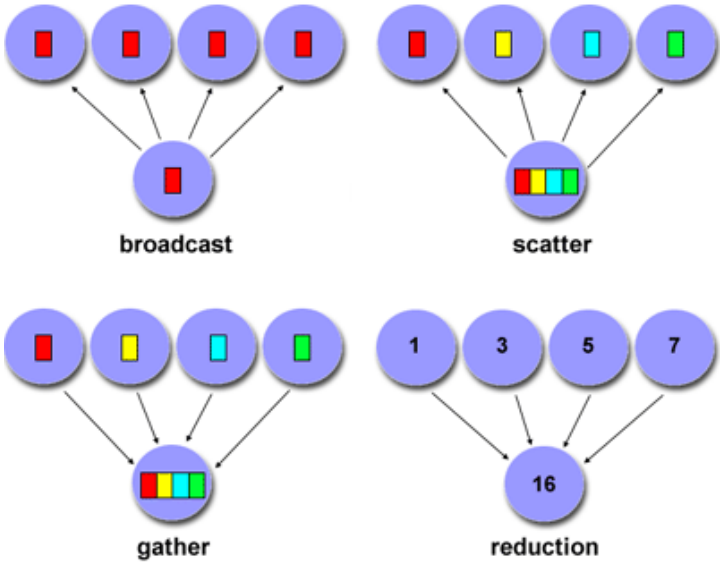


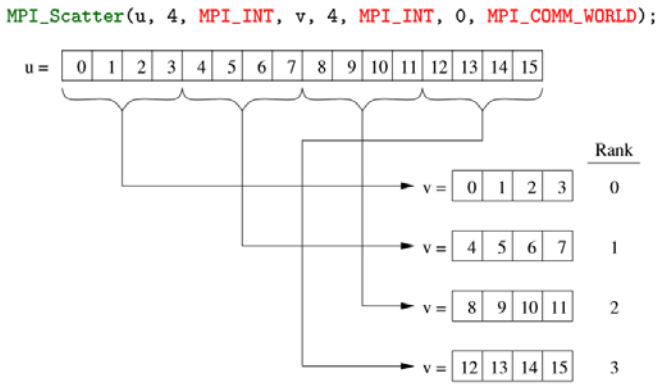
https://www.uio.no/studier/emner/matnat/ifi/INF3380/v11/undervisningsmateriale/inf3380-week11.pdf

Point-to-point communications

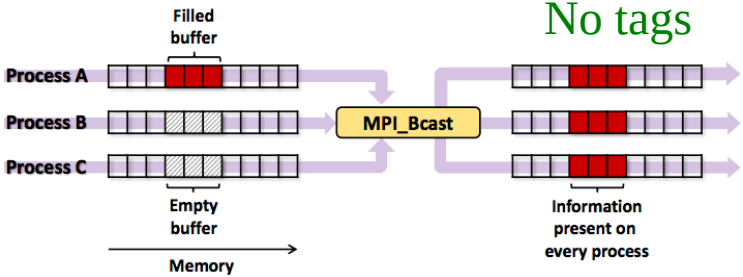
Collective communications

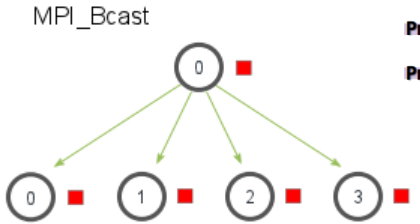
No tags



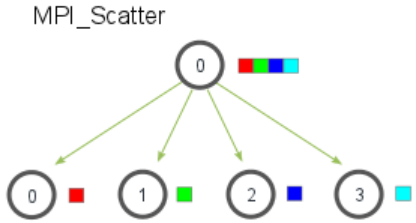


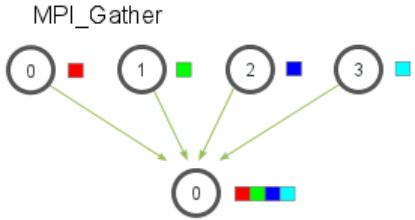
No tags

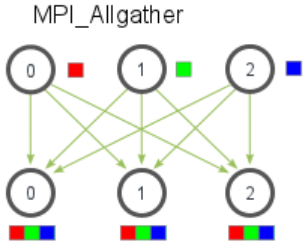




MPI_Bcast() – Broadcast (one to all)
MPI_Reduce() – Reduction (all to one)
MPI_Allreduce() – Reduction (all to all)
MPI_Scatter() – Distribute data (one to all)
MPI_Gather() – Collect data (all to one)
MPI_Alltoall() – Distribute data (all to all)
MPI_Allgather() – Collect data (all to all)





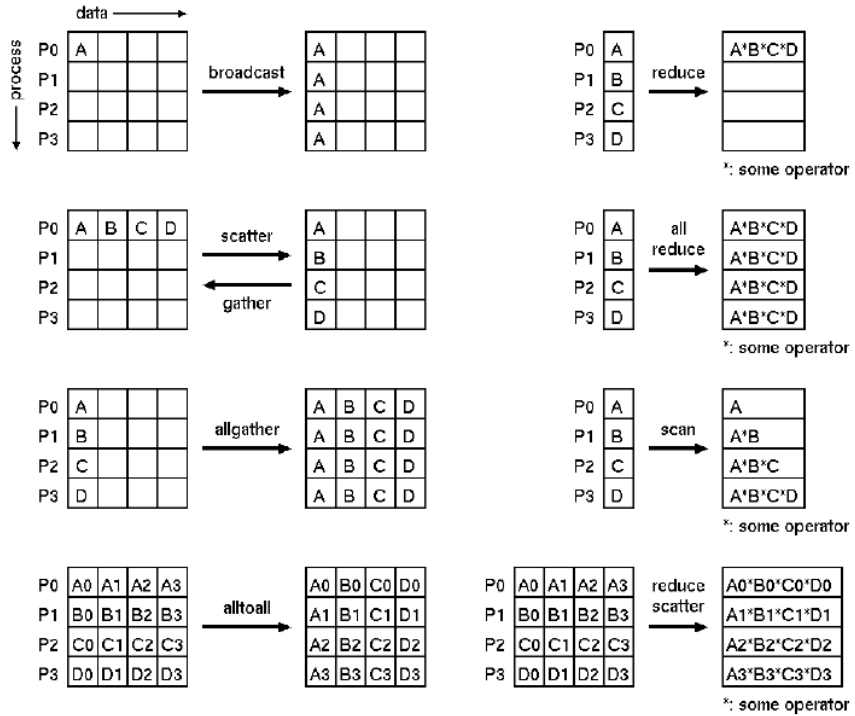


Collective communications

오류의 가능성이 현저히 적다.
이미 최적화된 통신 방법이다.

No tags

scatterv
gatherv

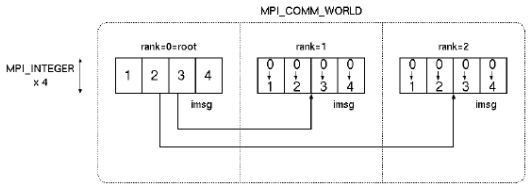


Collective communications

No tags

bcast.f

```
 1       PROGRAM bcast
 2       INCLUDE 'mpif.h'
 3       INTEGER imsg(4)
 4       CALL MPI_INIT(ierr)
 5       CALL MPI_COMM_SIZE(MPI_COMM_WORLD, nprocs, ierr)
 6       CALL MPI_COMM_RANK(MPI_COMM_WORLD, myrank, ierr)
 7       IF (myrank==0) THEN
 8         DO i=1,4
 9           imsg(i) = i
10         ENDDO
11       ELSE
12         DO i=1,4
13           imsg(i) = 0
14         ENDDO
15       ENDIF
16       PRINT *,'Before:',imsg
17       CALL MP_FLUSH(1)
18       CALL MPI_BCAST(imsg, 4, MPI_INTEGER,
19      &               0, MPI_COMM_WORLD, ierr)
20       PRINT *,'After :',imsg
21       CALL MPI_FINALIZE(ierr)
22       END
```



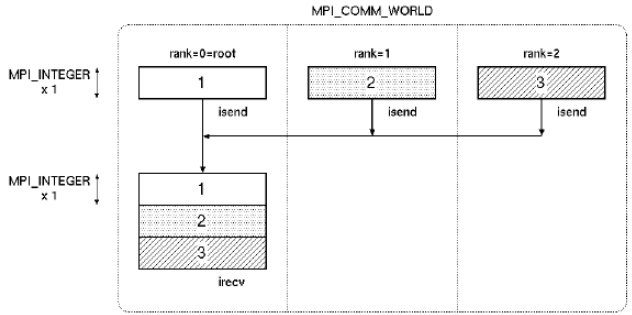
No tags

Collective communications

No tags

gather.f

```
1       PROGRAM gather
2       INCLUDE 'mpif.h'
3       INTEGER irecv(3)
4       CALL MPI_INIT(ierr)
5       CALL MPI_COMM_SIZE(MPI_COMM_WORLD, nprocs, ierr)
6       CALL MPI_COMM_RANK(MPI_COMM_WORLD, myrank, ierr)
7       isend = myrank + 1
8       CALL MPI_GATHER(isend, 1, MPI_INTEGER,
9      &                irecv, 1, MPI_INTEGER,
10     &                0, MPI_COMM_WORLD, ierr)
11      IF (myrank==0) THEN
12        PRINT *,'irecv =',irecv
13      ENDIF
14      CALL MPI_FINALIZE(ierr)
15      END
```

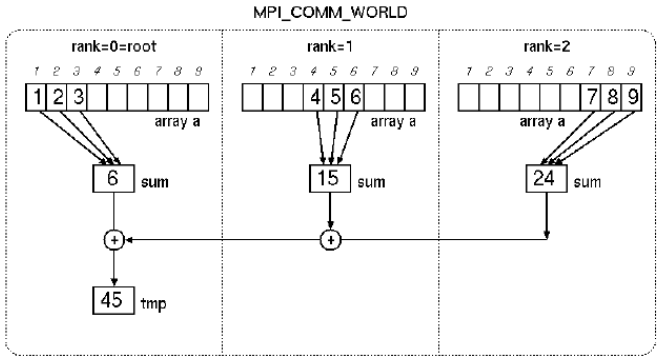


Collective communications

No tags

```
1       PROGRAM reduce
2       INCLUDE 'mpif.h'
3       REAL a(9)
4       CALL MPI_INIT(ierr)
5       CALL MPI_COMM_SIZE(MPI_COMM_WORLD, nprocs, ierr)
6       CALL MPI_COMM_RANK(MPI_COMM_WORLD, myrank, ierr)
7       ista = myrank * 3 + 1
8       iend = ista + 2
9       DO i=ista,iend
10        a(i) = i
11      ENDDO
12      sum = 0.0
13      DO i=ista,iend
14        sum = sum + a(i)
15      ENDDO
16      CALL MPI_REDUCE(sum, tmp, 1, MPI_REAL, MPI_SUM, 0,
17     &                MPI_COMM_WORLD, ierr)
18      sum = tmp
19      IF (myrank==0) THEN
20        PRINT *,'sum =',sum
21      ENDIF
22      CALL MPI_FINALIZE(ierr)
23      END
```

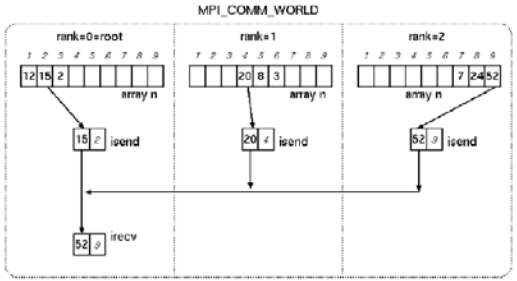
No tags



Collective communications

No tags

```
      PROGRAM maxloc_p
      INCLUDE 'mpif.h'
      INTEGER n(9)
    INTEGER isend(2), irecv(2)
    DATA n /12, 15, 2, 20, 8, 3, 7, 24, 52/
    CALL MPI_INIT(ierr)
    CALL MPI_COMM_SIZE(MPI_COMM_WORLD, nprocs, ierr)
    CALL MPI_COMM_RANK(MPI_COMM_WORLD, myrank, ierr)
    ista = myrank * 3 + 1
    iend = ista + 2
    imax = -999
    DO i = ista, iend
      IF (n(i) > imax) THEN
        imax = n(i)
        iloc = i
      ENDIF
    ENDDO
    isend(1) = imax
    isend(2) = iloc
    CALL MPI_REDUCE(isend, irecv, 1, MPI_2INTEGER,
   &                MPI_MAXLOC, 0, MPI_COMM_WORLD, ierr)
    IF (myrank == 0) THEN
      PRINT *, 'Max =', irecv(1), 'Location =', irecv(2)
    ENDIF
    CALL MPI_FINALIZE(ierr)
    END
```



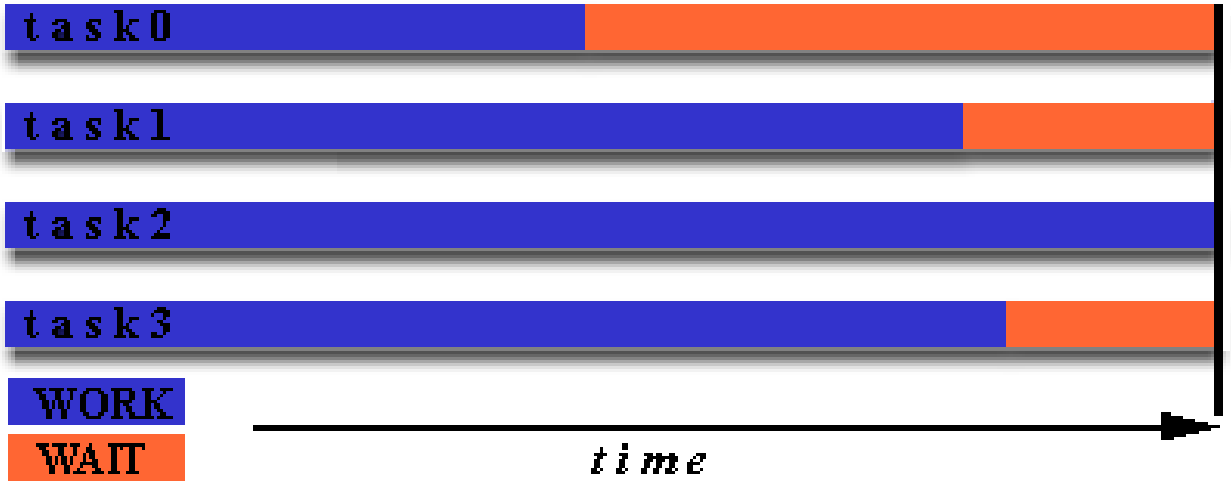
No tags

do not use a call such as `MPI_Reduce(&x, &x, 1, MPI_DOUBLE, 0, comm);`

Reduce operators

MPI Reduction Operation		C Data Types	Fortran Data Type
MPI_MAX	maximum	integer, float	integer, real, complex
MPI_MIN	minimum	integer, float	integer, real, complex
MPI_SUM	sum	integer, float	integer, real, complex
MPI_PROD	product	integer, float	integer, real, complex
MPI_LAND	logical AND	integer	logical
MPI_BAND	bit-wise AND	integer, MPI_BYTE	integer, MPI_BYTE
MPI_LOR	logical OR	integer	logical
MPI_BOR	bit-wise OR	integer, MPI_BYTE	integer, MPI_BYTE
MPI_LXOR	logical XOR	integer	logical
MPI_BXOR	bit-wise XOR	integer, MPI_BYTE	integer, MPI_BYTE
MPI_MAXLOC	max value and location	float, double and long double	real, complex,double precision
MPI_MINLOC	min value and location	float, double and long double	real, complex, double precision

Load balancing



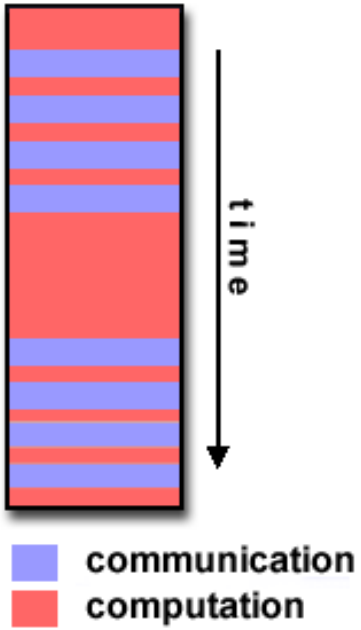
top, 각 노드에서 직접 확인, “평균하는 시간간격”을 조절함 (top→d).

Equally partition the work

Use dynamic work assignment

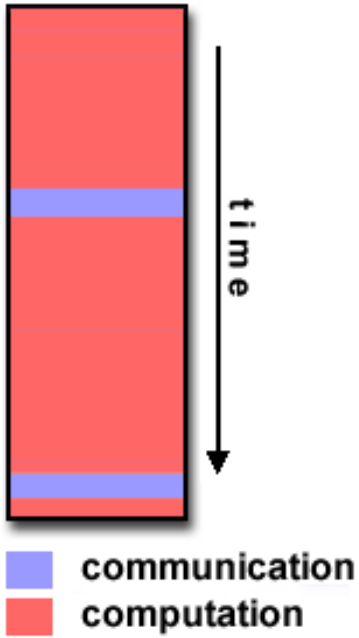
Granularity

Fine-grain Parallelism



새로운 알고리듬?

