INTRODUCTION TO PARALLELIZATION
WITH
BASIC MPI

A SHORT COURSE
BY
SPIRO VELLAS

TEXAS A&M SUPERCOMPUTING FACILITY

(Revised October 2010)
AGENDA AND TOPICS COVERED

DAY 1
-- MPI Reference Material
-- Message Passing Concepts
-- Typical Layout of MPI Programs
-- Compiling and Running MPI Programs
-- Scheduling: Block, Cyclic, Block-cyclic, etc
-- Examples

Day 2
-- Predefined MPI Data Types
-- Standard Point-to-Point Process Communication
(blocking and non-blocking send, receive, and some accessory routines)
-- Deadlocks in Communication
-- Overview of Collective Communication
-- Message Broadcasting (MPI_Bcast)
-- Example: Matrix-Vector Multiplication (version 1)

Day 3
-- Review key features of the Matrix-Vector Multiplication code
-- Matrix-Matrix Multiplication Example Code
-- Other Collective Routines: Gather(v), Allgather, Scatter(v), Reduce, Allreduce, MPI_OP_CREATE
-- Examples

Day 4
-- Solving the 2-dim Poisson Problem
-- Serial code for solving 2-dim Poisson Problem
-- Parallel (column-wise block distribution) code for 2-dim Poisson Problem
-- Parallel (column- and row-wise block distribution) code for 2-dim Poisson Problem
( this one is still under preparation)
INTRODUCTION

The MPI material covered in these "notes" is delivered as a "Short Course" in three or four two-hour lecture sessions at Texas A&M University by the staff of its supercomputing facility. The intended scope of the course is basic and, to some extent, intermediate MPI. Free use is made of the multitude of MPI teaching material available on the Web. In particular, I made considerable direct use of references [3] and [5].

The format here is informal and the coverage is in no way comprehensive. To a good extent it is example-driven. Many obvious omissions and relevant elaborations are covered during the lectures.

Spiros Vellas
Analyst
CIS, Supercomputing Facility
February 2005
1) http://www-unix.mcs.anl.gov/mpi/ (Contains links to, among others, the "MPI standard", and "Materials for learning MPI". The latter has links to all the examples cited in ref. [3])

2) http://webct.ncsa.uiuc.edu:8900/public/MPI/ (This is a very good on-line MPI tutorial course)

3) USING MPI. Portable Parallel Programming with the Message Passing Interface. W. Gropp, E. Lusk, A. Skjellum. MIT Press, 2nd edition. (The authors of this well written book have played very active roles in the design and development of MPI)

4) PARALLEL PROGRAMMING with MPI. P. S. Pacheco. Morgan Kaufman Publishers, Inc.(Good reference for the MPI's C binding)


6) http://www.csit.fsu.edu/~burkardt/mpi_workshop.html A Florida State website that contains a wealth of MPI materials, including PDF and Power-Point presentations of a four-day workshop
MPI - MESSAGE PASSING INTERFACE

- MPI is a library of routines, with bindings in Fortran, C, and C++, that implement the message passing model of parallel computation

- Message-passing computation is predicated on a set of processes that coordinate computation by explicitly sending and receiving messages

- All processes have local memories only. That is, each process has a separate address space, its own program counter (PC), and its own call stack

- Data transfers from the local memory of one process to that of another occur only by permission and explicit operations performed by the involved parties

- The common MPI implementation (MPI 1.2) creates a fixed set of processes at program initialization, one mpi process per cpu. In MPI 2.0, the number of processes can change dynamically. But as of now (circa Feb 2005) not all vendors have implemented dynamic process addition or deletion
MPI - MESSAGE PASSING INTERFACE

MPI implements point-to-point communication between any two processes, whereby one process explicitly sends and the another explicitly receives. It also implements collective communication operations whereby a group of processes perform global/collective operations, such as, gather and scatter.

- MPI implements the concept of "communicator" to identify the process group with respect to which an operation is to be performed. This also enables the creation of sub-groups of processes within a group. The parameter constant MPI_COMM_WORLD specifies the total universe of processes within an MPI run session.

- An MPI session assigns a unique identity, called rank, to a process within a group of processes. The assigned ranks within a process group are consecutive integers: 0, 1, 2, ...

- MPI processes can exchange data through messages. A message is characterized by its "envelope" and its message body:
MPI's DISTRIBUTED MEMORY MODEL

- Most commonly, but not necessarily, one mpi process executes on one cpu (an implementation issue!!)

- In reality one or more CPU's may share the same physical local memory of the "node" they belong in, but an mpi process has unrestricted access only within its own address space

-The MPI distributed memory model is implemented on both cluster systems with truly physically distributed memories, such as IBM's p5-575+ clusters, and on shared memory machines like the SGI Altix
TYPICAL LAYOUT OF AN MPI PROGRAM

IN FORTRAN

program sample1
include "mpif.h" !USE MPI in F90
integer numprocs, rank, ierr
:            ! No MPI calls here
:           :
call MPI_INIT(ierr)
call MPI_COMM_RANK(MPI_COMM_WORLD, rank, ierr)
call MPI_COMM_SIZE(MPI_COMM_WORLD, numprocs, ierr)

    ! process
! rank = 0 1 ... Numprocs-1

call MPI_FINALIZE(ierr) ! All processes MUST
: ! execute this
: ! No MPI calls here
stop
end

IMPORTANT
1. All pending communications involving a process must complete before the process calls MPI_FINALIZE. Similarly, all files opened by a process must be closed before the process calls MPI_FINALIZE. When either of the above conditions is violated error results.