Coarse-grain Parallelism



Relatively **large amounts of computational work are done** between *communication/synchronization events*

Problem decomposition

- Domain decompostion
- Functional decomposition

이더넷으로 감당할 수 있는 경우

★ 같은 방식의 계산인데 데이터가 서로 다른 경우

★ 서로 다른 노드의 데이터가 필요하고 노드 자체의 데이터와 연합하여 계산이 진행되는 경우

Infiniband가 필요한 경우

Parallel programming issues

★load balancing
★minimizing communication
★overlapping communication and computation

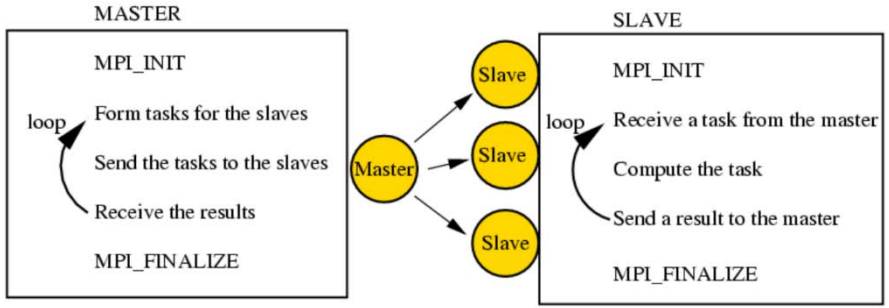
계산과 통신을 연합하여야 한다. 통신은 될 수 있으면 자주 하지 않아야 한다. (minimizing communication)
왜냐하면 통신은 매우 느리기 때문이다.
노드들이 거의 균등한 정도로 계산을 수행하면 아주 좋다. (load balancing)
이러한 계산을 충분히 오랬 동안 수행한 다음에 통신을 하는 방식이 좋다.

계속해서 계산할 인풋자료를 계산 노드들에 전달하는 경우가 있다. 물론, 계산된 결과는 계산 즉시 받아들인다. 또한, 다음에 계산할 추가 작업을 위한 인풋자료를 계속해서 나누는 경우가 있을 수 있다.
이 때, MPI_ANY_SOURCE를 이용해서 자료를 받아 들인다. 어느 계산노드에서 온 계산결과 자료인지 알 수 있다. 노드에서 계산하는 것들이 모두 동일한 컴퓨터 시간을 요구하지 않는 경우에 보다 더 적합한 방법이다. 예를 들면, 계속적인 방법으로 최소화 하는 작업을 하는 경우, 계산마다 최종 수렴 계산결과를 얻어내는 시간이 동일하지 않을 수 있다.

It is very difficult in practice to interleave communication with computation.

수행해야 할 일들을 나누고 결과물을 받아오는 방식.

```
      allocate(exqq(natom_ac,3,0:np_ac), exforce(natom_ac,3,0:np_ac), exvofqj(0:np_ac))
      if(myid == 0)then    ! -----{ PROCESS ID = 0
      exqq=qq
                  endif   ! -----} PROCESS ID = 0
      iroot=0 ; kount=3*natom_ac*(np_ac+1)
      call MPI_BCAST(exqq, kount,MPI_REAL8, iroot,MPI_COMM_WORLD,ierr)
      n1=0 ; n2=np_ac
      call equal_load(n1,n2,nproc,myid,jstart,jfinish)
!
      exforce=0.0d0 ; exvofqj=0.0d0
      do j=jstart,jfinish
      call xtinker(exqq(1,1,j),exforce(1,1,j),exvofqj(j),natom_ac,isequence,isymbol,iattyp,ii12)
      enddo
      iroot=0 ; kount=3*natom_ac*(np_ac+1)
      call MPI_REDUCE(exforce,force_ac,kount,MPI_DOUBLE_PRECISION,MPI_SUM,iroot,MPI_COMM_WORLD,ierr)
      iroot=0 ; kount=(np_ac+1)
      call MPI_REDUCE(exvofqj,vofqj,kount,MPI_DOUBLE_PRECISION,MPI_SUM,iroot,MPI_COMM_WORLD,ierr)
!
      deallocate(exforce, exvofqj, exqq)
```



참조

```
real*8 aa(n,n), bb(n,n)

 do j=1,n
 do i=1,n
 aa(i,j)=0.d0
 bb(i,j)=0.0d0
 enddo
 enddo

 do j=1,n
 do i=1+myid,n, nproc
 aa(i,j)=...
 enddo
 enddo

 call MPI_REDUCE(aa,bb,n*n, MPI_DOUBLE_PRECISION, MPI_SUM, 0, MPI_COMM_WORLD, ierr)
```

```
PROGRAM main
PARAMETER (n = 1000)
DIMENSION a(n)
DO i = 1, n
  a(i) = i
ENDDO
sum = 0.0
DO i = 1, n
  sum = sum + a(i)
ENDDO
PRINT *,'sum =',sum
END
```

나누어서 더할 수 있다!

```
SUBROUTINE para_range(n1, n2, nprocs, irank, ista, iend)
iwork1 = (n2 - n1 + 1) / nprocs
iwork2 = MOD(n2 - n1 + 1, nprocs)
ista = irank * iwork1 + n1 + MIN(irank, iwork2)
iend = ista + iwork1 - 1
IF (iwork2 > irank) iend = iend + 1
END
```

Iteration	1	2	3	4	5	6	7	8	9	10	11	12	13	14
Rank	0	0	0	0	1	1	1	1	2	2	2	3	3	3

```
SUBROUTINE para_range(n1, n2, nprocs, irank, ista, iend)
iwork = (n2 - n1) / nprocs + 1
ista = MIN(irank * iwork + n1, n2 + 1)
iend = MIN(ista + iwork - 1, n2)
END
```

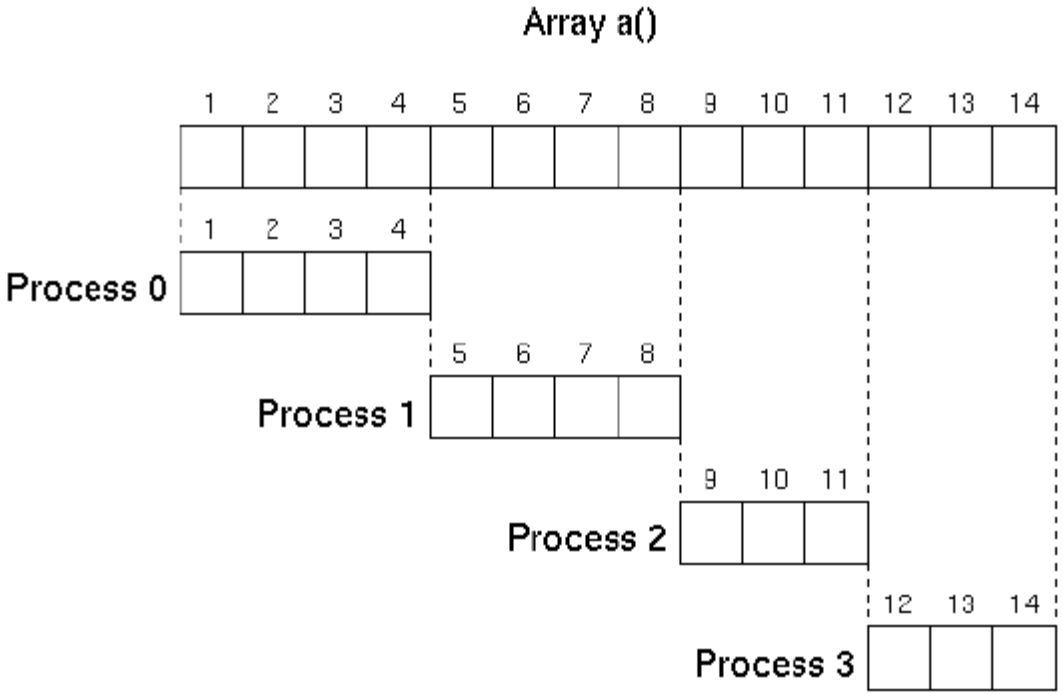
Iteration	1	2	3	4	5	6	7	8	9	10	11	12	13	14
Rank	0	0	0	0	1	1	1	1	2	2	2	2	3	3

```
      subroutine equal_load(n1,n2,nproc,myid,istart,ifinish)
      implicit none
      integer nproc,myid,istart,ifinish,n1,n2
      integer iw1,iw2
      iw1=(n2-n1+1)/nproc ; iw2=mod(n2-n1+1,nproc)
      istart=myid*iw1+n1+min(myid,iw2)
      ifinish=istart+iw1-1 ; if(iw2 > myid) ifinish=ifinish+1
!     print*, n1,n2,myid,nproc,istart,ifinish
      if(n2 < istart) ifinish=istart-1
      return
      end
```

```
PROGRAM main
INCLUDE 'mpif.h'
PARAMETER (n = 1000)
DIMENSION a(n)
CALL MPI_INIT(ierr)
CALL MPI_COMM_SIZE(MPI_COMM_WORLD, nprocs, ierr)
CALL MPI_COMM_RANK(MPI_COMM_WORLD, myrank, ierr)
CALL para_range(1, n, nprocs, myrank, ista, iend)
DO i = ista, iend
  a(i) = i
ENDDO
sum = 0.0
DO i = ista, iend
  sum = sum + a(i)
ENDDO
CALL MPI_REDUCE(sum, ssum, 1, MPI_REAL,
&                 MPI_SUM, 0, MPI_COMM_WORLD, ierr)
sum = ssum
IF (myrank == 0) PRINT *,'sum =',sum
CALL MPI_FINALIZE(ierr)
END
```

계산하는 노드별로 다른 시작, 종료 지점을 알려 준다.

노드별로 시작, 종료점이 다르다는 것을 알려준다. 잘 나누어서 알려준다.



가장 많이 사용되는 do loop 병렬화의 방식을 아래에 표시했다.
do loop 병렬화 (3가지 방식)

1/3. block distribution

```
do i=n1,n2
......
end do
----------> 아래와 같이 serial에서 parallel로 바뀝니다.
do i=ista,iend
.....
end do
여기에서 ista, iend는 노드별(irank)로 다른값이 할당된다.

subroutine para_range(n1,n2,nprocs,irank,ista,iend)
implicit none
integer n1,n2,nprocs,irank,ista,iend
integer iwork1,iwork2
iwork1=(n2-n1+1)/nprocs
iwork2=mod(n2-n1+1,nprocs)
ista=irank*iwork1+n1+min(irank,iwork2)
iend=ista+iwork1-1
if(iwork2> irank) iend=iend+1
end
```

Iteration	1	2	3	4	5	6	7	8	9	10	11	12	13	14
Rank	0	0	0	0	1	1	1	1	2	2	2	3	3	3

Figure 49. Block Distribution

2/3. cyclic distribution

round-robin fashion

do i=n1,n2
.....
end do
----------> 아래와 같이 serial에서 parallel로 바뀝니다.
do i=n1+irank, n2, nprocs
.....
end do

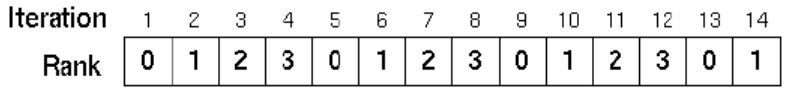


Figure 51. Cyclic Distribution

3/3. block-cyclic distribution

```
do i=n1,n2
.....
end do
----------> 아래와 같이 serial에서 parallel로 바뀝니다.
do ii=n1+irank*iblock, n2, nprocs*iblock
do i=ii,min(ii+iblock-1,n2)
.....
end do
end do
```

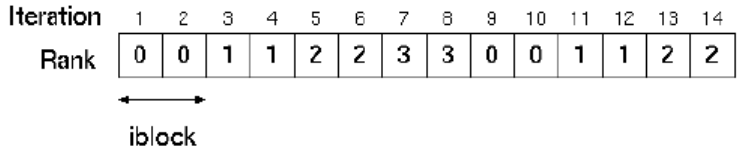


Figure 52. Block-Cyclic Distribution

```
      PROGRAM main
      implicit none
      real sum1,ssum
      integer i,ista,iend
      integer ierr,n1,n2,nprocs,myrank
      INCLUDE 'mpif.h'
      PARAMETER (n1 = 1, n2 = 1000)
      REAL, ALLOCATABLE :: a(:)

      CALL MPI_INIT(ierr)
      CALL MPI_COMM_SIZE(MPI_COMM_WORLD, nprocs, ierr)
      CALL MPI_COMM_RANK(MPI_COMM_WORLD, myrank, ierr)
      CALL para_range(n1, n2, nprocs, myrank, ista, iend)
      ALLOCATE (a(ista:iend))
      DO i = ista, iend
      a(i) = i
      ENDDO
!     sum1 = 0.0
!     DO i = ista, iend
!     sum1 = sum1 + a(i)
!    ENDDO
     sum1=sum(a)
      DEALLOCATE (a)
     CALL MPI_REDUCE(sum1, ssum, 1, MPI_REAL,MPI_SUM, 0, MPI_COMM_WORLD, ierr)
     sum1 = ssum
     PRINT *,'sum1 =',sum1, myrank
     CALL MPI_FINALIZE(ierr)
     END
```

real*8

shrink

MPI_REAL8
MPI_DOUBLE_PRECISION

```
PROGRAM main
INCLUDE 'mpif.h'
PARAMETER (n1 = 1, n2 = 1000)
REAL, ALLOCATABLE :: a(:)
CALL MPI_INIT(ierr)
CALL MPI_COMM_SIZE(MPI_COMM_WORLD, nprocs, ierr)
CALL MPI_COMM_RANK(MPI_COMM_WORLD, myrank, ierr)
CALL para_range(n1, n2, nprocs, myrank, ista, iend)
ALLOCATE (a(ista:iend))
DO i = ista, iend
  a(i) = i
ENDDO
sum = 0.0
DO i = ista, iend
  sum = sum + a(i)

ENDDO
DEALLOCATE (a)
CALL MPI_REDUCE(sum, ssum, 1, MPI_REAL,
&               MPI_SUM, 0, MPI_COMM_WORLD, ierr)
sum = ssum
PRINT *,'sum =',sum
CALL MPI_FINALIZE(ierr)
END
```

real*8 sum, ssum

real*8, allocatable ::

MPI_DOUBLE_PRECISION

MPI_REAL8

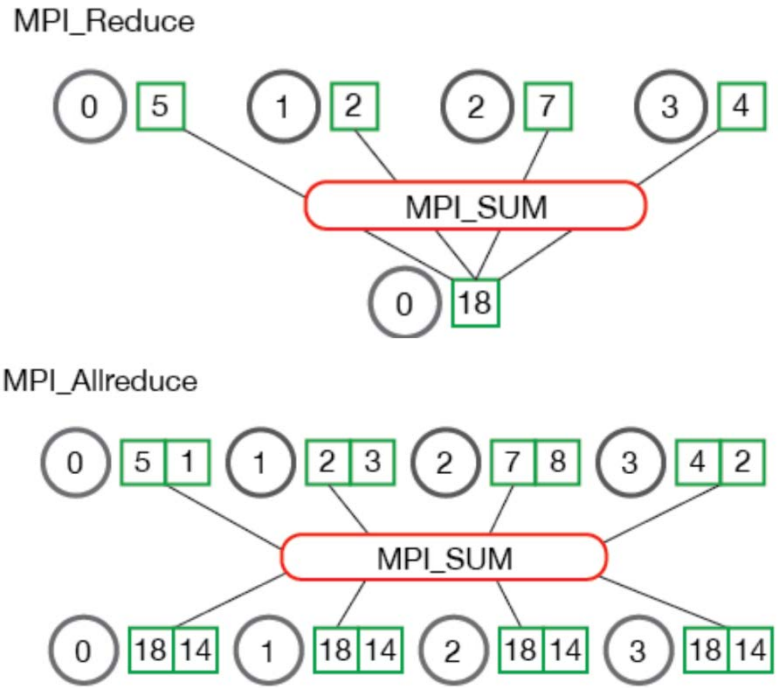
One-sided communication이 아닌 collective communication 이 매우 유용하다.
예를 들어, 노드별로 흩어져 있는 결과들을 종합하거나
특정 데이터를 노드별로 나누고자 할 때 collective communication 방법이 유용하다.

Processors

Memory

CPU

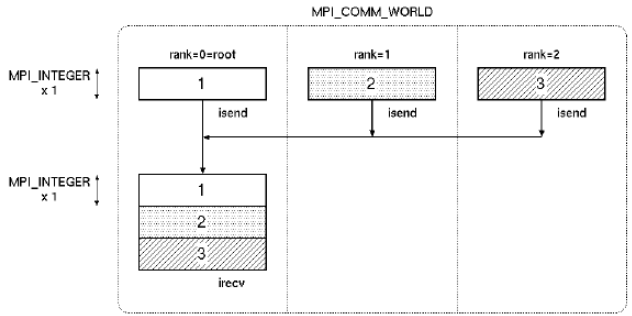
top



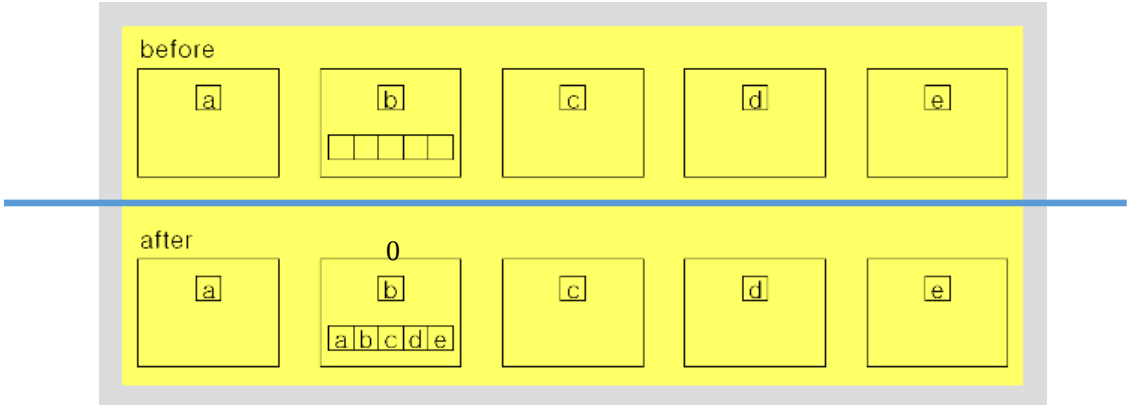
Collective communications

gather.f

```
 1       PROGRAM gather
 2       INCLUDE 'mpif.h'
 3       INTEGER irecv(3)
 4       CALL MPI_INIT(ierr)
 5       CALL MPI_COMM_SIZE(MPI_COMM_WORLD, nprocs, ierr)
 6       CALL MPI_COMM_RANK(MPI_COMM_WORLD, myrank, ierr)
 7       isend = myrank + 1
 8       CALL MPI_GATHER(isend, 1, MPI_INTEGER,
 9      &                irecv, 1, MPI_INTEGER,
10      &                0, MPI_COMM_WORLD, ierr)
11       IF (myrank==0) THEN
12         PRINT *,'irecv =',irecv
13       ENDIF
14       CALL MPI_FINALIZE(ierr)
15       END
```



gatherv



각 노드에 할당된 특정 크기(M)의 자료들을 특정 노드(0 노드)에 모두 모으고자 할 때가 있을 수 있다. 이 때, 특정 노드(0 노드)에서는 노드 수(nproc)에 비례하는 배열 할당(M × nproc)이 필요하게 된다.

irregular message sizes allowed

gaps allowed between messages in source data

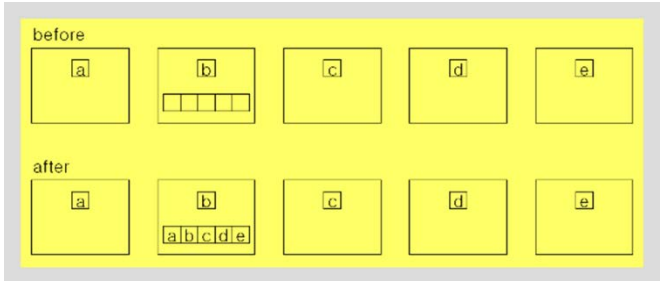
data can be distributed to processes in any order

http://incredible.egloos.com/4086075

http://www.cac.cornell.edu/Ranger/MPIcc/scatterv.aspx

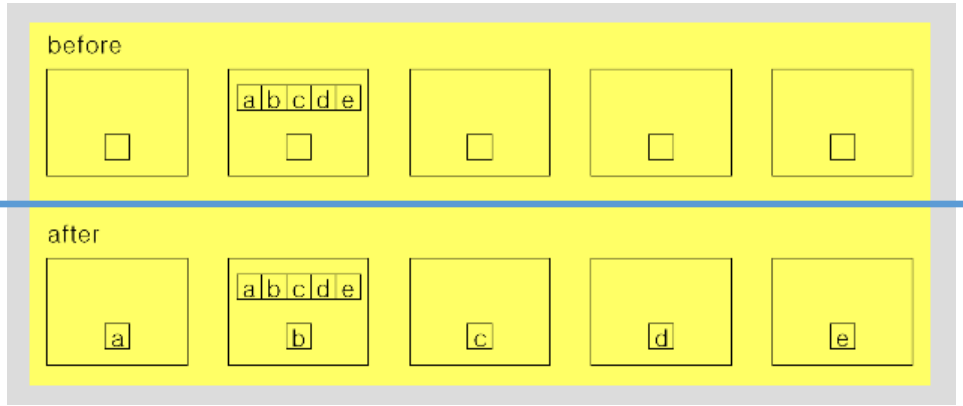
gatherv

```
!234567890
implicit none
include 'mpif.h'
integer, allocatable :: isend(:), irecv(:)
integer, allocatable :: ircnt(:), idisp(:)
integer ierr,nproc,myid
integer i,iscnt, ndsize
call MPI_INIT(ierr)
call MPI_COMM_SIZE(MPI_COMM_WORLD, nproc, ierr)
call MPI_COMM_RANK(MPI_COMM_WORLD, myid, ierr)
ndsize=10
allocate(isend(ndsize),irecv(nproc*ndsize))
allocate(ircnt(0:nproc-1),idisp(0:nproc-1))
ircnt=ndsize
idisp(0)=0
do i=1,nproc-1
idisp(i)=idisp(i-1)+ircnt(i)
enddo
do i=1,ndsize                                   ! node specific data with a data-size ndsize
isend(i)=myid+1
enddo
iscnt=ndsize
call MPI_GATHERV(isend, iscnt, MPI_INTEGER, irecv, ircnt, idisp, MPI_INTEGER,0, MPI_COMM_WORLD, ierr)
if(myid == 0)then
print*, 'irecv= ',irecv
                endif
deallocate(ircnt,idisp)  ;   deallocate(isend,irecv)
call MPI_FINALIZE(ierr)  ;  stop  ;  end
```



scatterv

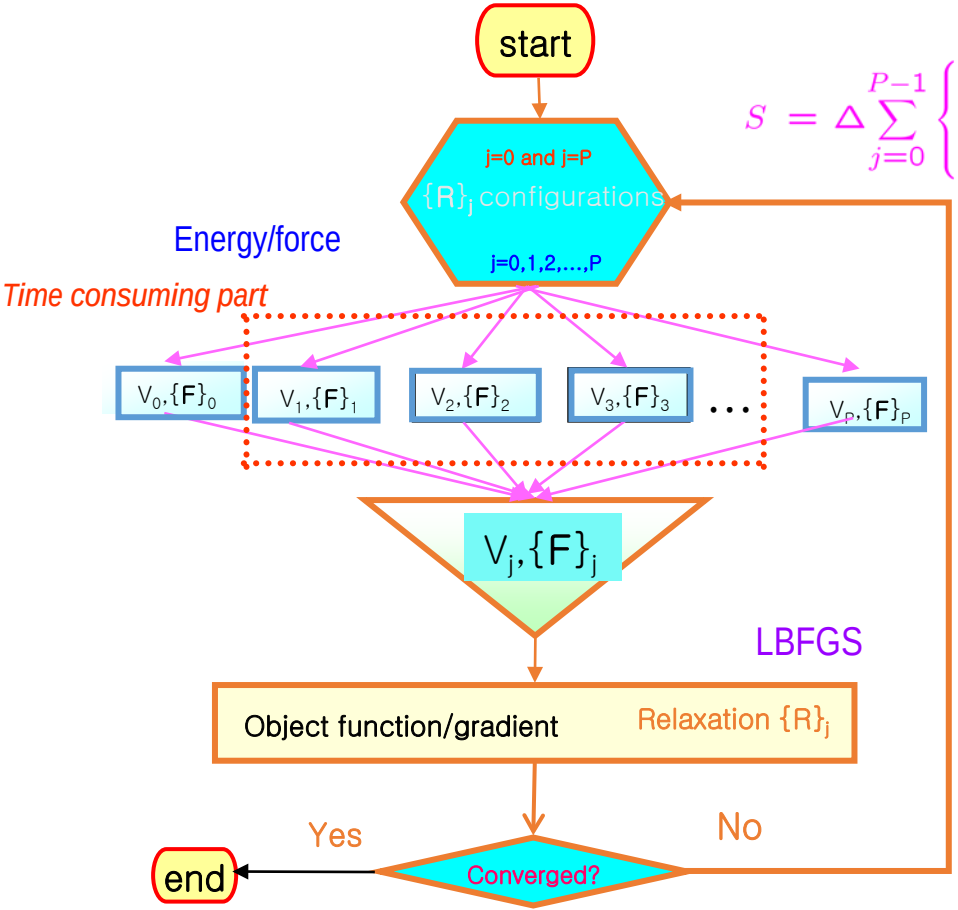
```
real*8 a(25), rbuf(MAX); integer displs(NX), rcounts(NX), nsize
do i= 1, nsize
displs(i) = (i-1)*stride
rcounts(i) = 25
enddo
call mpi_gatherv(a, 25, MPI_REAL, rbuf, rcounts, displs,  MPI_REAL8, root, MPI_COMM_WORLD, ierr)
```



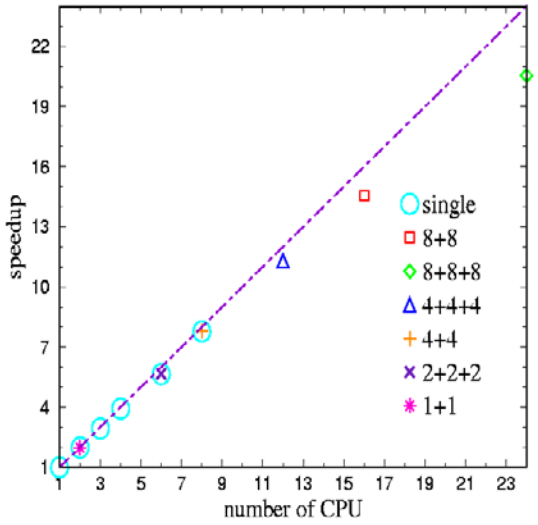

```
real*8 a(25), sbuf(MAX); integer displs(NX), scounts(NX), nsize
do i= 1, nsize
displs(i) = (i-1)*stride
scounts(i) = 25
enddo
call mpi_scatterv(sbuf, scounts, displs, MPI_REAL8, a, 25,  MPI_REAL, root, MPI_COMM_WOLRD, ierr)
```


http://www.cac.cornell.edu/Ranger/MPIcc/gathervscatterv.aspx

Parallel ADMD



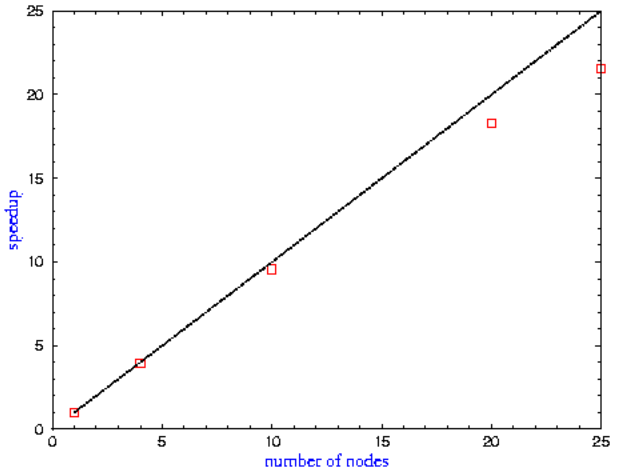
$$S = \Delta \sum_{j=0}^{P-1} \left\{ \sum_{I=1}^{N} \frac{M_I}{2} \left(\frac{\vec{R}_I^{\,j} - \vec{R}_I^{\,j+1}}{\Delta} \right)^2 - V(\{\vec{R}_I^{\,j}\}) \right\}$$



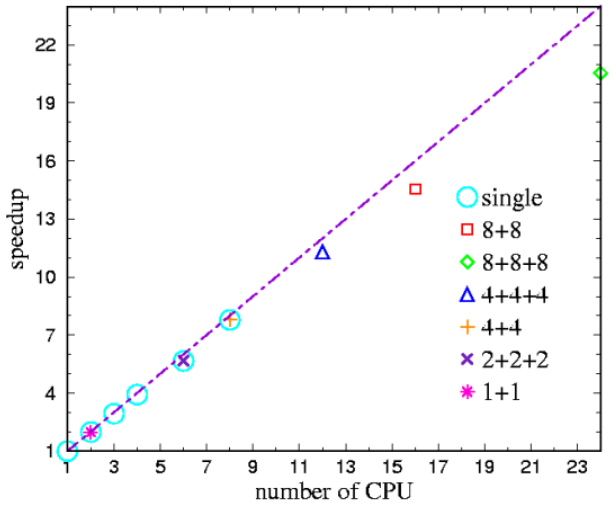
{Communication}/{CPU} ratio

Distributed computing

Observed speedup



top 각 노드에서 직접 확인



GLOBUS; MPICH_G2

```
        program main
        include 'mpif.h'
        integer ierr
        call MPI_INIT(ierr)
        print*, 'hello world'
        call MPI_FINALIZE(ierr)
        end
-------------------------------------------------------
mpif77 hello_world_parallel_1.f
mpiexec –n 2 ./a.out
mpiexec –np 2 ./a.out
mpirun –np 2 ./a.out
```

```
PROGRAM env
INCLUDE ’mpif.h’
CALL MPI_INIT(ierr)
CALL MPI_COMM_SIZE(MPI_COMM_WORLD, nprocs, ierr)
CALL MPI_COMM_RANK(MPI_COMM_WORLD, myrank, ierr)
PRINT *,’nprocs =’,nprocs,’myrank =’,myrank
CALL MPI_FINALIZE(ierr)
END
```

```
$ mpxlf env.f
** env === End of Compilation 1 ===
1501-510 Compilation successful for file env.f.
$ export MP_STDOUTMODE=ordered
$ export MP_LABELIO=yes
$ a.out -procs 3
0: nprocs = 3 myrank = 0
1: nprocs = 3 myrank = 1
2: nprocs = 3 myrank = 2
```


From Dr. Jongsoo Kim (KASI), The 1st CAC Summer School on Parallel Computing

```
integer isize
character*9 string,fname



isize=4
call xnumeral( myid, string, isize)
fname='conf1'//trim(string)                !   confi10000, confi10001, confi10002
open(29,file=fname,form='formatted')
write(29,*) nparticle,mm
write(29,*) tau,beta,ss_lowest
do ia=1,nparticle
do j=0,mm
write(29,*) (qqq(jcomp,j,ia),jcomp=1,idims)
enddo
enddo
do ia=1,nparticle
write(29,*) iiq(ia)
enddo
close(29)
```

ASCII vs. binary

> 10 speedup

Format	C	FORTRAN
ASCII	`fprintf()`	open(6,file='test',form='formatted') write(6,*)
Binary	`fwrite()`	open(6,file='test',form='unformatted') write(6)

포트란 direct access 기법이 MPI 환경하에서 유용한 입출력 방식으로 사용되는 한 예를 소개합니다.

포트란 문법에서 파일을 다룰 때, 디폴트는 순차적 (" sequential ") 접근을 의미한다.
읽을 때, 적을 때 마찬가지이다.
아래와 같은 경우, 통상 디폴트 옵션에 따라서 access='sequential' 는 생략하는 경우가 대부분이다.

```
open(1,file='fort.1',form='formatted',access='sequential')

open(2,file='fort.2',access='direct', recl=8*nnn)
```
위와 같이 할 경우, 즉 명시적으로 direct 접근을 활용한다고 선언하면, 레코드 길이를 정한 상태에서 데이터에 직접 접근할 수 있다. 즉, 읽고 적을 수 있다. 데이터 부분, 부분별로 적고 읽을 수 있다.
프로그램 실행 중에 데이터를 읽고 적을 수 있다. 데이터는 부분으로 나누어져 있어서 전부일 필요가 없다.

데이터 용량이 클 때, 유용하게 사용할 수 있는 기법이다.
사실상 하나의 배열처럼 프로그램에서 불러서 사용할 수 있다.

이 편리한 방법은 MPI 환경에서도 여전히 유용하다. 노드간 통신을 수행한 다음 0 번 노드가 파일 적기를 담당할 수 있다. 데이터 용량이 클 경우 노드가 클 수 있다. 하지만, 각 노드에서 적절히 배당된 레코드에 각자 적어 버리면, 나중에 이 단일 파일에 접근할 수 있다. 노드간 통신 없이 파일에 적을 수 있다.

노드별로 서로 다른 레코드에 직접 접근해서 각자 데이터를 적고 읽을 수 있다.

```
           <----------------------------레코드 길이---------------------------->
1번 레코드 : ㅁ ㅁ ㅁ ㅁ ㅁ ㅁ ㅁ ㅁ ㅁ ㅁ ㅁ ㅁ ㅁ ㅁ ㅁ ㅁ ㅁ ㅁ ㅁ ㅁ ㅁ ㅁ ㅁ ㅁ ㅁ ㅁ
2번 레코드 : ㅁ ㅁ ㅁ ㅁ ㅁ ㅁ ㅁ ㅁ ㅁ ㅁ ㅁ ㅁ ㅁ ㅁ ㅁ ㅁ ㅁ ㅁ ㅁ ㅁ ㅁ ㅁ ㅁ ㅁ ㅁ ㅁ
3번 레코드 : ㅁ ㅁ ㅁ ㅁ ㅁ ㅁ ㅁ ㅁ ㅁ ㅁ ㅁ ㅁ ㅁ ㅁ ㅁ ㅁ ㅁ ㅁ ㅁ ㅁ ㅁ ㅁ ㅁ ㅁ ㅁ ㅁ
4번 레코드 : ㅁ ㅁ ㅁ ㅁ ㅁ ㅁ ㅁ ㅁ ㅁ ㅁ ㅁ ㅁ ㅁ ㅁ ㅁ ㅁ ㅁ ㅁ ㅁ ㅁ ㅁ ㅁ ㅁ ㅁ ㅁ ㅁ
5번 레코드 : ㅁ ㅁ ㅁ ㅁ ㅁ ㅁ ㅁ ㅁ ㅁ ㅁ ㅁ ㅁ ㅁ ㅁ ㅁ ㅁ ㅁ ㅁ ㅁ ㅁ ㅁ ㅁ ㅁ ㅁ ㅁ ㅁ
--------
----
```

```
!234567890
program d_access
implicit none
include 'mpif.h'
integer myid, nproc, ierr, iroot, kount
real*8 time_start,time_end
integer nsites
real*8, allocatable :: spin_lattice(:),tspin_lattice(:)
real*8 before,after
integer kdum,kk
call MPI_INIT( ierr )
call MPI_COMM_RANK( MPI_COMM_WORLD, myid, ierr )
call MPI_COMM_SIZE( MPI_COMM_WORLD, nproc, ierr )
if(myid == 0 .and. nproc > 1) print *, nproc," processes are alive"
if(myid == 0 .and. nproc ==1) print *, nproc," process is alive"
time_start=MPI_WTIME()

nsites=100
allocate(spin_lattice(nsites))
allocate(tspin_lattice(nsites))
spin_lattice=0.d0
tspin_lattice=0.d0
before=float(myid)
print*, before, myid,' node,in the memory'
open(97,file='fort.97',access='direct',recl=8*(nsites+1))
write(97,rec=myid+1) before,(spin_lattice(kdum),kdum=1,nsites)
close(97)
```

포트란 프로그램 실행 중(on the fly) 특정 파일 지우기는 아래와 같이 실행하면 된다.