2. In the Fortran binding, all MPI routines, except the functions MPI_Wtime() and MPI_tick(), must specify as the last argument an (integer) error status variable—even though in these notes you may find it missing here and there. A status value equal to MPI_SUCCESS indicates successful completion of an MPI call, otherwise the occurrence of error.

3. The MPI standard does not specify the the handling of common-block or global variables. Almost all vendors, however, replicate them for each MPI process...
TYPICAL LAYOUT of an MPI PROGRAM

In C
#include "mpi.h"

main( int argc, char** argv) {
 :
 : /* No MPI calls here */
 int numprocs, myid;
 MPI_Init(&argc, &argv);
 MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
 MPI_Comm_rank(MPI_COMM_WORLD,&myid);

 /* rank */ 0 1 ... Numprocs -1*/

MPI_Finalize(); /* All processes MUST execute this */
 : /* No MPI calls after this */
 : }
 /* main */

Comments:

The return value (mostly integer) of an MPI call captures the error code. A returned value of MPI_SUCCESS from an MPI call signifies successful completion of the call, otherwise an error has occurred.
COMPILING AND RUNNING MPI PROGRAMS
ON THE IBM p5-575/AIX CLUSTER

To compile and run interactively standard mpi codes on HYDRA you need to do two things:

1) Your .rhosts file in your home directory and the host.list file in the directory where you plan to execute your MPI program must exist and have the entries shown.

2) The second file, host.list, must have multiple line entries, at most 4 f1n1-s and at most 4 f1n10-s. This reflects the restriction of using a maximum of 4 cpus on hydra (f1n1) and 4 cpus on hydra2 (f1n10)

---

**.rhosts file**

In $HOME directory

<table>
<thead>
<tr>
<th>hydra.tamu.edu username</th>
</tr>
</thead>
<tbody>
<tr>
<td>hydra2.tamu.edu username</td>
</tr>
<tr>
<td>:</td>
</tr>
<tr>
<td>:</td>
</tr>
</tbody>
</table>

**host.list file**

in execution directory

<table>
<thead>
<tr>
<th>f1n1-s</th>
</tr>
</thead>
<tbody>
<tr>
<td>f1n1-s</td>
</tr>
<tr>
<td>f1n1-s</td>
</tr>
<tr>
<td>f1n10-s</td>
</tr>
<tr>
<td>f1n10-s</td>
</tr>
<tr>
<td>f1n10-s</td>
</tr>
</tbody>
</table>
2) Compile and link your MPI code with the following commands:

**Fortran**
```
mpxlf90_r -o prog.exe [options] prog.f90 ...
```

**C**
```
mpcc_r    -o prog.exe [options] prog.c ...
```

**C++**
```
mpCC_r    -o prog.exe [options] prog.C ...
```

3) Execute your MPI program under the Parallel Operating Environment (POE)
```
prog.exe -procs nn -shared_memory yes -resd no -euilib us \ 
 -single_thread yes
```
COMPILING AND RUNNING MPI PROGRAMS
ON THE IBM p5-575/AIX CLUSTER

-procs nn  Specifies that the number of processors to run your program in parallel be set to nn.

-shared_memory yes  Specifies that MPI will use shared memory protocol (NOT IP) for message passing between two or more tasks within the same IBM Regatta. Make sure that you set this option to "yes", because the default ("no") results in much lower performance.

-infolevel n  Specifies the level of message reporting. The default is 1 (warning and error). Higher levels (2,3,...,6) provide progressively more diagnostic information.

-wait_mode poll  Directs that an MPI thread engage in polling, when blocked waiting for a message to arrive, in order to detect such arrivals. (Other nonoptimal values are yield, sleep, and nopoll)

-resd no  Specifies that the Partition Manager (PM) should NOT connect to LoadLeveler (LL) when running a MPI program.
Most of poe's command-line options, including the above, can also be set with environment variables:

```
export MP_PROCS=n         (Borne or Korn Shell)
export MP_SHARED_MEMORY=yes
export MP_INFOLEVEL=n (0,1,..,6)
export MP_WAIT_MODE=poll
export MP_RESD=no

setenv MP_PROCS n         (C-Shell)
setenv MP_SHARED_MEMORY yes
setenv MP_INFOLEVEL n
setenv MP_WAIT_MODE poll
setenv MP_RESD no
```
Several MPI Implementations Available
We will use & illustrate

Open MPI (Default)
INTEL MPI

There are several implementations of the MPI standard. Open MPI is the one we recommend. If your MPI program/application requires an MPI implementation different from Open MPI, contact the help desk.
COMPILING–LINKING AND RUNNING MPI PROGRAMS
On the Eos (IBM iDataPlex) Linux Cluster

MPI Implementation: **Open MPI**
(Interactive Case)

To compile and run interactively standard mpi codes on Eos you need to insure that:

- In the current/execution directory the `login.hosts` file exists and has up to five entries (see below) corresponding to the five available login nodes.
- You must not mix different MPI implementations, e.g., INTEL MPI & Open MPI.
- You must not use more than 4 cpus/cores per (login) node.
- Only for small & short (less than 15 minutes) duration jobs.
- The terms **MPI process** and **MPI task** are used interchangeably here.
- Must load intel compilers and Open MPI environment

```bash
module load intel/compilers
module load openmpi
```

`./login.hosts` file needed only when using multiple nodes

It specifies which nodes are available for parallel mpi processing during interactive sessions/logins. This is the simplest form for a hosts file.

| login001  |
| login002  |
| login005  |

There are several implementations of the MPI standard. Open MPI is the one we recommend. If your MPI program/application requires an MPI implementation different from Open MPI, contact the help desk.
COMPILING-LINKING AND RUNNING MPI PROGRAMS
On the Eos (IBM iDataPlex) Linux Cluster

MPI Implementation: Open MPI

Compile and link your MPI code for the OpenMPI (NOT OpenMP) environment with the following commands (compiler wrappers):

**Fortran**

```bash
mpif90 -o prog.exe [-openmp] [options] prog.f90 ... 
```

**C**

```bash
mpicc  -o prog.exe [-openmp] [options] prog.c ... 
```

**C++**

```bash
mpic++ -o prog.exe [-openmp] [options] prog.cpc ... 
```

Interactive Case: (1) Set environment vars locally, if any; (2) Execute your MPI program. The `-openmp` is used only when an MPI process spawns multiple threads.

```bash
mpirun/mpiexec [-hostfile ./login.hosts] \
  [-x environment_var1] [-x environment_var2] . . .\ 
  [-npersnode procs_per_node] \ 
  [-tag-output] [-bynode] -np mpi_processes \ 
  prog.exe arg1 arg2 . . .
```

Additional information: See the mpirun manpage (man mpirun)
Program Hello_World
include "mpif.h" ! Or just say, USE MPI
integer np, myid
character*(MPI_MAX_PROCESSOR_NAME) proc_name
character (len=4) :: itoa
integer np, myid, name_len, ierr
!
call MPI_INIT(ierr)
if (ierr /= MPI_SUCCESS) then
  print *, "MPI initialization error"
  stop
end if

call MPI_COMM_RANK(MPI_COMM_WORLD, myid, ierr)
call MPI_COMM_SIZE(MPI_COMM_WORLD, np, ierr)
!
call MPI_Get_processor_name(proc_name, name_len, ierr) ! Gets node name on Eos
!
print *, '- Hello from Proc/node ', trim(proc_name),' MPI tasks # ', &
     trim(itoa(myid)),', out of ',trim(itoa(np)),' MPI tasks -'

call MPI_FINALIZE(ierr)
stop
End

RECURSIVE FUNCTION itoa(n) RESULT(str)
! Converts int# n to numeric str
CHARACTER (LEN=4) :: str ! limited to 4-digit integers
INTEGER, INTENT(IN) :: n
IF (n < 10) THEN
  str = CHAR(n + 48) ! Fortran intrinsic. Converts integers, 0-9, to numeric ascii strings
ELSE
  str = itoa(n/10)
  str = TRIM(str)//TRIM(CHAR(MOD(n, 10) + 48))
END IF
END FUNCTION itoa
EXAMPLE: MPI in C

```c
#include <stdio.h>
#include <string.h>
#include "mpi.h"

main(int argc, char **argv) {
    int myid, np, name_len, ierr;
    ierr = MPI_Init(&argc, &argv);
    char proc_name[20];
    if ( ierr == MPI_SUCCESS ) {
        MPI_Comm_size(MPI_COMM_WORLD,&np);
        MPI_Comm_rank(MPI_COMM_WORLD,&myid);
        MPI_Get_processor_name( proc_name, &name_len );
        printf("- Hello from MPI proc/node %s process id %d out of %d MPI tasks -\n", proc_name, myid, np);
        MPI_Finalize();
    } else
        printf("MPI initialization error\n" );
}
```
COMPILING-LINKING AND RUNNING MPI PROGRAMS
On the Eos (IBM iDataPlex) Linux Cluster

MPI Implementation: Open MPI

$ module list
Currently Loaded Modulefiles:
1) intel/compilers/11.1.073(default) 2) openmpi/1.4.3(default)

$ mpicc -o hello_world_ompi_c.exe hello_world.c
$ mpif90 -o hello_world_ompi_f.exe hello_world.f90

$ cat login.hosts
login001
login002
login003
login004
Login005

$ mpirun -np 4 ./hello_world_ompi_f.exe
- Hello from Proc/node login004 MPI process # 2 out of 4 MPI tasks -
- Hello from Proc/node login004 MPI process # 1 out of 4 MPI tasks -
- Hello from Proc/node login004 MPI process # 3 out of 4 MPI tasks -
- Hello from Proc/node login004 MPI process # 0 out of 4 MPI tasks -

spiros@login004 mpi]$ mpirun -hostfile ./login.hosts -np 4 \
./hello_world_ompi_f.exe 2>/dev/null <-- used to suppress message output ...
- Hello from Proc/node login001 MPI process # 0 out of 4 MPI tasks -
- Hello from Proc/node login003 MPI process # 2 out of 4 MPI tasks -
- Hello from Proc/node login002 MPI process # 1 out of 4 MPI tasks -
- Hello from Proc/node login004 MPI process # 3 out of 4 MPI tasks -