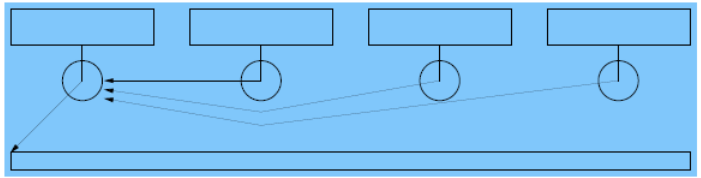
```
OPEN(11,FILE='del')
CLOSE(11,STATUS='DELETE') ! 파일 지우기를 실행함.
```

MPI 인 경우 0 번 노드에서만 지워야한다. 모든 노드가 다 지울 수 없다.

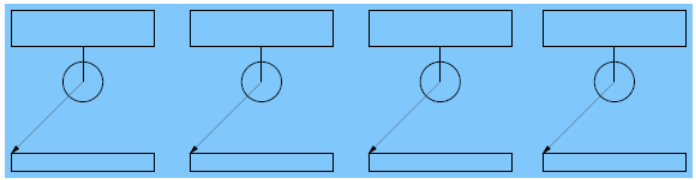
```
call MPI_BARRIER( MPI_COMM_WORLD, ierr )
if(myid == 0)then ! -----[ process id = 0
open(97,file='fort.97',access='direct',recl=8*(nsites+1))
before=2.d222
do kk=1,nproc
read(97,rec=kk) after,(tspin_lattice(kdum),kdum=1,nsites)
print*, after, kk-1,' node,in the file'
! if(before > after)then
! before=after
! spin_lattice=tspin_lattice
! endif
enddo
close(97)
! print*, before,' before'
endif                ! -------=== } process id =0

deallocate(spin_lattice)
deallocate(tspin_lattice)
time_end=MPI_WTIME()
if(myid == 0) then ! --------=== { process id =0
write(6,'(4(f14.5,1x,a))') (time_end-time_start),'s', (time_end-time_start)/60.d0,'m', (time_end-time_start)/3600.d0,'h',
(time_end-time_start)/3600.d0/24.d0,'d'
endif                    ! --------=== } process id =0
call MPI_FINALIZE(ierr)
stop
end program d_access
```

I/O → Parallel I/O

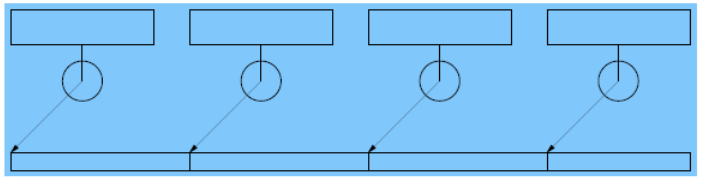
- Non-parallel I/O



- I/O to separate files



- Parallel I/O



Non-parallel I/O from an MPI program

```
#include "mpi.h"
#include <stdio.h>
#define BUFSIZE 100
int main(int argc, char *argv[])
{
int i, myrank, numprocs, buf[BUFSIZE];
MPI_Status status;
FILE *myfile;
MPI_Init(&argc, &argv);
MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
MPI_Comm_size(MPI_COMM_WORLD, &numprocs);
for (i=0; i<BUFSIZE; i++)
buf[i] = myrank * BUFSIZE + i;
if (myrank != 0)
MPI_Send(buf, BUFSIZE, MPI_INT, 0, 123, MPI_COMM_WORLD);
else {
myfile = fopen("testfile", "w");
fwrite(buf, sizeof(int), BUFSIZE, myfile);
for (i=1; i<numprocs; i++) {
MPI_Recv(buf, BUFSIZE, MPI_INT, i, 123, MPI_COMM_WORLD, &status);
fwrite(buf, sizeof(int), BUFSIZE, myfile);
}
fclose(myfile);
}
MPI_Finalize();
return 0;
}
```

Non-MPI I/O to separate files

```
#include "mpi.h"
#include <stdio.h>
#define BUFSIZE 100
int main(int argc, char *argv[])
{
int i, myrank, buf[BUFSIZE];
char filename[128];
FILE *myfile;
MPI_Init(&argc, &argv);
MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
for (i=0; i<BUFSIZE; i++)
buf[i] = myrank * BUFSIZE + i;
sprintf(filename, "testfile.%d", myrank);
myfile = fopen(filename, "w");
fwrite(buf, sizeof(int), BUFSIZE, myfile);
fclose(myfile);
MPI_Finalize();
return 0;
}
```

MPI I/O to separate files

```
#include "mpi.h"
#include <stdio.h>
#define BUFSIZE 100
int main(int argc, char *argv[])
{
int i, myrank, buf[BUFSIZE];
char filename[128];
MPI_File myfile;
MPI_Init(&argc, &argv);
MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
for (i=0; i<BUFSIZE; i++)
buf[i] = myrank * BUFSIZE + i;
sprintf(filename, "testfile.%d", myrank);
MPI_File_open(MPI_COMM_SELF, filename, MPI_MODE_WRONLY | MPI_MODE_CREATE, MPI_INFO_NULL, &myfile);
MPI_File_write(myfile, buf, BUFSIZE, MPI_INT, MPI_STATUS_IGNORE);
MPI_File_close(&myfile);
MPI_Finalize();
return 0;
}
```

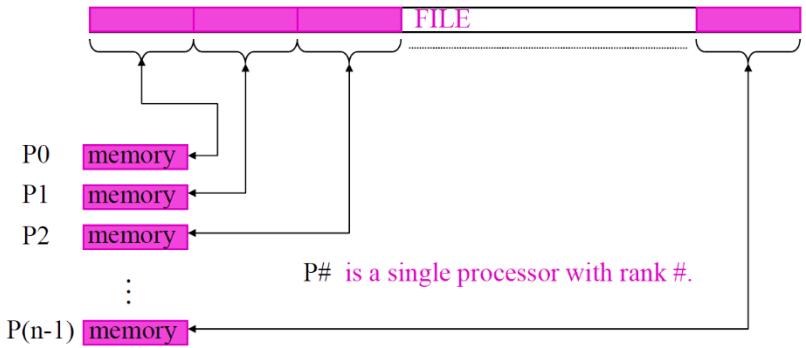
Parallel MPI I/O to a single file

```
#include "mpi.h"
#include <stdio.h>
#define BUFSIZE 100
int main(int argc, char *argv[])
{
int i, myrank, buf[BUFSIZE];
MPI_File thefile;
MPI_Init(&argc, &argv);
MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
for (i=0; i<BUFSIZE; i++)
buf[i] = myrank * BUFSIZE + i;
MPI_File_open(MPI_COMM_WORLD, "testfile", MPI_MODE_CREATE | MPI_MODE_WRONLY, MPI_INFO_NULL, &thefile);
MPI_File_set_view(thefile, myrank*BUFSIZE*sizeof(int), MPI_INT, MPI_INT, "native", MPI_INFO_NULL);
MPI_File_write(thefile, buf, BUFSIZE, MPI_INT, MPI_STATUS_IGNORE);
MPI_File_close(&thefile);
MPI_Finalize();
return 0;
}
```

Declaring a File Pointer

| : C

+ : Fortran



Reading a file (1/3)

```
#include "mpi.h"
#include <stdio.h>
int main(int argc, char *argv[])
{
int myrank, numprocs, bufsize, *buf, count;
MPI_File thefile;                                   Declaring a File Pointer
MPI_Status status;
MPI_Offset filesize;
MPI_Init(&argc, &argv);
MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
MPI_Comm_size(MPI_COMM_WORLD, &numprocs);
MPI_File_open(MPI_COMM_WORLD, "testfile", MPI_MODE_RDONLY, MPI_INFO_NULL, &thefile);
MPI_File_get_size(thefile, &filesize);          /* in bytes */
filesize = filesize / sizeof(int);              /* in number of ints */
bufsize = filesize / numprocs + 1;              /* local number to read */
buf = (int *) malloc (bufsize*sizeof(int));
MPI_File_set_view(thefile, myrank*bufsize*sizeof(int), MPI_INT, MPI_INT, "native", MPI_INFO_NULL);
MPI_File_read(thefile, buf, bufsize, MPI_INT, &status);
MPI_Get_count(&status, MPI_INT, &count);
printf("process %d read %d ints\n", myrank, count);
MPI_File_close(&thefile);
MPI_Finalize();
return 0;
}
```

Reading a file (2/3)

```
#include<stdio.h>
#include "mpi.h"
#define FILESIZE 80
int main(int argc, char **argv){
int rank, size, bufsize, nints;
MPI_File fh;
MPI_Status status;
MPI_Init(&argc, &argv);
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
MPI_Comm_size(MPI_COMM_WORLD, &size);
bufsize = FILESIZE/size;
nints = bufsize/sizeof(int);
int buf[nints];
MPI_File_open(MPI_COMM_WORLD,"dfile",MPI_MODE_RDONLY,MPI_INFO_NULL,&fh);
MPI_File_seek(fh, rank * bufsize, MPI_SEEK_SET);
MPI_File_read(fh, buf, nints, MPI_INT, &status);
printf("\nrank: %d, buf[%d]: %d", rank, rank*bufsize, buf[0]);
MPI_File_close(&fh);
MPI_Finalize();
return 0;
}
```

Declaring a File Pointer

Calculating Buffer Size

After opening the file, read data from files by using either MPI_File_seek & MPI_File_read or MPI_File_read_at

Reading a file (3/3)

```
#include<stdio.h>
#include "mpi.h"
#define FILESIZE 80
int main(int argc, char **argv){
int rank, size, bufsize, nints;
MPI_File fh;
MPI_Status status;
MPI_Init(&argc, &argv);
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
MPI_Comm_size(MPI_COMM_WORLD, &size);
bufsize = FILESIZE/size;
nints = bufsize/sizeof(int);
int buf[nints];
MPI_File_open(MPI_COMM_WORLD,"dfile",MPI_MODE_RDONLY,MPI_INFO_NULL,&fh);
MPI_File_read_at(fh, rank*bufsize, buf, nints, MPI_INT, &status);
printf("\nrank: %d, buf[%d]: %d", rank, rank*bufsize, buf[0]);
MPI_File_close(&fh);
MPI_Finalize();
return 0;
}
```

Declaring a File Pointer

After opening the file, read data from files by using either MPI_File_seek & MPI_File_read or MPI_File_read_at

Writing a file (1/2)

```
#include<stdio.h>
#include "mpi.h"
int main(int argc, char **argv){
int i, rank, size, offset, nints, N=16 ;
MPI_File fhw;
MPI_Status status;
MPI_Init(&argc, &argv);
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
MPI_Comm_size(MPI_COMM_WORLD, &size);
int buf[N];
for ( i=0;i<N;i++){
buf[i] = i ;
 }
 offset = rank*(N/size)*sizeof(int);
MPI_File_open(MPI_COMM_WORLD, "datafile", MPI_MODE_CREATE|MPI_MODE_WRONLY, MPI_INFO_NULL, &fhw);
printf("\nRank: %d, Offset: %d\n", rank, offset);
MPI_File_write_at(fhw, offset, buf, (N/size), MPI_INT, &status);
MPI_File_close(&fhw);
MPI_Finalize();
return 0;
}
```

Declaring a File Pointer

For writing, use either **MPI_File_set_view** & **MPI_File_write** or **MPI_File_write_at**

Writing a file (2/2)

```
#include<stdio.h>
#include "mpi.h"
int main(int argc, char **argv){
 int i, rank, size, offset, nints, N=16;
 MPI_File fhw;
 MPI_Status status;
 MPI_Init(&argc, &argv);
 MPI_Comm_rank(MPI_COMM_WORLD, &rank);
 MPI_Comm_size(MPI_COMM_WORLD, &size);
 int buf[N];
 for ( i=0;i<N;i++){
 buf[i] = i ;
 }
 offset = rank*(N/size)*sizeof(int);
MPI_File_open(MPI_COMM_WORLD, "datafile3", MPI_MODE_CREATE|MPI_MODE_WRONLY, MPI_INFO_NULL, &fhw);
 printf("\nRank: %d, Offset: %d\n", rank, offset);
 MPI_File_set_view(fhw, offset, MPI_INT, MPI_INT, "native", MPI_INFO_NULL);
 MPI_File_write(fhw, buf, (N/size), MPI_INT, &status);
 MPI_File_close(&fhw);
 MPI_Finalize();
 return 0;
}
```

Declaring a File Pointer

For writing, use either **MPI_File_set_view** & **MPI_File_write** or **MPI_File_write_at**

Reading a file (FORTRAN)

```
integer istatus(MPI_STATUS_SIZE)
integer (kind=MPI_OFFSET_KIND) ioffset
call MPI_FILE_OPEN(MPI_COMM_WORLD,'/pfs/datafile', MPI_MODE_RDONLY, MPI_INFO_NULL, fh, ierr)
nints = FILESIZE / (nprocs*INTSIZE)
ioffset =rank*nints*INTSIZE
call MPI_FILE_READ_AT(fh, ioffset, buf, nints, MPI_INTEGER, istatus, ierr)
call MPI_GET_COUNT(istatus, MPI_INTEGER, count, ierr)
print *, 'process ', rank, 'read ', count,'integers'
call MPI_FILE_CLOSE(fh, ierr)
```

MPI_File_open flags:

- `MPI_MODE_RDONLY` (read only)
- `MPI_MODE_WRONLY` (write only)
- `MPI_MODE_RDWR` (read and write)
- `MPI_MODE_CREATE` (create file if it doesn't exist)
- Use bitwise-or '|' in C, or addition '+" in Fortran, to combine multiple flags

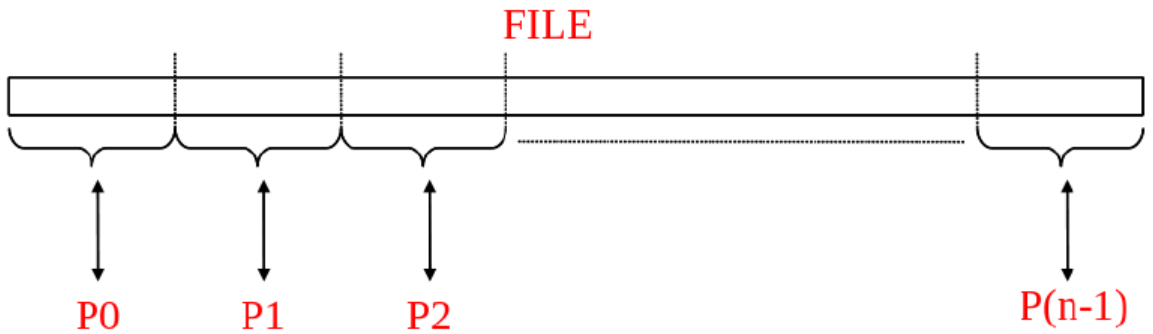
To write into a file, use MPI_File_write or MPI_File_write_at, or…

Writing to a file (FORTRAN)

```
PROGRAM main
use mpi
integer ierr, i, myrank, BUFSIZE, thefile
parameter (BUFSIZE=100)
integer buf(BUFSIZE)
integer(kind=MPI_OFFSET_KIND) disp
call MPI_INIT(ierr)
call MPI_COMM_RANK(MPI_COMM_WORLD, myrank, ierr)
do i = 0, BUFSIZE
buf(i) = myrank * BUFSIZE + i
enddo

call MPI_FILE_OPEN(MPI_COMM_WORLD, 'testfile', MPI_MODE_WRONLY + MPI_MODE_CREATE, MPI_INFO_NULL, thefile, ierr)
call MPI_TYPE_SIZE(MPI_INTEGER, intsize, ierr)
disp = myrank * BUFSIZE * intsize
call MPI_FILE_SET_VIEW(thefile, disp, MPI_INTEGER, MPI_INTEGER, 'native', MPI_INFO_NULL, ierr)
call MPI_FILE_WRITE(thefile, buf, BUFSIZE, MPI_INTEGER,MPI_STATUS_IGNORE, ierr)
call MPI_FILE_CLOSE(thefile, ierr)
call MPI_FINALIZE(ierr)
END PROGRAM main
```

- What is Parallel I/O?

Multiple processes of a parallel program accessing data (reading or writing) from a common file.