$ mpirun -hostfile ./login.hosts -host login001 -np 2 hello_world_ompi_f.exe : -host login005 -np 2 hello_world_ompi_c.exe 2>/dev/null
- Hello from MPI proc/node login005 process id 2 out of 4 MPI tasks -
- Hello from MPI proc/node login005 process id 3 out of 4 MPI tasks -
- Hello from Proc/node login001 MPI process # 0 out of 4 MPI tasks -
- Hello from Proc/node login001 MPI process # 1 out of 4 MPI tasks -
# Example job file 1

#PBS -l nodes=4:ppn=2,walltime=00:10:00,mem=2gb
#PBS -N helloworld1
#PBS -S /bin/bash
#PBS -j oe
#PBS other PBS directives
#
module load intel/compilers
module load openmpi
#
$PBS_O_WORKDIR
# the mpirun command will run 8 MPI tasks across 4 nodes, 2 MPI tasks/node
# assumes mpiprog.exe is in $PBS_O_WORKDIR
#
mpirun ./hello_world_ompi_f.exe

# PBS automatically conveys host node information to OpenMPI–no need for host file specification.
$ qsub hello_world.job  
Your job has been submitted as "76570.login006.sc.cluster.tamu".

$ more helloworld1*76570

-- Hello from Proc/node node086 MPI process # 6 out of 8 MPI tasks --
-- Hello from Proc/node node008 MPI process # 0 out of 8 MPI tasks --
-- Hello from Proc/node node008 MPI process # 1 out of 8 MPI tasks --
-- Hello from Proc/node node081 MPI process # 3 out of 8 MPI tasks --
-- Hello from Proc/node node085 MPI process # 4 out of 8 MPI tasks --
-- Hello from Proc/node node086 MPI process # 7 out of 8 MPI tasks --
-- Hello from Proc/node node081 MPI process # 2 out of 8 MPI tasks --
-- Hello from Proc/node node085 MPI process # 5 out of 8 MPI tasks --
EXAMPLE: MPI+OpenMP in Fortran

Program Hello_World_mpi_omp
USE MPI        !or include "mpif.h"
USE OMP_LIB    !or include "omp_lib.h"
character*(MPI_MAX_PROCESSOR_NAME) proc_name
character (len=4) :: itoa
integer np, myid, name_len, ierr, thread_id, thread_np
!
call MPI_INIT(ierr)
if (ierr /= MPI_SUCCESS) then
    print *, "MPI initialization error"
    stop
end if

!$OMP PARALLEL PRIVATE(thread_id, thread_np)
    thread_id = OMP_GET_THREAD_NUM(); thread_np = OMP_GET_NUM_THREADS()
    print *,'- Hello from Proc/node ', trim(proc_name),' MPI process # ',' &
    trim(itoa(myid)),' THREAD_ID ', trim(itoa(thread_id)),' out of ',trim(itoa(np)),' MPI tasks -'
!$OMP END PARALLEL
!
call MPI_FINALIZE(ierr)
stop end

RECURSIVE FUNCTION itoa(n) RESULT(str)
...
END FUNCTION itoa
EXAMPLE: MPI+OpenMP in C

#include <stdio.h>
#include <string.h>
#include "mpi.h"
#include <omp.h>

main(int argc, char **argv)
{
  int myid, np, name_len, thread_id ierr;
  ierr = MPI_Init(&argc, &argv);
  char proc_name[20];
  if ( ierr == MPI_SUCCESS ) {
    MPI_Comm_size(MPI_COMM_WORLD,&np);
    MPI_Comm_rank(MPI_COMM_WORLD,&myid);
    MPI_Get_processor_name( proc_name, &name_len );

    #pragma omp parallel private(thread_id)
    {
      thread_id = omp_get_thread_num();
      printf("- Hello from MPI proc/node %s MPIprocess id %d thread %d out of %d MPI tasks -\n", proc_name, myid, thread_id, np);
    }
    MPI_Finalize();
  }
  else
    printf( "MPI initialization error\n" );
}
MPI Implementation: Open MPI

$ mpicc -openmp -o hello_world_omp.exe hello_world_omp.c
/g/software/intel/Compiler/11.1/073/lib/intel64/libimf.so: warning: warning: feupdateenv is not implemented and will always

$ echo $OMP_NUM_THREADS $OMP_DYNAMIC
4 false

$ mpirun -hostfile ./login.hosts -x OMP_NUM_THREADS -x OMP_DYNAMIC \
   -npernode 1 -np 4 /hello_world_omp.exe 2>/dev/null

-Hello from MPI proc/node login003 MPIprocess id 2 thread 0 out of 4 MPI tasks -
-Hello from MPI proc/node login003 MPIprocess id 2 thread 1 out of 4 MPI tasks -
-Hello from MPI proc/node login002 MPIprocess id 1 thread 0 out of 4 MPI tasks -
-Hello from MPI proc/node login002 MPIprocess id 1 thread 1 out of 4 MPI tasks -
-Hello from MPI proc/node login001 MPIprocess id 0 thread 0 out of 4 MPI tasks -
-Hello from MPI proc/node login001 MPIprocess id 0 thread 2 out of 4 MPI tasks -
-Hello from MPI proc/node login004 MPIprocess id 3 thread 0 out of 4 MPI tasks -
-Hello from MPI proc/node login004 MPIprocess id 3 thread 3 out of 4 MPI tasks -
-Hello from MPI proc/node login004 MPIprocess id 3 thread 3 out of 4 MPI tasks -
-Hello from MPI proc/node login004 MPIprocess id 3 thread 2 out of 4 MPI tasks -
-Hello from MPI proc/node login004 MPIprocess id 3 thread 1 out of 4 MPI tasks -
-Hello from MPI proc/node login004 MPIprocess id 3 thread 0 out of 4 MPI tasks -
-Hello from MPI proc/node login002 MPIprocess id 1 thread 3 out of 4 MPI tasks -
-Hello from MPI proc/node login001 MPIprocess id 0 thread 1 out of 4 MPI tasks -
-Hello from MPI proc/node login003 MPIprocess id 2 thread 3 out of 4 MPI tasks -
-Hello from MPI proc/node login003 MPIprocess id 2 thread 2 out of 4 MPI tasks -
-Hello from MPI proc/node login003 MPIprocess id 2 thread 2 out of 4 MPI tasks -
-Hello from MPI proc/node login003 MPIprocess id 2 thread 1 out of 4 MPI tasks -
-Hello from MPI proc/node login003 MPIprocess id 2 thread 0 out of 4 MPI tasks -
-Hello from MPI proc/node login003 MPIprocess id 2 thread 0 out of 4 MPI tasks -
-Hello from MPI proc/node login003 MPIprocess id 2 thread 0 out of 4 MPI tasks -
-Hello from MPI proc/node login003 MPIprocess id 2 thread 0 out of 4 MPI tasks -
-Hello from MPI proc/node login003 MPIprocess id 2 thread 0 out of 4 MPI tasks -
-Hello from MPI proc/node login003 MPIprocess id 2 thread 0 out of 4 MPI tasks -

$ mpirun -np 1 ./hello_world_omp.exe

-Hello from MPI proc/node login003 MPIprocess id 0 thread 0 out of 1 MPI tasks -
-Hello from MPI proc/node login003 MPIprocess id 0 thread 0 out of 1 MPI tasks -
-Hello from MPI proc/node login003 MPIprocess id 0 thread 3 out of 1 MPI tasks -
-Hello from MPI proc/node login003 MPIprocess id 0 thread 2 out of 1 MPI tasks -
-Hello from MPI proc/node login003 MPIprocess id 0 thread 1 out of 1 MPI tasks -
COMPILING−LINKING AND RUNNING MPI+OpenMP PROGRAMS
On the Eos (IBM iDataPlex) Linux Cluster
MPI Implementation: Open MPI
*** Batch mode ***

###MPI (OpenMPI+OpenMP) 32-way job: 4 nodes x 2 MPI tasks per node x 4 OpenMP threads per MPI task
#PBS -l nodes=4:ppn=2,walltime=00:10:00,mem=2gb
#PBS -N helloworld1
#PBS -S /bin/bash
#PBS -j oe
###PBS other PBS directives
#
module load intel/compilers
module load openmpi
#
cd $PBS_O_WORKDIR
export OMP_NUM_THREADS=4 OMP_DYNAMIC=false OMP_STACKSIZE=100M
#
mpirun -x OMP_NUM_THREADS \    # PBS -v OMP_NUM_THREADS,... also works
   -x OMP_DYNAMIC \
   -x OMP_STACKSIZE \
   hello_world_omp_c.exe

# PBS automatically conveys host node information to OpenMPI—no need for host file specification.
MPI Implementation: Open MPI