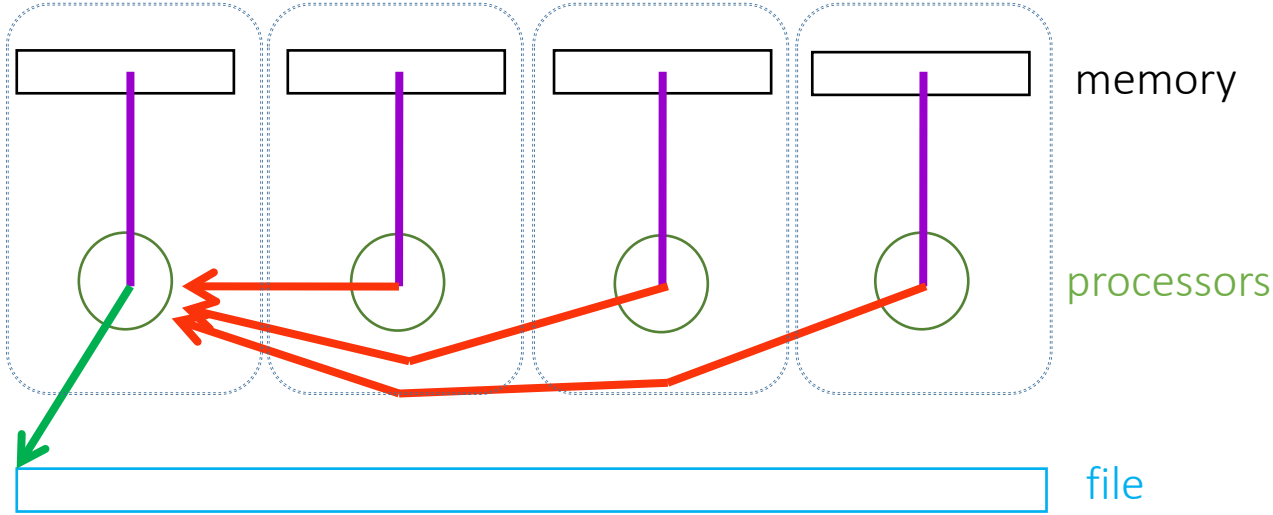
C: int MPI_File_open (MPI_Comm comm, char *filename, int amode, MPI_Info info, MPI_File *fh)
Fortran: MPI_FILE_OPEN(COMM, FILENAME, AMODE, INFO, FH, IERROR) CHARACTER*(*) FILENAME INTEGER COMM, AMODE, INFO, FH, IERROR
Python: Open(type cls, Intracomm comm, filename, int amode=MODE_RDONLY, Info info=INFO_NULL)

C: MPI_File file;
Fortran: Type(MPI_File) file
Python: MPI.File

https://pages.tacc.utexas.edu/~eijkhout/pcse/html/mpi-io.html

Parallel I/O (1/3)

Sequential I/O from an parallel program

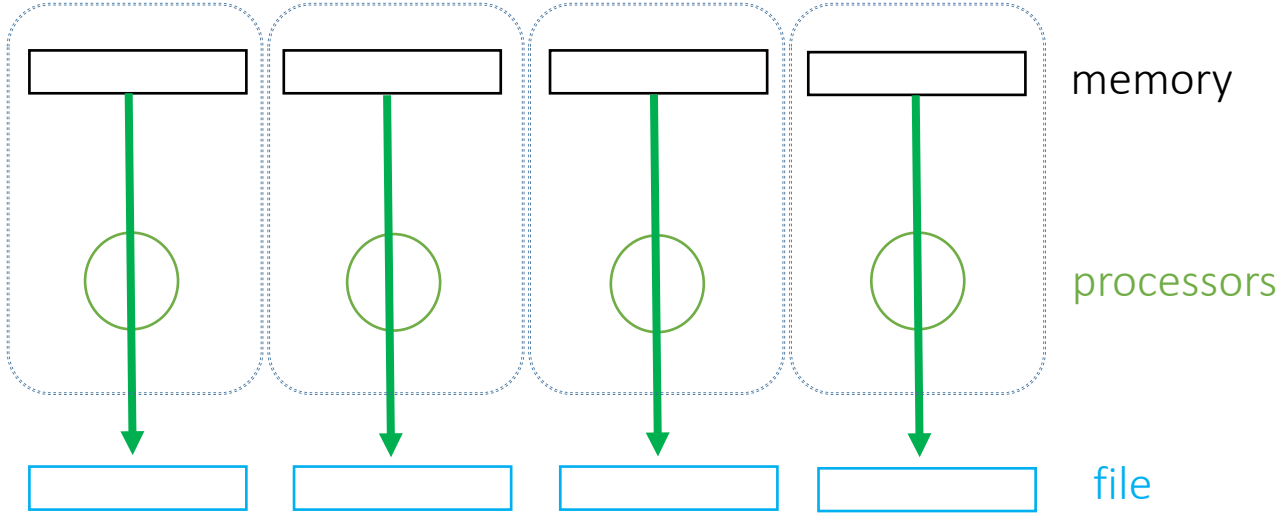


Send : write

Recv : read

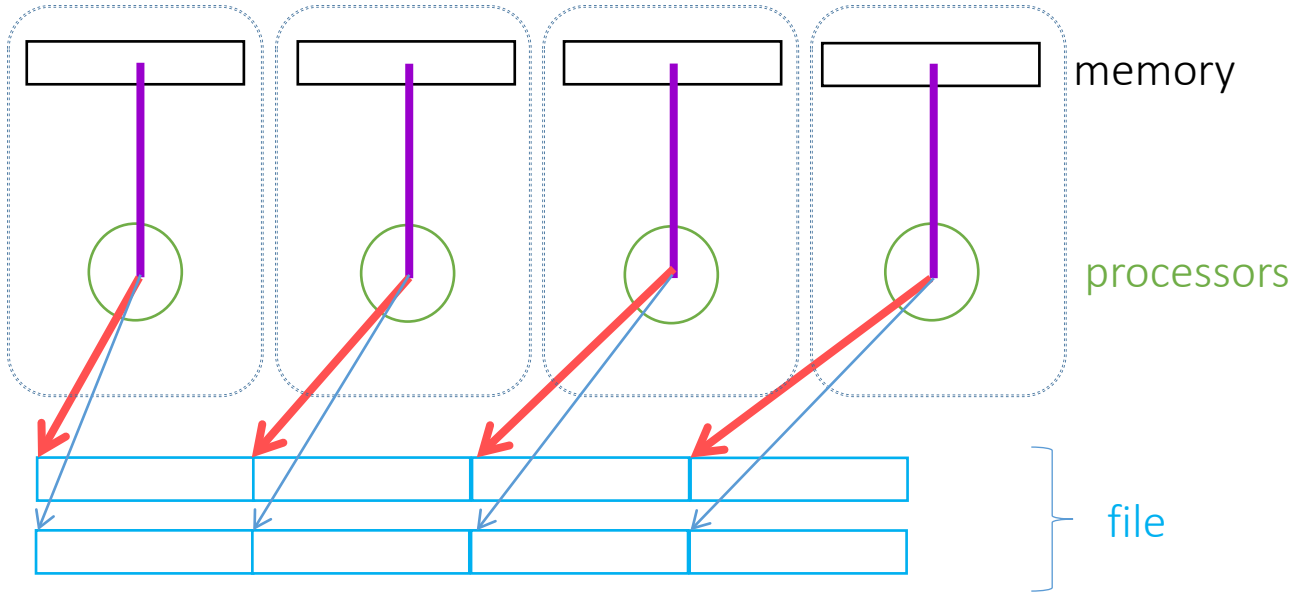
Parallel I/O (2/3)

Parallel I/O to multiple files



Parallel I/O (3/3)

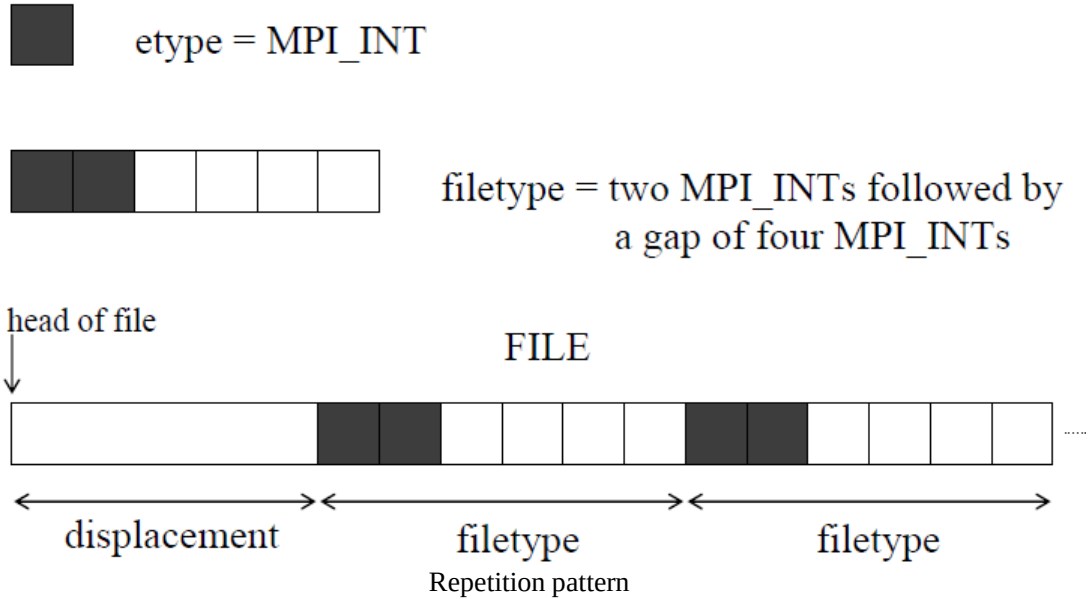
Parallel I/O to a single file

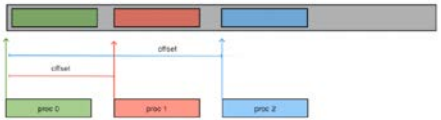


parallel MPI write into multiple files

```
!  ! example of parallel MPI write into multiple files
PROGRAM main
    ! Fortran 90 users can (and should) use
    !     use mpi
    ! instead of include 'mpif.h' if their MPI implementation provides a
    ! mpi module.
    include 'mpif.h'

    integer ierr, i, myrank, BUFSIZE, thefile
    parameter (BUFSIZE=100)
    integer buf(BUFSIZE)
    character*12 ofname

    call MPI_INIT(ierr)
    call MPI_COMM_RANK(MPI_COMM_WORLD, myrank, ierr)

    do i = 1, BUFSIZE
       buf(i) = myrank * BUFSIZE + i
    enddo
    write(ofname,'(a8,i4.4)') 'testfile',myrank
    open(unit=11,file=ofname,form='unformatted')
    write(11) buf

    call MPI_FINALIZE(ierr)
END PROGRAM main
```

Specified by a triplet (*displacement*, *etype*, and *filetype*) passed to **MPI_File_set_view**





```
! example of parallel MPI write into a single file, in Fortran
    PROGRAM main
    ! Fortran 90 users can (and should) use
    !     use mpi
    ! instead of include 'mpif.h' if their MPI implementation provides a
    ! mpi module.
    include 'mpif.h'

    integer ierr, i, myrank, BUFSIZE, thefile
    parameter (BUFSIZE=100)
    integer buf(BUFSIZE)
    integer(kind=MPI_OFFSET_KIND) disp

    call MPI_INIT(ierr)
    call MPI_COMM_RANK(MPI_COMM_WORLD, myrank, ierr)

    do i = 0, BUFSIZE
       buf(i) = myrank * BUFSIZE + i
    enddo
    call MPI_FILE_OPEN(MPI_COMM_WORLD, 'testfile', MPI_MODE_WRONLY +   MPI_MODE_CREATE,   MPI_INFO_NULL, thefile, ierr)
    ! assume 4-byte integers
    disp = myrank * BUFSIZE * 4
    call MPI_FILE_SET_VIEW(thefile, disp, MPI_INTEGER,  MPI_INTEGER, 'native', MPI_INFO_NULL, ierr)
    call MPI_FILE_WRITE(thefile, buf, BUFSIZE, MPI_INTEGER, MPI_STATUS_IGNORE, ierr)
    call MPI_FILE_CLOSE(thefile, ierr)
    call MPI_FINALIZE(ierr)

END PROGRAM main
```

MPI_MODE_RDWR
MPI_MODE_RDONLY
MPI_MODE_WRONLY
MPI_MODE_CREATE

| : C
+ : Fortran

For writing, use either **MPI_File_set_view** & **MPI_File_write** or **MPI_File_write_at**

MPI_MODE_CREATE | MPI_MODE_WRONLY
MPI_MODE_WRONLY + MPI_MODE_CREATE

http://www.mcs.anl.gov/research/projects/mpi/usingmpi2/examples/starting/io3f_f90.htm

```
program write_individual_pointer
use mpi
implicit none
integer ier,nproc,myid,ifile,intsize
integer mode,info,ietype,ifiletype
integer istatus(MPI_STATUS_SIZE)
integer (kind=MPI_OFFSET_KIND) :: idisp
integer, parameter :: kount=100
integer ibuf(kount)
integer i
real*8 aa1,aa2

call MPI_INIT(ier)
call MPI_COMM_RANK(MPI_COMM_WORLD, myid, ier)
call MPI_COMM_SIZE(MPI_COMM_WORLD, nproc, ier)
mode=ior(MPI_MODE_CREATE,MPI_MODE_WRONLY)
info=0
call MPI_TYPE_EXTENT(MPI_INTEGER,intsize,ier)
aa1=MPI_WTIME()
do i=1,kount
ibuf(i)=myid*kount+i
enddo
CALL MPI_FILE_OPEN(MPI_COMM_WORLD, 'test',mode, info, ifile, ier)
idisp=myid* kount* intsize
ietype=MPI_INTEGER
ifiletype=MPI_INTEGER
CALL MPI_FILE_SET_VIEW(ifile,idisp,ietype,ifiletype,'native',info, ier)
CALL MPI_FILE_WRITE(ifile,myid,1,MPI_INTEGER,istatus, ier)

write(6,*) 'hello form myid',myid,'i wrote:', myid,'.'
CALL MPI_FILE_CLOSE(ifile,ier)
aa2=MPI_WTIME()
call MPI_FINALIZE(ier)
end program write_individual_pointer
```

```
program write_individual_pointer
use mpi
implicit none
integer ier,nproc,myid,ifile,intsize
integer mode,info,ietype,ifiletype
integer istatus(MPI_STATUS_SIZE)
integer (kind=MPI_OFFSET_KIND) :: idisp
integer, parameter :: kount=100
integer ibuf(kount)
integer i,itest
real*8 aa1,aa2

call MPI_INIT(ier)
call MPI_COMM_RANK(MPI_COMM_WORLD, myid, ier)
call MPI_COMM_SIZE(MPI_COMM_WORLD, nproc, ier)
mode=MPI_MODE_RDONLY
info=0
call MPI_TYPE_EXTENT(MPI_INTEGER,intsize,ier)
aa1=MPI_WTIME()
do i=1,kount
ibuf(i)=myid*kount+i
enddo
CALL MPI_FILE_OPEN(MPI_COMM_WORLD, 'test',mode, info, ifile, ier)
idisp=myid* kount* intsize
ietype=MPI_INTEGER
ifiletype=MPI_INTEGER
CALL MPI_FILE_SET_VIEW(ifile,idisp,ietype,ifiletype,'native',info, ier)
CALL MPI_FILE_READ(ifile,itest,1,MPI_INTEGER,istatus, ier)
write(6,*) 'hello form myid',myid,'i read:', itest,'.'
CALL MPI_FILE_CLOSE(ifile,ier)

aa2=MPI_WTIME()
call MPI_FINALIZE(ier)
end program write_individual_pointer
```

```c
#include<stdio.h>
#include "mpi.h"
#define FILESIZE 80
int main(int argc, char **argv){
  int rank, size, bufsize, nints;
  MPI_File fh;
  MPI_Status status;
  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  MPI_Comm_size(MPI_COMM_WORLD, &size);
  bufsize = FILESIZE/size;
  nints = bufsize/sizeof(int);
  int buf[nints];
  MPI_File_open(MPI_COMM_WORLD,"dfile",MPI_MODE_RDONLY,MPI_INFO_NULL,&fh);
  MPI_File_seek(fh, rank * bufsize, MPI_SEEK_SET);
  MPI_File_read(fh, buf, nints, MPI_INT, &status);
  printf("\nrank: %d, buf[%d]: %d", rank, rank*bufsize, buf[0]);
  MPI_File_close(&fh);
  MPI_Finalize();
  return 0;
}
```

Declaring a File Pointer

Calculating Buffer Size

Opening a File

File seek &

Read

Closing a File

https://www.tacc.utexas.edu/documents/13601/900558/MPI-IO-Final.pdf/eea9d7d3-4b81-471c-b244-41498070e35d

```
#include<stdio.h>
#include "mpi.h"
#define FILESIZE 80
int main(int argc, char **argv){
  int rank, size, bufsize, nints;
  MPI_File fh;
  MPI_Status status;
  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  MPI_Comm_size(MPI_COMM_WORLD, &size);
  bufsize = FILESIZE/size;
  nints = bufsize/sizeof(int);
  int buf[nints];
  MPI_File_open(MPI_COMM_WORLD,"dfile",MPI_MODE_RDONLY,MPI_INFO_NULL,&fh);
  MPI_File_read_at(fh, rank*bufsize, buf, nints, MPI_INT, &status);
  printf("\nrank: %d, buf[%d]: %d", rank, rank*bufsize, buf[0]);
  MPI_File_close(&fh);
  MPI_Finalize();
  return 0;
}
```