$ cat helloworld2.o77270

-Hello from MPI proc/node node049 MPIprocess id 4 thread 0 out of 8 MPI tasks -
-Hello from MPI proc/node node049 MPIprocess id 5 thread 0 out of 8 MPI tasks -
-Hello from MPI proc/node node046 MPIprocess id 2 thread 0 out of 8 MPI tasks -
-Hello from MPI proc/node node045 MPIprocess id 1 thread 0 out of 8 MPI tasks -
-Hello from MPI proc/node node049 MPIprocess id 4 thread 1 out of 8 MPI tasks -
-Hello from MPI proc/node node045 MPIprocess id 1 thread 1 out of 8 MPI tasks -
-Hello from MPI proc/node node050 MPIprocess id 7 thread 0 out of 8 MPI tasks -
-Hello from MPI proc/node node050 MPIprocess id 7 thread 1 out of 8 MPI tasks -
-Hello from MPI proc/node node049 MPIprocess id 5 thread 3 out of 8 MPI tasks -
-Hello from MPI proc/node node049 MPIprocess id 5 thread 2 out of 8 MPI tasks -
-Hello from MPI proc/node node050 MPIprocess id 7 thread 3 out of 8 MPI tasks -
-Hello from MPI proc/node node050 MPIprocess id 7 thread 2 out of 8 MPI tasks -
-Hello from MPI proc/node node045 MPIprocess id 1 thread 3 out of 8 MPI tasks -
-Hello from MPI proc/node node045 MPIprocess id 1 thread 2 out of 8 MPI tasks -
-Hello from MPI proc/node node049 MPIprocess id 5 thread 1 out of 8 MPI tasks -
-Hello from MPI proc/node node049 MPIprocess id 4 thread 3 out of 8 MPI tasks -
-Hello from MPI proc/node node050 MPIprocess id 6 thread 0 out of 8 MPI tasks -
-Hello from MPI proc/node node050 MPIprocess id 6 thread 1 out of 8 MPI tasks -
-Hello from MPI proc/node node045 MPIprocess id 0 thread 0 out of 8 MPI tasks -
-Hello from MPI proc/node node045 MPIprocess id 0 thread 1 out of 8 MPI tasks -
-Hello from MPI proc/node node045 MPIprocess id 0 thread 2 out of 8 MPI tasks -
-Hello from MPI proc/node node049 MPIprocess id 4 thread 2 out of 8 MPI tasks -
-Hello from MPI proc/node node050 MPIprocess id 6 thread 2 out of 8 MPI tasks -
-Hello from MPI proc/node node046 MPIprocess id 2 thread 1 out of 8 MPI tasks -
-Hello from MPI proc/node node046 MPIprocess id 3 thread 0 out of 8 MPI tasks -
-Hello from MPI proc/node node046 MPIprocess id 2 thread 2 out of 8 MPI tasks -
-Hello from MPI proc/node node046 MPIprocess id 2 thread 3 out of 8 MPI tasks -
-Hello from MPI proc/node node046 MPIprocess id 3 thread 2 out of 8 MPI tasks -
-Hello from MPI proc/node node046 MPIprocess id 3 thread 3 out of 8 MPI tasks -
-Hello from MPI proc/node node050 MPIprocess id 6 thread 3 out of 8 MPI tasks -
-Hello from MPI proc/node node046 MPIprocess id 3 thread 1 out of 8 MPI tasks -
-Hello from MPI proc/node node046 MPIprocess id 0 thread 3 out of 8 MPI tasks -
COMPILING–LINKING AND RUNNING MPI PROGRAMS
On the Eos (IBM iDataPlex) Linux Cluster

MPI Implementation: INTEL MPI
(Interactive Case)

To compile and run interactively standard mpi codes on Eos you need to insure that:

- In the current/execution directory the login.hosts file exists and has up to five entries (see below) corresponding to the five available login nodes.
- You must not mix different MPI implementations, e.g., INTEL MPI & Open MPI.
- You must not use more than 4 cpus/cores per (login) node.
- Only for small & short (less than 15 minutes) duration executions.
- The terms MPI process and MPI task are used interchangeably here.
- Must load the Intel compilers and Intel’s MPI

```
module load intel/compilers
module load intel/mpi

./login.hosts  -> file needed only when using multiple nodes
```

```
In execution
directory

login001
login002
:
:
login005
```

It specifies which nodes are available for parallel mpi processing during interactive sessions/logins. This is the simplest form for a hosts file.

There are several implementations of the MPI standard. Open MPI is the one we recommend. If your MPI program/application requires an MPI implementation different from Open MPI, contact the help desk.
COMPILING-LINKING AND RUNNING MPI PROGRAMS
On the Eos (IBM iDataPlex) Linux Cluster
MPI Implementation: INTEL MPI

Compile and link your MPI code for the INTEL MPI environment with the following commands (compiler wrappers):

**Fortran**

```
mpiifort -o prog.exe [-openmp] [options] prog.f90 ...
```

**C**

```
mpiicc -o prog.exe [-openmp] [options] prog.c ...
```

**C++**

```
mpiicpc -o prog.exe [-openmp] [options] prog.cpc ...
```

Interactive Case: (1) Set environment vars locally, if any; (2) Execute your MPI program. The `-openmp` is used only when an MPI process spawns multiple threads.

```
mpirun [-f ./login.hosts] [-r ssh] [-l] \ 
[-genv environment_var1] [-genv environment_var2] . . .\ 
[-ppn procs_per_node] [-rr] \ 
-pp mpi_processes \ 
prog.exe arg1 arg2 . . .
```

Additional information: See the mpirun manpage (man mpirun/mpiexec)
Compile and link your MPI code for the **INTEL MPI** environment with the following commands (compiler wrappers):

**Fortran**

    mpiifort -o prog.exe [-openmp] [options] prog.f90 ...

**C**

    mpiicc  -o prog.exe [-openmp] [options] prog.c ...

**C++**

    mpiicpc -o prog.exe [-openmp] [options] prog.cpc ...

Batch: (1) Set environment vars locally, if any; (2) Execute your MPI program. The `-openmp` is used only when an MPI process spawns multiple threads.

    pbs.mpiexec [-config config_file] [-kill]\
                 [-pernode procs_per_node] [-npervnode nprocs_per_node] \ 
                 -np mpi_processes \ 
                 prog.exe arg1 arg2 ... 

Environment vars are propagated by setting the `-v` or `-V` PBS options

Additional information: See the pbs.mpirun manpage
SCHEDULING OF LOOP ITERATIONS

- MUST BE DONE EXPLICITLY BY USER

- SCHEDULING IS THE ASSIGNMENT OF SPECIFIC ITERATIONS TO SPECIFIC MPI PROCESSES

- TYPES OF SCHEDULING
  - BLOCK DISTRIBUTION
  - CYCLIC DISTRIBUTION
  - BLOCK-CYCLIC DISTRIBUTION
  - SELF-SCHEDULING
  - OTHER
SCHEDULING LOOP ITERATIONS
BY BLOCK DISTRIBUTION

- FOR NP AVAILABLE PROCESSES, THE NUMBER OF ITERATIONS, N, IS DIVIDED INTO NP PARTS OR BLOCKS. EACH BLOCK CONSISTS OF CONSECUTIVE ITERATIONS

- SEVERAL WAYS AVAILABLE TO DISTRIBUTE N ITERATIONS TO NP PROCESSES IN BLOCK FASHION

- A COMMON APPROACH:

\[ N = NP \times Q + R \quad \text{(floor}(N/NP) = Q) \]

ASSIGN Q+1 ITERATIONS TO EACH OF PROCESSES 0, 1, ..., R-1
ASSIGN Q ITERATIONS TO EACH OF PROCESSES R, R+1, ..., NP-1
(N.B. 0<= R <= Q-1)

- EXAMPLE: N1=1, N2=11, NP=3

<table>
<thead>
<tr>
<th>Iter #</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
</tr>
</thead>
<tbody>
<tr>
<td>Proc #</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
</tbody>
</table>
SCHEDULING LOOP ITERATIONS
BY BLOCK DISTRIBUTION

SCALAR TO MPI

CALL BLOCK_MAP(N1, N2, NPROCS, MYID, L1, L2)

DO I=N1, N2
  Computation
END DO

DO I=L1, L2
  Computation
END DO

SUBROUTINE BLOCK_MAP(N1, N2, NPROCS, MYID, L1, L2)
  ! Assigns L2-L1+1 consecutive iterations to MYID process. Assigned number of iterations vary by, at most, 1.
  INTEGER N1, N2, NPROCS, MYID, MYSIZ, L1, L2, RES
  !
  MYSIZ = (N2-N1+1)/NPROCS
  RES = MOD(N2-N1+1,NPROCS)
  IF ( MYID < RES ) THEN  ! First RES-1 proc’s get one extra
    MYSIZ = MYSIZ + 1
    L1 = N1 + MYSIZ*MYID
  ELSE
    L1 = N1 + RES + MYID*MYSIZ
  ENDIF
  L2 = MIN(L1 + MYSIZ - 1,N2)
  !
  RETURN
END
SCHEDULING LOOP ITERATIONS
BY CYCLIC DISTRIBUTION

FOR NP AVAILABLE PROCESSES, EACH OF THE N ITERATIONS
ARE ASSIGNED IN A ROUND-ROBIN MANNER TO THOSE PROCESSES

- CODE NEEDED TO TRANSFORM

SCALAR           TO           MPI

DO I = N1, N2
    Computation
END DO

DO I = N1+MYID, N2, NP
    Computation
END DO

OR

DO I = N1, N2
    IF (MOD(I-N1,NP)== MYID) THEN
        Computation
    END IF
END DO

Note: If N1<0 use ABS(MOD(I-1,NP))

EXAMPLE: N1=1, N2=11; NP=3

<table>
<thead>
<tr>
<th>Iter #</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
</tr>
</thead>
<tbody>
<tr>
<td>Proc #</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>
EXAMPLE: TWO CASES OF CYCLIC DISTRIBUTION

Assume each MPI process has whole copies of array A

Cyclic row-wise computation

\[
\text{DO } I = N1+\text{MYID, N2, NP} \\
\text{DO } J = N1, N2 \\
A(I,J) = \text{Computation} \\
\text{END DO} \\
\text{END DO}
\]

Cyclic column-wise computation

\[
\text{DO } J = N1+\text{MYID, N2, NP} \\
\text{DO } I = N1, N2 \\
A(I,J) = \text{Computation} \\
\text{END DO} \\
\text{END DO}
\]
SCHEDULING LOOP ITERATIONS
BY BLOCK-CYCLIC DISTRIBUTION

N ITERATIONS ARE PARTITIONED IN EQUALLY SIZED BLOCKS OF CONSECUTIVE ITERATIONS. THESE BLOCKS ARE ASSIGNED TO THE NP AVAILABLE PROCESSES IN A ROUND-ROBIN MANNER. THE (LEXICALLY) LAST ASSIGNED BLOCK OF ITERATIONS MAY BE SMALLER.