Combining File Seek & Read in One Step for Thread Safety in **MPI_File_read_at**

https://www.tacc.utexas.edu/documents/13601/900558/MPI-IO-Final.pdf/eea9d7d3-4b81-471c-b244-41498070e35d

```
#include <stdio.h>
#include "mpi.h"
int main(int argc, char **argv){
int i, rank, nproc, offset, nints, N=16;
MPI_File  fhw;
MPI_Status  status;
MPI_Init(&argc, &argv);
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
MPI_Comm_size(MPI_COMM_WORLD, &nproc);
int buf[N];
for ( i=0; i < N; i++) { buf[i] = i; }
offset=rank*(N/nproc)*sizeof(int);
MPI_File_open(MPI_COMM_WORLD, "datafile3", MPI_MODE_CREATE|MPI_MODE_WRONLY,MPI_INFO_NULL, &fhw);
printf("\nRank: %d, Offset: %d\n", rank,offset);
MPI_File_set_view(fhw ,buf, offset, MPI_INT, MPI_INT, "native", MPI_INFO_NULL);
MPI_File_write(fhw,  buf, (N/nproc), MPI_INT,&status);
MPI_Finalize(); return 0; }
```

https://www.tacc.utexas.edu/documents/13601/900558/MPI-IO-Final.pdf/eea9d7d3-4b81-471c-b244-41498070e35d

```
PROGRAM bcast
INCLUDE ’mpif.h’
INTEGER imsg(4)
CALL MPI_INIT(ierr)
CALL MPI_COMM_SIZE(MPI_COMM_WORLD, nprocs, ierr)
CALL MPI_COMM_RANK(MPI_COMM_WORLD, myrank, ierr)
IF (myrank==0) THEN
DO i=1,4
imsg(i) = i
ENDDO
                    ELSE
DO i=1,4
imsg(i) = 0
ENDDO
                    ENDIF
 PRINT *,’Before:’,imsg
 CALL MP_FLUSH(1)
 CALL MPI_BCAST(imsg, 4, MPI_INTEGER,  0, MPI_COMM_WORLD, ierr)
 PRINT *,’After :’,imsg
 CALL MPI_FINALIZE(ierr)
 END
```

```
$ a.out -procs 3
0: Before: 1 2 3 4
1: Before: 0 0 0 0
2: Before: 0 0 0 0
0: After : 1 2 3 4
1: After : 1 2 3 4
2: After : 1 2 3 4
```

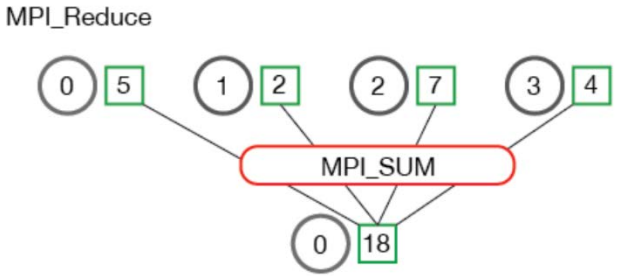
```
PROGRAM gather
INCLUDE ’mpif.h’
INTEGER irecv(3)
CALL MPI_INIT(ierr)
CALL MPI_COMM_SIZE(MPI_COMM_WORLD, nprocs, ierr)
CALL MPI_COMM_RANK(MPI_COMM_WORLD, myrank, ierr)
isend = myrank + 1
CALL MPI_GATHER(isend, 1, MPI_INTEGER, irecv, 1, MPI_INTEGER, 0, MPI_COMM_WORLD, ierr)
IF (myrank==0) THEN
PRINT *,’irecv =’,irecv
                    ENDIF
CALL MPI_FINALIZE(ierr)
END
```

mpirun –np 3 ./a.out

```
$ a.out -procs 3
0: irecv = 1 2 3
```

```
/*gather*/

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

void main (int argc, char *argv[]){

  int i, nprocs, myrank ;
  int isend, irecv[3];

  MPI_Init(&argc, &argv);
  MPI_Comm_size(MPI_COMM_WORLD, &nprocs);
  MPI_Comm_rank(MPI_COMM_WORLD, &myrank);

  isend = myrank + 1;
  MPI_Gather(&isend,1,MPI_INT,irecv,1,MPI_INT,0, MPI_COMM_WORLD);
  if(myrank == 0) {
     printf(“ irecv = ”);
     for(i=0; i<3; i++)
        printf(“ %d”, irecv[i]); printf(“\n”);
  }
  MPI_Finalize();
}
```

```
 PROGRAM reduce
 INCLUDE 'mpif.h'
 REAL a(9)
 CALL MPI_INIT(ierr)
 CALL MPI_COMM_SIZE(MPI_COMM_WORLD, nprocs, ierr)
 CALL MPI_COMM_RANK(MPI_COMM_WORLD, myrank, ierr)
 ista = myrank * 3 + 1
 iend = ista + 2
 DO i=ista,iend
 a(i) = i
ENDDO
sum = 0.0
DO i=ista,iend
sum = sum + a(i)
ENDDO
CALL MPI_REDUCE(sum, tmp, 1, MPI_REAL, MPI_SUM, 0,MPI_COMM_WORLD, ierr)
sum = tmp
IF (myrank==0) THEN
PRINT *,'sum =',sum
                    ENDIF
CALL MPI_FINALIZE(ierr)
END
```




```
$ a.out -procs 3
0: sum = 45.00000000
```

```
PROGRAM maxloc_p
INCLUDE ’mpif.h’
INTEGER n(9)
INTEGER isend(2), irecv(2)
DATA n /12, 15, 2, 20, 8, 3, 7, 24, 52/
CALL MPI_INIT(ierr)
CALL MPI_COMM_SIZE(MPI_COMM_WORLD, nprocs, ierr)
CALL MPI_COMM_RANK(MPI_COMM_WORLD, myrank, ierr)
 ista = myrank * 3 + 1
 iend = ista + 2
 imax = -999
 DO i = ista, iend
 IF (n(i) > imax) THEN
 imax = n(i)
 iloc = i
                    ENDIF
 ENDDO
 isend(1) = imax
 isend(2) = iloc
  CALL MPI_REDUCE(isend, irecv, 1, MPI_2INTEGER, MPI_MAXLOC, 0, MPI_COMM_WORLD, ierr)
  IF (myrank == 0) THEN
  PRINT *, ’Max =’, irecv(1), ’Location =’, irecv(2)
                      ENDIF
  CALL MPI_FINALIZE(ierr)
  END
```

```
$ a.out -procs 3
0: Max = 52 Location = 9
```

```
      SUBROUTINE para_range(n1, n2, nprocs, irank, ista, iend)
      iwork = (n2 - n1) / nprocs + 1
      ista = MIN(irank * iwork + n1, n2 + 1)
      iend = MIN(ista + iwork - 1, n2)
      END

PROGRAM main
PARAMETER (n = 1000)
DIMENSION a(n)
DO i = 1, n
a(i) = i
ENDDO
sum = 0.0
DO i = 1, n
sum = sum + a(i)
ENDDO
PRINT *,'sum =',sum
END
```

```
PROGRAM main
INCLUDE ’mpif.h’
PARAMETER (n = 1000)
DIMENSION a(n)
CALL MPI_INIT(ierr)
CALL MPI_COMM_SIZE(MPI_COMM_WORLD, nprocs, ierr)
CALL MPI_COMM_RANK(MPI_COMM_WORLD, myrank, ierr)
CALL para_range(1, n, nprocs, myrank, ista, iend)
DO i = ista, iend
a(i) = i
ENDDO
sum = 0.0
DO i = ista, iend
sum = sum + a(i)
ENDDO
CALL MPI_REDUCE(sum, ssum, 1, MPI_REAL, MPI_SUM, 0, MPI_COMM_WORLD, ierr)
sum = ssum
IF (myrank == 0) PRINT *,’sum =’,sum
CALL MPI_FINALIZE(ierr)
END
```

```
DO i = n1, n2
computation
ENDDO

DO i = n1 + myrank, n2, nprocs
computation
ENDDO
```

```
PROGRAM main
INCLUDE ’mpif.h’
PARAMETER (n = 1000)
DIMENSION a(n)
CALL MPI_INIT(ierr)
CALL MPI_COMM_SIZE(MPI_COMM_WORLD, nprocs, ierr)
CALL MPI_COMM_RANK(MPI_COMM_WORLD, myrank, ierr)
DO i = 1 + myrank, n, nprocs
 a(i) = i
ENDDO
sum = 0.0
DO i = 1 + myrank, n, nprocs
sum = sum + a(i)
ENDDO
CALL MPI_REDUCE(sum, ssum, 1, MPI_REAL, MPI_SUM, 0, MPI_COMM_WORLD, ierr)
sum = ssum
PRINT *,’sum =’,sum
CALL MPI_FINALIZE(ierr)
END
```

```
DO i = n1, n2
computation
ENDDO
```

```
DO ii = n1 + myrank * iblock, n2, nprocs * iblock
DO i = ii, MIN(ii + iblock - 1, n2)
computation
ENDDO
ENDDO
```

shrink

FORTRAN 90

```
PROGRAM main
INCLUDE ’mpif.h’
PARAMETER (n1 = 1, n2 = 1000)
REAL, ALLOCATABLE :: a(:)
CALL MPI_INIT(ierr)
CALL MPI_COMM_SIZE(MPI_COMM_WORLD, nprocs, ierr)
CALL MPI_COMM_RANK(MPI_COMM_WORLD, myrank, ierr)
CALL para_range(n1, n2, nprocs, myrank, ista, iend)
ALLOCATE (a(ista:iend))
DO i = ista, iend
a(i) = i
ENDDO
sum = 0.0
DO i = ista, iend
sum = sum + a(i)
ENDDO
DEALLOCATE (a)
CALL MPI_REDUCE(sum, ssum, 1, MPI_REAL,MPI_SUM, 0, MPI_COMM_WORLD, ierr)
sum = ssum
PRINT *,’sum =’,sum
CALL MPI_FINALIZE(ierr)
END
```

shrink

FORTRAN 90

```
      PROGRAM main
      implicit none
      real sum1,ssum
      integer i,ista,iend
      integer ierr,n1,n2,nprocs,myrank
      INCLUDE 'mpif.h'
      PARAMETER (n1 = 1, n2 = 1000)
      REAL, ALLOCATABLE :: a(:)

      CALL MPI_INIT(ierr)
      CALL MPI_COMM_SIZE(MPI_COMM_WORLD, nprocs, ierr)
      CALL MPI_COMM_RANK(MPI_COMM_WORLD, myrank, ierr)
      CALL para_range(n1, n2, nprocs, myrank, ista, iend)
      ALLOCATE (a(ista:iend))
      DO i = ista, iend
      a(i) = i
      ENDDO
!     sum1 = 0.0
!     DO i = ista, iend
!     sum1 = sum1 + a(i)
!    ENDDO
     sum1=sum(a)
      DEALLOCATE (a)
     CALL MPI_REDUCE(sum1, ssum, 1, MPI_REAL,MPI_SUM, 0, MPI_COMM_WORLD, ierr)
     sum1 = ssum
     PRINT *,'sum1 =',sum1, myrank
     CALL MPI_FINALIZE(ierr)
     END
```

```
      subroutine equal_load(n1,n2,nproc,myid,istart,ifinish)
!     Written by In-Ho Lee, KRISS, September (2006)
      implicit none
      integer nproc,myid,istart,ifinish,n1,n2
      integer iw1,iw2
      iw1=(n2-n1+1)/nproc ; iw2=mod(n2-n1+1,nproc)
      istart=myid*iw1+n1+min(myid,iw2)
      ifinish=istart+iw1-1 ; if(iw2 > myid) ifinish=ifinish+1
!     print*, n1,n2,myid,nproc,istart,ifinish
      if(n2 < istart) ifinish=istart-1
      return
      end
```

```
!234567890
implicit none
include 'mpif.h'
integer, allocatable :: isend(:), irecv(:)
integer, allocatable :: ircnt(:), idisp(:)
integer ierr,nproc,myid
integer i,iscnt
integer ndsize

call MPI_INIT(ierr)
call MPI_COMM_SIZE(MPI_COMM_WORLD,nproc,ierr)
call MPI_COMM_RANK(MPI_COMM_WORLD,myid,ierr)
ndsize=10
allocate(isend(ndsize),irecv(nproc*ndsize))
allocate(ircnt(0:nproc-1),idisp(0:nproc-1))
ircnt=ndsize
idisp(0)=0
do i=1,nproc-1
idisp(i)=idisp(i-1)+ircnt(i)
enddo
do i=1,ndsize          ! node specific data with a data-size ndsize
isend(i)=myid+1
enddo
iscnt=ndsize
call MPI_GATHERV(isend, iscnt, MPI_INTEGER, irecv, ircnt, idisp, MPI_INTEGER,0, MPI_COMM_WORLD, ierr)
if(myid == 0)then
print*, 'irecv= ',irecv
endif
deallocate(ircnt,idisp)
deallocate(isend,irecv)
call MPI_FINALIZE(ierr)
stop
end
```

```
irecv= 1 1 1 1 1
1 1 1 1 1 2
2 2 2 2 2 2
2 2 2 3 3 3
3 3 3 3 3 3
3 4 4 4 4 4
4 4 4 4 4
```

```
!234567890
      program integration
      implicit none
      include 'mpif.h'
      integer i,n
      integer myid,nproc,ierr,kount,iroot
      real*8 ff,xx,xsum,pi,hh,xpi
      character*8 fnnd ; character*10 fnnt
      integer itemp,itemq,irate
      ff(xx)= 4.0d0/(1.0d0+xx*xx)
      call MPI_init(ierr)
      call MPI_comm_size(MPI_COMM_WORLD,nproc,ierr)
      call MPI_comm_rank(MPI_COMM_WORLD,myid,ierr)
      if(myid == 0 .and. nproc > 1) print *,  nproc," processes are alive"
      if(myid == 0 .and. nproc ==1) print *,  nproc," process is alive"
      if(myid == 0)then   ! -----[   process id = 0
      call date_and_time(date=fnnd,time=fnnt)
      write(6,'(a10,2x,a8,2x,a10)') 'date,time ', fnnd,fnnt
                endif  ! -----]   process id = 0
      if( myid == 0) then
      write(6,*) 'number of intervals?'
      read(5,*) n
      write(6,*) 'number of intervals:',n
      end if
      iroot=0
      call MPI_bcast(n,1,MPI_INTEGER, iroot, MPI_COMM_WORLD, ierr)
      hh=1.d0/float(n)
      xsum=0.0d0
      do i=myid +1, n, nproc
         xx=(float(i)-0.5d0)*hh
         xsum=xsum+ff(xx)
      end do
      xpi=hh*xsum
      call MPI_reduce(xpi,pi, 1, MPI_DOUBLE_PRECISION, MPI_SUM, 0, MPI_COMM_WORLD,ierr)
      if( myid == 0 )then
      write(6,'(a,f18.8)')  'estimated pi value', pi
      end if
      call MPI_finalize(ierr)
      end program integration
```

```
      program main
      include "mpif.h"
      double precision  PI25DT
      parameter        (PI25DT = 3.14159265358979323846264338327950288419716939937510d0)
      double precision  mypi, pi, h, sum, x, f, a
      double precision starttime, endtime
      integer n, myid, numprocs, i, ierr
c                       function to integrate
      f(a) = 4.d0 / (1.d0 + a*a)

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

10    if ( myid .eq. 0 ) then
        print *, 'Enter the number of intervals: (0 quits) '
        read(*,*) n
      endif
c                       broadcast n
      starttime = MPI_WTIME()
      call MPI_BCAST(n,1,MPI_INTEGER,0,MPI_COMM_WORLD,ierr)
c                       check for quit signal
      if ( n .le. 0 ) goto 30
c                       calculate the interval size
      h = 1.0d0/n
      sum  = 0.0d0
      do 20 i = myid+1, n, numprocs
        x = h * (dble(i) - 0.5d0)
        sum = sum + f(x)
20    continue
      mypi = h * sum
c                       collect all the partial sums
      call MPI_REDUCE(mypi,pi,1,MPI_DOUBLE_PRECISION,MPI_SUM,0,
     &                MPI_COMM_WORLD,ierr)
c                       node 0 prints the answer.
      endtime = MPI_WTIME()
      if (myid .eq. 0) then
        print *, 'pi is ', pi, ' Error is', abs(pi - PI25DT)
        print *, 'time is ', endtime-starttime, ' seconds'
      endif
      goto 10
30    call MPI_FINALIZE(ierr)
      stop
      end
```

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

```
SUBROUTINE PAR_BLAS2(m, n, a, b, c, comm)
REAL a(m), b(m,n) ! local slice of array
REAL c(n) !result
REAL sum(n)
INTEGER n, comm, i, j, ierr

! local sum
DO j= 1, n
sum(j) = 0.0
DO i = 1, m
sum(j) = sum(j) + a(i)*b(i,j)
END DO
END DO

! global sum
CALL MPI_ALLREDUCE(sum, c, n, MPI_REAL, MPI_SUM, comm, ierr)

! return result at all nodes
RETURN
END
```

```
      dot_product: compute a scalar product

      subroutine dot_product(global,x,y,n)
      implicit none
      include "mpif.h"
      integer n,i,ierr
      double precision global,x(n),y(n)
      double precision tmp,local
      local = 0.0d0
      global = 0.0d0
     do i=1,n
     local = local + x(i)*y(i)
    enddo
    call MPI_ALLREDUCE(local,tmp,1,MPI_DOUBLE_PRECISION,
   >  MPI_SUM, MPI_COMM_WORLD, ierr)
    global = tmp
    return
    end
```

```
SUBROUTINE PAR_BLAS2(m, n, a, b, c, comm)
REAL a(m), b(m,n) ! local slice of array
REAL c(n) ! result
REAL sum(n)
INTEGER n, comm, i, j, ierr

! local sum
DO j= 1, n
sum(j) = 0.0
DO i = 1, m
sum(j) = sum(j) + a(i)*b(i,j)
END DO
END DO

! global sum
CALL MPI_REDUCE(sum, c, n, MPI_REAL, MPI_SUM,0, comm, ierr)

! return result at node zero (and garbage at the other nodes)
RETURN
END
```

```
!234567890
      program equal_load_sum
      implicit none
      include 'mpif.h'
      integer nn
      real*8, allocatable :: aa(:)
      integer nproc,myid,ierr,istart,ifinish
      integer i
      real*8 xsum,xxsum

      nn=10000

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

      call equal_load(1,nn,nproc,myid,istart,ifinish)

      allocate(aa(istart:ifinish)) ! 단순한 인덱스의 분할 뿐만아니라 메모리의 분할이 이루어지고 있다. 노드별로

      do i=istart,ifinish
      aa(i)=float(i)
      enddo

      xsum=0.0d0
      do i=istart,ifinish
      xsum=xsum+aa(i)
      enddo

     call MPI_REDUCE(xsum,xxsum,1,MPI_DOUBLE_PRECISION,MPI_SUM,0,MPI_COMM_WORLD,ierr)
     xsum=xxsum

     if(myid == 0)then
     write(6,*) xsum,' xsum'
     endif

    deallocate(aa)
     call MPI_FINALIZE(ierr)
     end program equal_load_sum
```

MPI-IO: The “Big Six”

While there are a large number of MPI-IO calls, most basic I/O can be handled with just six routines:

- MPI_File_open() – associate a file with a file handle.
- MPI_File_seek() – move the current file position to a given location in the file.
- MPI_File_read() – read some fixed amount of data out of the file beginning at the current file position.
- MPI_File_write() – write some fixed amount of data into the file beginning at the current file position.
- MPI_File_sync() – flush any caches associated with the file handle.
- MPI_File_close() – close the file handle.

Most of the other MPI-IO routines are variations or optimizations on these basic calls.

https://github.com/wkliao/mpi-io-examples

https://github.com/ljdursi/parallel-IO-tutorial/blob/master/presentation/presentation.md

MPI_File_open()

• C syntax:
int MPI_File_open(MPI_Comm comm, char *filename, int amode, MPI_Info info, MPI_File *fh);
• Fortran syntax:
subroutine MPI_File_open(icomm, filename, iamode, info, ifh, ierr)
character*(*) filename
integer icomm, iamode, info, ifh, ierr
• ifh is the file handle, which will be used by all other MPI-IO routines to refer to the file. In C, this must be passed as an address (i.e. &fh).
• filename is the name of the file to open. It may use a relative or absolute path. It may also contain a file system identifier prefix, e.g.:
– ufs: -- normal UNIX-style file system
– nfs: -- NFS file system
– pvfs: -- PVFS file system

MPI_File_open() (con't)

• amode is the access mode to open the file with. It should be a bitwise ORing (C) or sum (Fortran) of the following:
– MPI_MODE_RDONLY (read-only)
– MPI_MODE_RDWR (read/write)
– MPI_MODE_WRONLY (write-only)
– MPI_MODE_CREATE (create file if it doesn't exist)
– MPI_MODE_EXCL (error if file does already exist)
– MPI_MODE_DELETE_ON_CLOSE (delete file when closed)
– MPI_MODE_UNIQUE_OPEN (file cannot be opened by other processes)
– MPI_MODE_SEQUENTIAL (file can only be accessed sequentially)
– MPI_MODE_APPEND (set initial file position to the end of the file)
• info is a set of file hints, the creation of which will be discussed later.
When in doubt, use MPI_INFO_NULL.

MPI_File_open mode	**Description**
MPI_MODE_RDONLY	read only
MPI_MODE_WRONLY	write only
MPI_MODE_RDWR	read and write
MPI_MODE_CREATE	create file if it doesn't exist

MPI_File_seek()

• C syntax:
int MPI_File_seek(MPI_File fh, MPI_Offset offset, int whence);
• Fortran syntax:
subroutine MPI_File_seek(ifh, ioffset, iwhence, ierr)
integer ifh, ioffset, iwhence, ierr
• offset determines how far from the current file position to move the current file position; this can be negative, although seeking beyond the beginning of the file (or the current view if one is in use) is an error.
• whence determines to where the seek offset is relative:
– MPI_SEEK_SET (relative to the beginning of the file)
– MPI_SEEK_CUR (relative to the current file position)
– MPI_SEEK_END (relative to the end of the file)
• Seeks are done in terms of the current record type (MPI_BYTE by default, although this can be changed by setting a file view).

MPI_File_read()

• C syntax:

int MPI_File_read(MPI_File fh, void *buf, int count,MPI_Datatype type, MPI_Status *status);

• Fortran syntax:

subroutine MPI_File_read(ifh, buf, icount, itype,istatus, ierr)

<type> BUF(*)

integer ifh, icount, itype, istatus(MPI_STATUS_SIZE),ierr

• This reads count values of datatype type from the file into buf. buf must be at least as big as count*sizeof(type).

• istatus can be used to query for information such as how many bytes were actually read.

MPI_File_write()

• C syntax:
int MPI_File_write(MPI_File fh, void *buf, int count, MPI_Datatype type, MPI_Status *status);
• Fortran syntax:
subroutine MPI_File_write(ifh, buf, icount, itype,istatus, ierr)
<type> BUF(*)
integer ifh, icount, itype, istatus(MPI_STATUS_SIZE), ierr
• This reads count values of datatype type from buf into the file. buf must be at least as big as count*sizeof(type).
• istatus can be used to query for information such as how many bytes were actually written.

MPI_File_sync()

• C syntax:

```
int MPI_File_sync(MPI_File fh);
```

• Fortran syntax:

```
subroutine MPI_File_sync(ifh, ierr)
integer ifh, ierr
```

• Forces any caches associated with a file handle to be flushed to disk; maybe be very expensive for large files on slow file systems.

• Provides a way to force written data to be committed to disk.

• Collective; must be called by all processes.

MPI_File_close()

• C syntax:

```
int MPI_File_close(MPI_File *fh);
```

• Fortran syntax:

```
subroutine MPI_File_close(ifh, ierr)
integer ifh, ierr
```

• This closes access to the file associated with the file handle ifh.

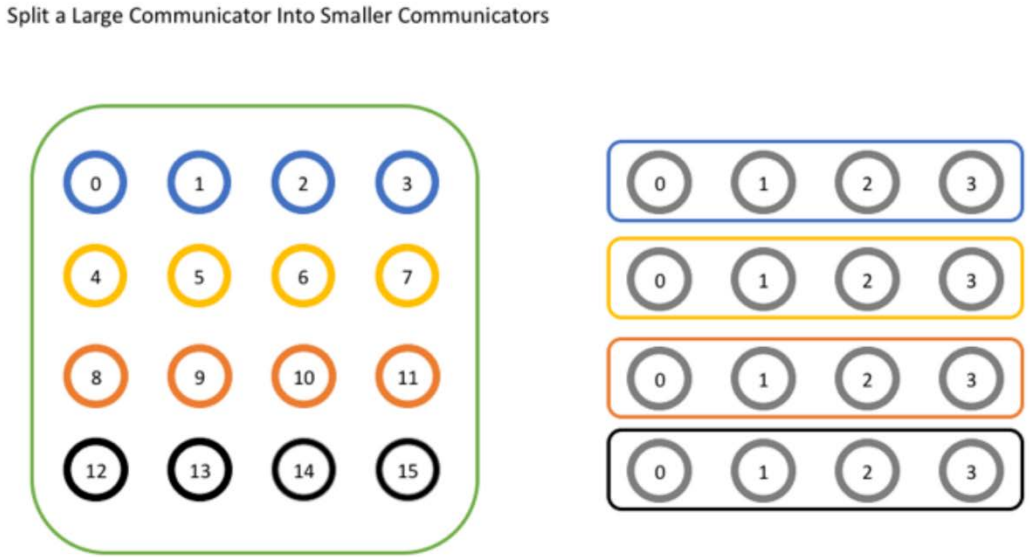
MPI I/O example

```
! Basic MPI-I/O
call MPI_BARRIER(MPI_COMM_WORLD,ierr)
tstart=MPI_WTIME()
call MPI_FILE_OPEN(MPI_COMM_WORLD,'u.dat',MPI_MODE_WRONLY, MPI_INFO_NULL,outfile,ierr)
if (jstart.eq.2)  jstart=1
if (jend.eq.(jmax-1))   jend=jmax
call MPI_FILE_SEEK(outfile,(jstart-1)*imax*8,MPI_SEEK_SET,ierr)
do j=jstart,jend
call MPI_FILE_WRITE(outfile,u(1,j),imax,MPI_REAL8,istatus, ierr)
enddo
call MPI_FILE_SYNC(outfile,ierr)
call MPI_FILE_CLOSE(outfile,ierr)
call MPI_BARRIER(MPI_COMM_WORLD,ierr)
tend=MPI_WTIME()
if (rank.eq.0) then
write(*,*) 'Using basic MPI-I/O‘
write(*,*) 'Wrote ',8*imax*jmax,' bytes in ',tend-tstart, ' seconds.'
write(*,*) 'Transfer rate = ', (8*imax*jmax)/(tend-tstart)/(1024.**2), ' MB/s'
                    endif
```

write into a single file, in Fortran

```
! example of parallel MPI write into a single file, in Fortran
        PROGRAM main
        include 'mpif.h'

        integer ierr, i, myrank, BUFSIZE, thefile
        parameter (BUFSIZE=100)
        integer buf(BUFSIZE)
        integer(kind=MPI_OFFSET_KIND) disp

        call MPI_INIT(ierr)
        call MPI_COMM_RANK(MPI_COMM_WORLD, myrank, ierr)

       do i = 0, BUFSIZE
        buf(i) = myrank * BUFSIZE + i
      enddo
    call MPI_FILE_OPEN(MPI_COMM_WORLD, 'testfile', MPI_MODE_WRONLY + MPI_MODE_CREATE,  MPI_INFO_NULL, thefile, ierr)
  ! assume 4-byte integers
     disp = myrank * BUFSIZE * 4
     call MPI_FILE_SET_VIEW(thefile, disp, MPI_INTEGER, MPI_INTEGER, 'native', MPI_INFO_NULL, ierr)
     call MPI_FILE_WRITE(thefile, buf, BUFSIZE, MPI_INTEGER,  MPI_STATUS_IGNORE, ierr)
     call MPI_FILE_CLOSE(thefile, ierr)
     call MPI_FINALIZE(ierr)

     END PROGRAM main
```

Split a large communicator



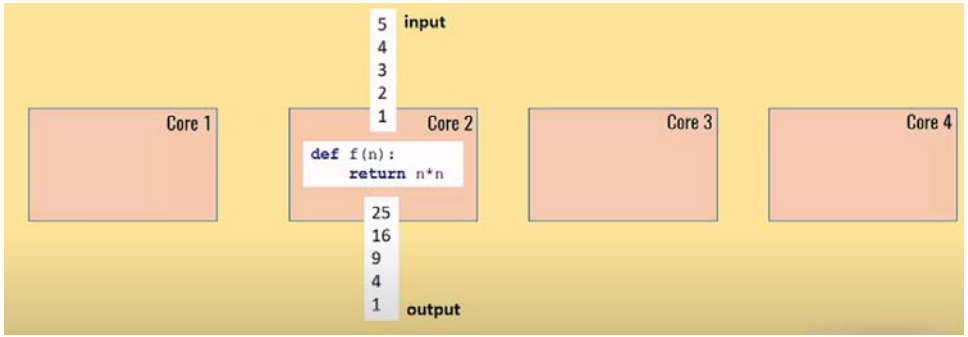
Split a large communicator

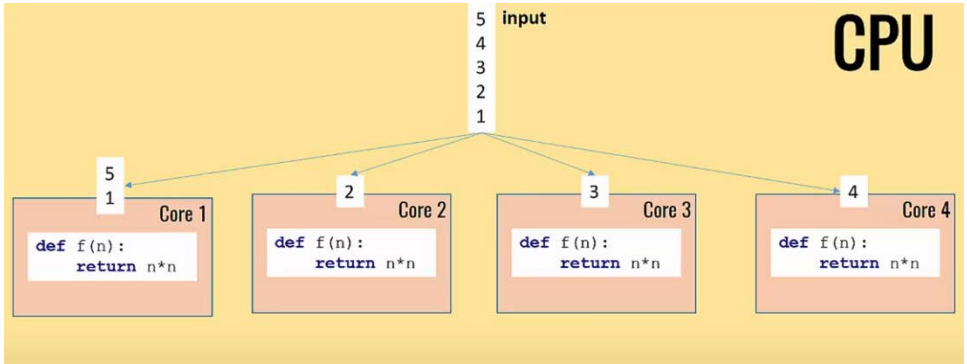
```
                                                    // Get the rank and size in the original communicator
int world_rank, world_size;
MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);
MPI_Comm_size(MPI_COMM_WORLD, &world_size);

int color = world_rank / 4;                         // Determine color based on row

                                                    // Split the communicator based on the color and use the
// original rank for ordering
MPI_Comm row_comm;
MPI_Comm_split(MPI_COMM_WORLD, color, world_rank, &row_comm);

int row_rank, row_size;
MPI_Comm_rank(row_comm, &row_rank);
MPI_Comm_size(row_comm, &row_size);

printf("WORLD RANK/SIZE: %d/%d \t ROW RANK/SIZE: %d/%d\n",world_rank, world_size, row_rank, row_size);

MPI_Comm_free(&row_comm);
```

Split a large communicator

```
                                                    // Get the rank and size in the original communicator
int world_rank, world_size;
MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);
MPI_Comm_size(MPI_COMM_WORLD, &world_size);
                                                    // Get the group of processes in MPI_COMM_WORLD
MPI_Group world_group;
MPI_Comm_group(MPI_COMM_WORLD, &world_group);

int n = 7;
const int ranks[7] = {1, 2, 3, 5, 7, 11, 13};
                                                  // Construct a group containing all of the prime ranks in world_group
MPI_Group prime_group;
MPI_Group_incl(world_group, 7, ranks, &prime_group);
                                                    // Create a new communicator based on the group
MPI_Comm prime_comm;
MPI_Comm_create_group(MPI_COMM_WORLD, prime_group, 0, &prime_comm);
int prime_rank = -1, prime_size = -1;
                                                    // If this rank isn't in the new communicator, it will be
                                                 // MPI_COMM_NULL. Using MPI_COMM_NULL for MPI_Comm_rank or
                                                 // MPI_Comm_size is erroneous
if (MPI_COMM_NULL != prime_comm) {
   MPI_Comm_rank(prime_comm, &prime_rank);
   MPI_Comm_size(prime_comm, &prime_size);
}
printf("WORLD RANK/SIZE: %d/%d \t PRIME RANK/SIZE: %d/%d\n",world_rank, world_size, prime_rank, prime_size);
MPI_Group_free(&world_group);
MPI_Group_free(&prime_group);
MPI_Comm_free(&prime_comm);
```

Multiprocessing Pool (1/2)






```
from multiprocessing import Pool

def f(n):
    return n*n

if __name__ == "__main__":
    array = [1,2,3,4,5]
    p = Pool()
    result = p.map(f, array)
    print(result)
```

Multiprocessing Pool (2/2)

```
from multiprocessing import Pool
import time

def f(n):
    sum = 0
    for x in range(1000):
        sum += x*x
    return sum

if __name__ == "__main__":

    t1=time.time()
    p = Pool()
    result = p.map(f,range(100000))
    p.close()
    p.join()

    print("Pool took:",time.time()-t1)

    t2 = time.time()
    result = []
    for x in range(100000):
        result.append(f(x))

    print("Serial processing took: ",time.time()-t2)
```

mpi4py

MPI4Py provides an interface very similar to the MPI-2 standard C++ Interface

Focus is in translating MPI syntax and semantics: If you know MPI, MPI4Py is "obvious"

You can communicate Python objects!!

What you lose in performance, you gain in shorter development time

mpi4py

There are hundreds of functions in the MPI standard, not all of them are necessarily available in MPI4Py, most commonly used are

No need to call MPI_Init() or MPI_Finalize()

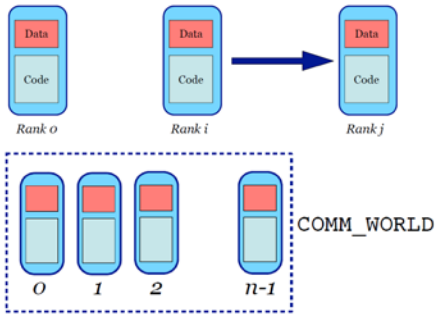
–MPI_Init() is called when you import the module

–MPI_Finalize() is called before the Python process ends

- petsc4py
- mpi4py
- NumPy

Python with MPI

- All processors execute the same program
- Every process has a unique rank
- Branch on the rank
- Point-to-point, collective communications
- Blocking, nonblocking communications



mpi4py-1.3.1

3.0

```
[ihlee@tucana-master mpi4py_tuto]$ ls /usr/local/mpi4py-1.3.1
total 80
4 build/   4 demo/   8 HISTORY.txt   8 mpi.cfg    4 README.txt  20 setup.py   4 test/
4 conf/    4 docs/   4 LICENSE.txt   4 PKG-INFO   4 setup.cfg   4 src/        4 THANKS.txt
[ihlee@tucana-master mpi4py_tuto]$ ls /usr/local/mpi4py-1.3.1/demo/
total 116
4 compute-pi/     4 helloworld.f90  4 mpe-logging/     4 osu_bw.py          4 sequential/   4 wrap-c/
4 cython/         4 helloworld.py   4 mpi-ref-v1/      4 osu_latency.py     4 spawning/     4 wrap-cython/
4 embedding/      4 init-fini/      4 nxtval/          4 osu_multi_lat.py   4 threads/      4 wrap-f2py/
4 helloworld.c    4 makefile        4 osu_alltoall.py  4 README.txt         4 vampirtrace/  4 wrap-swig/
4 helloworld.cxx  4 mandelbrot/     4 osu_bibw.py      4 reductions/        4 wrap-boost/
[ihlee@tucana-master mpi4py_tuto]$
```

Hello World

```
[ihlee@tucana-master mpi4py_tuto]$ cat /usr/local/mpi4py-1.3.1/demo/helloworld.py
#!/usr/bin/env python
"""
Parallel Hello World
"""

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))
[ihlee@tucana-master mpi4py_tuto]$
```

test

```
[ihlee@tucana-master mpi4py_tuto]$ cp /usr/local/mpi4py-1.3.1/demo/helloworld.py .
[ihlee@tucana-master mpi4py_tuto]$ mpirun -np 5 python helloworld.py
Hello, World! I am process 2 of 5 on master.hpc.
Hello, World! I am process 3 of 5 on master.hpc.
Hello, World! I am process 4 of 5 on master.hpc.
Hello, World! I am process 0 of 5 on master.hpc.
Hello, World! I am process 1 of 5 on master.hpc.
[ihlee@tucana-master mpi4py_tuto]$
```

Issuing at the command line:

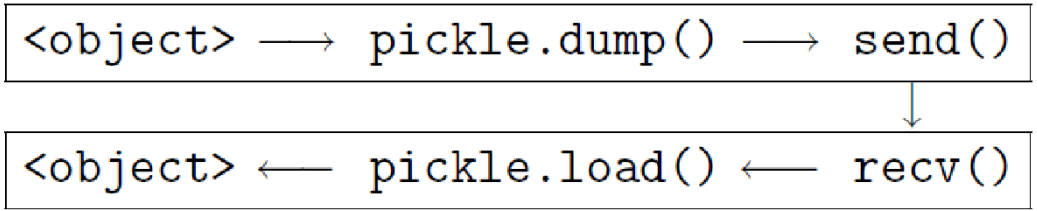
```
$ mpiexec -n 5 python demo/helloworld.py
```

or (in the case of older MPI-1 implementations):

```
$ mpirun -np 5 python demo/helloworld.py
```

mpiexec *vs* mpirun

- Communication of Python objects.
  - high level and very convenient, based in `pickle` serialization
  - can be slow for large data (CPU and memory consuming)



- Communication of array data (e.g. **NumPy** arrays).
  - lower level, slightly more verbose
  - very fast, almost C speed (for messages above 5-10 KB)

```
message = [<object>, (count, displ), datatype]
```

- Broadcasting a Python dictionary:

```
from mpi4py import MPI

comm = MPI.COMM_WORLD
rank = comm.Get_rank()

if rank == 0:
    data = {'key1' : [7, 2.72, 2+3j],
            'key2' : ( 'abc', 'xyz')}
else:
    data = None
data = comm.bcast(data, root=0)
```

- Scattering Python objects:

```
from mpi4py import MPI

comm = MPI.COMM_WORLD
size = comm.Get_size()
rank = comm.Get_rank()

if rank == 0:
   data = [(i+1)**2 for i in range(size)]
else:
   data = None
data = comm.scatter(data, root=0)
assert data == (rank+1)**2
```

- Gathering Python objects:

```
from mpi4py import MPI

comm = MPI.COMM_WORLD
size = comm.Get_size()
rank = comm.Get_rank()

data = (rank+1)**2
data = comm.gather(data, root=0)
if rank == 0:
    for i in range(size):
        assert data[i] == (i+1)**2
else:
    assert data is None
```

Importing library:

```
from mpi4py import MPI
```

Getting important information:

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

Executing in parallel:

```
mpirun -np <P> python <code>.py
```

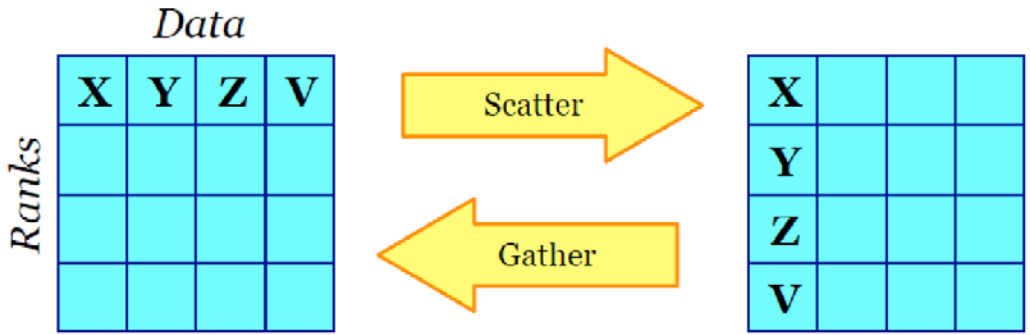
```
import time
t = time.time()
time.sleep(10)
t2 = time.time()

spendtime = t2 - t
print("Before timestemp: ", t)
print("After timestemp: ", t2)
print("Wait {0} seconds".format(spendtime))
```

```
>>>
('Before timestemp: ', 1321780317.218)
('After timestemp: ', 1321780327.218)
Wait 10.0 seconds
```

```
[ihlee@tucana-master ~]$ python
Python 2.6.6 (r266:84292, Jul 10 2013, 22:48:45)
[GCC 4.4.7 20120313 (Red Hat 4.4.7-3)] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> import time
>>> t1=time.time()
>>> t2=time.time()
>>> print t2-t1
7.2504940033
>>>
```

Scatter/Gather

```
./s_g.py
After Scatter:
[0] [ 0.  1.  2.  3.]
After Allgather:
[0] [ 0.  2.  4.  6.]
[ihlee@tucana-master mpi4py_tuto]$ vi s_g.py
```



```
#!/usr/bin/env python
#--------------
# Loaded Modules
#--------------
import numpy as np
from mpi4py import MPI
#-------------
# Communicator
#-------------
comm = MPI.COMM_WORLD

my_N = 4
N = my_N * comm.size

if comm.rank == 0:
   A = np.arange(N, dtype=np.float64)
else:
   #Note that if I am not the root processor A is an empty array
   A = np.empty(N, dtype=np.float64)
my_A = np.empty(my_N, dtype=np.float64)
#-------------------------
# Scatter data into my_A arrays
#-------------------------
comm.Scatter( [A, MPI.DOUBLE], [my_A, MPI.DOUBLE] )

if comm.rank == 0:
  print "After Scatter:"

for r in xrange(comm.size):
   if comm.rank == r:
      print "[%d] %s" % (comm.rank, my_A)
   comm.Barrier()
```

```
#-----------------------
# Everybody is multiplying  by 2
#-----------------------
my_A *= 2

#----------------------
# Allgather  data into A again
#----------------------
comm.Allgather(  [my_A, MPI.DOUBLE], [A, MPI.DOUBLE] )

if comm.rank == 0:
   print "After Allgather:"

for r in xrange(comm.size):
    if comm.rank == r:
        print "[%d] %s" % (comm.rank, A)
    comm.Barrier()
```

```
from mpi4py import MPI
import numpy as np

COMM = MPI.COMM_WORLD
RANK = COMM.Get_rank()
SIZE = COMM.Get_size()
N = 10

if(RANK == 0):
    DATA = np.arange(N*SIZE, dtype='i')
    for i in range(1, SIZE):
        SLICE = DATA[i*N:(i+1)*N]
        COMM.Send([SLICE, MPI.INT], dest=i)
    MYDATA = DATA[0:N]
else:
    MYDATA = np.empty(N, dtype='i')
    COMM.Recv([MYDATA, MPI.INT], source=0)
```

```
S = sum(MYDATA)
print RANK, 'has data', MYDATA, 'sum =', S

SUMS = np.zeros(SIZE, dtype='i')
if(RANK > 0):
    COMM.send(S, dest=0)
else:
    SUMS[0] = S
    for i in range(1, SIZE):
        SUMS[i] = COMM.recv(source=i)
    print 'total sum =', sum(SUMS)
```

```
from mpi4py import MPI
comm = MPI.COMM_WORLD

if comm.rank == 0:
    sendmsg = (7, "abc", [1.0,2+3j], {3:4})
else:
    sendmsg = None

recvmsg = comm.bcast(sendmsg, root=0)
```

```
from mpi4py import MPI
comm = MPI.COMM_WORLD

if comm.rank == 0:
    sendmsg = [i**2 for i in range(comm.size)]
else:
    sendmsg = None

recvmsg = comm.scatter(sendmsg, root=0)
```

```
from mpi4py import MPI
comm = MPI.COMM_WORLD

sendmsg = comm.rank**2

recvmsg1 = comm.gather(sendmsg, root=0)

recvmsg2 = comm.allgather(sendmsg)
```

```
from mpi4py import MPI
comm = MPI.COMM_WORLD

sendmsg = comm.rank

recvmsg1 = comm.reduce(sendmsg, op=MPI.SUM, root=0)

recvmsg2 = comm.allreduce(sendmsg)
```

MPI

- Many examples
- High-level usage of MPI
- Practice

Serial version

```
import math

def compute_pi(n):
    h = 1.0 / n
    s = 0.0
    for i in range(n):
        x = h * (i + 0.5)
        s += 4.0 / (1.0 + x**2)
    return s * h

n = 10
pi = compute_pi(n)
error = abs(pi - math.pi)

print ("pi is approximately %.16f, "
       "error is %.16f" % (pi, error))
```

Parallel version

```
from mpi4py import MPI
import math

def compute_pi(n, start=0, step=1):
    h = 1.0 / n
    s = 0.0
    for i in range(start, n, step):
        x = h * (i + 0.5)
        s += 4.0 / (1.0 + x**2)
    return s * h

comm = MPI.COMM_WORLD
nprocs = comm.Get_size()
myrank = comm.Get_rank()
```

```
if myrank == 0:
    n = 10
else:
    n = None

n = comm.bcast(n, root=0)

mypi = compute_pi(n, myrank, nprocs)

pi = comm.reduce(mypi, op=MPI.SUM, root=0)

if myrank == 0:
    error = abs(pi - math.pi)
    print ("pi is approximately %.16f, "
           "error is %.16f" % (pi, error))
```

Serial version

```
def mandelbrot(x, y, maxit):
    c = x + y*1j
    z = 0 + 0j
    it = 0
    while abs(z) < 2 and it < maxit:
        z = z**2 + c
        it += 1
    return it

x1, x2 = -2.0, 1.0
y1, y2 = -1.0, 1.0
w,  h  = 150, 100
maxit  = 127
```

```
import numpy
C = numpy.zeros([h, w], dtype='i')
dx = (x2 - x1) / w
dy = (y2 - y1) / h
for i in range(h):
    y = y1 + i * dy
    for j in range(w):
        x = x1 + j * dx
        C[i, j] = mandelbrot(x, y, maxit)

from matplotlib import pyplot
pyplot.imshow(C, aspect='equal')
pyplot.spectral()
pyplot.show()
```

Parallel version (1/3)

```
def mandelbrot(x, y, maxit):
    c = x + y*1j
    z = 0 + 0j
    it = 0
    while abs(z) < 2 and it < maxit:
        z = z**2 + c
        it += 1
    return it

x1, x2 = -2.0, 1.0
y1, y2 = -1.0, 1.0
w,  h  = 150, 100
maxit  = 127
```

Parallel version (2/3)

```
def mandelbrot(x, y, maxit):
    c = x + y*1j
    z = 0 + 0j
    it = 0
    while abs(z) < 2 and it < maxit:
        z = z**2 + c
        it += 1
    return it

x1, x2 = -2.0, 1.0
y1, y2 = -1.0, 1.0
w,  h  = 150, 100
maxit  = 127
```

```
from mpi4py import MPI
import numpy

comm = MPI.COMM_WORLD
size = comm.Get_size()
rank = comm.Get_rank()

# number of rows to compute here
N = h // size + (h % size > rank)

# first row to compute here
start = comm.scan(N)-N

# array to store local result
C1 = numpy.zeros([N, w], dtype='i')
```

Parallel version (3/3)

```
dx = (x2 - x1) / w
dy = (y2 - y1) / h
for i in range(N):
    y = y1 + (i + start) * dy
    for j in range(w):
        x = x1 + j * dx
        C1[i, j] = mandelbrot(x, y, maxit)
```

```
counts = comm.gather(N, root=0)
C = None
if rank == 0:
    C = numpy.zeros([h, w], dtype='i')

rowtype = MPI.INT.Create_contiguous(w)
rowtype.Commit()

comm.Gatherv(sendbuf=[C1, MPI.INT],
             recvbuf=[C, (counts, None), rowtype],
             root=0)

rowtype.Free()
```

```
if comm.rank == 0:
    from matplotlib import pyplot
    pyplot.imshow(C, aspect='equal')
    pyplot.spectral()
    pyplot.show()
```

Parallel

Most MPI programs can be run with the command **mpiexec**.

```
$ mpiexec -n 4 python script.py
```

```
#!/usr/bin/env python
"""
Parallel Hello World
"""
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))
```

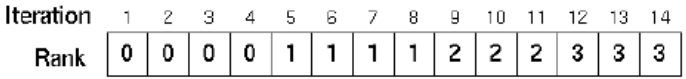
Parallel

```
from mpi4py import MPI

comm = MPI.COMM_WORLD
rank = comm.Get_rank()
if rank == 0:
    fd = open("dna_file.txt", "r")
    dna = fd.read()
else:
    dna = None

dna = comm.bcast(dna, root=0)
print rank, dna
```

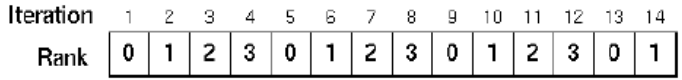
```
from mpi4py import MPI
comm = MPI.COMM_WORLD
rank = comm.Get_rank()

max = comm.reduce(rank, op=MPI.MAX, root =0)

if rank == 0:
    print "The reduction is %s" % max
```

Parallelizing Loops

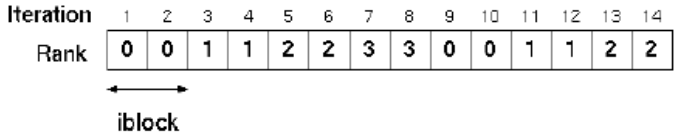
Block Distribution



Cyclic Distribution



Block-Cyclic Distribution



Y. Aoyama and J. Nakano, *RS/6000 SP: Practical MPI Programming*

http://incredible.egloos.com/3755171

Parameter range

```
SUBROUTINE para_range(n1, n2, nprocs, irank, ista, iend)
iwork1 = (n2 - n1 + 1) / nprocs
iwork2 = MOD(n2 - n1 + 1, nprocs)
ista = irank * iwork1 + n1 + MIN(irank, iwork2)
iend = ista + iwork1 - 1
IF (iwork2 > irank) iend = iend + 1
END

SUBROUTINE para_range(n1, n2, nprocs, irank, ista, iend)
iwork = (n2 - n1) / nprocs + 1
ista = MIN(irank * iwork + n1, n2 + 1)
iend = MIN(ista + iwork - 1, n2)
END
```

Y. Aoyama and J. Nakano, *RS/6000 SP: Practical MPI Programming*
http://incredible.egloos.com/3755171

References

- John W. Shipman, *A Python 2.7 programming tutorial*
- D. Beazley and B. K. Jones*, Python Cookbook 3rd ed.*
- Tutorialpoint, *Python 3*
- Richard L. Halterman, *Fundamentals of Python Programming*
- Brian Heinold, *A Practical Introduction to Python Programming*
- K. W. Smith*, Cython*
- Stephen Weston, *Parallel Computing in Python using mpi4py*
- Lisandro Dalcin, *MPI for Python*
- Konrad Hinsen, *Optimizing and Interfacing with Cython*
- Stéfan van der Walt, *The Quest for Speed: An Introduction to Cython*
- *http://cs231n.github.io/python-numpy-tutorial/*
- *https://code.tutsplus.com/tutorials/speeding-python-with-cython--cms-29557*
- *https://docs.scipy.org/doc/numpy/user/quickstart.html*
- *http://book.pythontips.com/en/latest/python_c_extension.html*
- *https://notmatthancock.github.io/2017/02/10/calling-fortran-from-python.html*
- Y. Aoyama and J. Nakano, *RS/6000 SP: Practical MPI Programming*
- *https://en.wikibooks.org/wiki/A_Beginner%27s_Python_Tutorial/Classes*

MPI Implementations

- Most parallel machine vendors have optimized versions
- Others:
  - http://www.mpi.nd.edu/MPI/Mpich
  - http://www-unix.mcs.anl.gov/mpi/mpich/
  - indexold.html
  - GLOBUS:
  - http://www.globus.org/mpi/
  - http://exodus.physics.ucla.edu/appleseed/

추가 참고 문헌

"parallel computing" on the search engine

https://computing.llnl.gov/tutorials/mpi/ http://condor.cc.ku.edu/~grobe/docs/intro-MPI.shtml

http://www-unix.mcs.anl.gov/mpi/

http://www.llnl.gov/computing/tutorials/workshops/workshop/parallel_comp/MAIN.html#Whatis

http://www.nas.nasa.gov/Groups/SciCon/Tutorials/MPIintro/toc.html

http://www-unix.mcs.anl.gov/mpi/tutorial/mpibasics/

http://www-unix.mcs.anl.gov/mpi/usingmpi/examples/main.htm

http://people.sc.fsu.edu/~jburkardt/f_src/f_src.html http://condor.cc.ku.edu/~grobe/docs/intro-MPI-C.shtml

http://people.sc.fsu.edu/~jburkardt/index.html

http://www.mcs.anl.gov/research/projects/mpi/tutorial/

http://www.bu.edu/tech/support/research/training-consulting/online-tutorials/mpi/

http://condor.cc.ku.edu/~grobe/docs/intro-MPI.shtml

Summary

- 새로운 패러다임 (병렬계산,성능/가격)
- 통신의 중요성 (latency + bandwidth)
- MPI 프로그램의 기본 이해 (SPMD)
- 실제문제 재설계 필요
- *점진적 병렬화 작업* (serial → parallel)
- 서로 다른 방식으로의 병렬화 모색
- 실질적 시간 절약 효과를 검증 해야 함