SCALAR TO MPI

BLKSZ =< floor((N2-N1+1)/NP) ! One way
DO I = N1+MYID*BLKSZ, N2, NP*BLKSZ
DO K = I, MIN(I+BLKSZ-1, N2)
Computation
END DO
END DO

EXAMPLE: N1=1, N2=17, NP=3; BLKSZ=3

<table>
<thead>
<tr>
<th>Iter #</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
</tr>
</thead>
<tbody>
<tr>
<td>Proc #</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Iter #</th>
<th>12</th>
<th>13</th>
<th>14</th>
<th>15</th>
<th>16</th>
<th>17</th>
</tr>
</thead>
<tbody>
<tr>
<td>Proc #</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>2</td>
</tr>
</tbody>
</table>

Comments:
Note that the selection BLKSZ=(Number of Loop Iterations)/NP, as above, renders block-cyclic scheduling equivalent to that of block scheduling. We can set the value of BLKSZ to anything appropriate. The optimal value depends on the specific loop.
PROGRAM COS_LOOP    ! Serial Version
INTEGER, PARAMETER :: N=40
REAL (KIND=8) AA(N), ANG_INCR, PI
!
OPEN(UNIT=10, FILE='scalar_cos.dat',FORM='FORMATTED',&
     STATUS='UNKNOWN')
PI = 4.0D1 * ATAN(1.0D1)
ANG_INCR = (2.0D1 * PI)/N
!
DO I = 1, N
    AA(I) = COS( REAL(I, KIND=8)*ANG_INCR )
END DO
!
WRITE(UNIT=10, FMT='(8(F6.4,X))' ) AA
STOP
END

titan% f90 -o cos_loop.exe cos_loop.f90

titan% cos_loop.exe

titan% cat scalar_cos.dat

- .4101 -.6636 0.9544 -.1192 -.8566 0.8219 0.1825 -.9716
 0.6144 0.4676 -.9979 0.3509 0.7101 -.9334 0.0554 0.8879
-.7837 -.2451 0.9847 -.5626 -.5233 0.9918 -.2902 -.7537
 0.9085 0.0086 -.9155 0.7424 0.3066 -.9939 0.5086 0.5767
-.9816 0.2284 0.7943 -.8799 -.0726 0.9394 -.6979 -.3669

titan%
EXAMPLE: BLOCK DISTRIBUTION

PROGRAM MPI_COS_LOOP
INCLUDE "mpif.h"
INTEGER np, myid, ierr
INTEGER, PARAMETER :: N=40  ! Total Array size (=NXN)
REAL (KIND=8), ALLOCATABLE :: AA_LOCAL(:)
REAL (KIND=8) ANG_INCR, PI
CHARACTER (LEN=2) itoa
CHARACTER (LEN=12) FN, fmtstr  ! File Name & write format strng
!
call MPI_INIT(ierr)
call MPI_COMM_RANK(MPI_COMM_WORLD, myid, ierr)
call MPI_COMM_SIZE(MPI_COMM_WORLD, np, ierr)
!
FN = 'mpi_cos'//CHAR(48+myid)//'.dat'  ! Proc-specific file name
OPEN(UNIT=myid+10, FILE=FN, FORM='formatted', STATUS='unknown')
!
PI = 4.0D1 * ATAN(1.0D1)
ANG_INCR = (2.0D1 * PI)/N  ! Computation increment
  ! Map iterations L1 thru L2 to myid

mydim = L2 - L1 + 1
PRINT *, 'myid =',myid,' array_size per process =',mydim
!
ALLOCATE( AA_LOCAL(1:mydim) )  ! AA_LOCAL per process
!
DO I=L1, L2
    AA_LOCAL(I-L1+1) = COS( REAL(I,KIND=8) * ANG_INCR )
END DO
!
fmtstr = '(//ITOA(mydim)('('F6.4,X))'
WRITE(UNIT=myid+10, FMT=fmtstr) AA_LOCAL
DEALLOCATE( AA_LOCAL )
!
call MPI_FINALIZE(ierr)
STOP
END
RECURSIVE FUNCTION itoa(n) RESULT(str)
! Converts int# to numeric strg
CHARACTER (LEN=4) :: str
INTEGER, INTENT(IN) :: n
IF (n < 10) THEN
  str = CHAR(n + 48)
ELSE
  str = itoa(n/10)
  str = TRIM(str)/TRIM(CHAR(MOD(n, 10) + 48))
END IF
END FUNCTION itoa

titan% f90 -o mpi_cos_loop.exe mpi_cos_loop.f90 -lmpi
titan% mpirun -np 6 mpi_cos_loop.exe

myid = 0 array_size per process = 7
myid = 1 array_size per process = 7
myid = 2 array_size per process = 7
myid = 3 array_size per process = 7
myid = 4 array_size per process = 6
myid = 5 array_size per process = 6

titan% cat mpi_cos[0-5].dat
-.4101 -.6636 0.9544 -.1192 -.8566 0.8219 0.1825
-.9716 0.6144 0.4676 -.9979 0.3509 0.7101 -.9334
0.0554 0.8879 -.7837 -.2451 0.9847 -.5626 -.5233
0.9918 -.2902 -.7537 0.9085 0.0086 -.9155 0.7424
0.3066 -.9939 0.5086 0.5767 -.9816 0.2284
0.7943 -.8799 -.0726 0.9394 -.6979 -.3669
### MPI PREDEFINED DATA TYPES

<table>
<thead>
<tr>
<th>MPI DATA TYPE</th>
<th>C / C++</th>
<th>FORTRAN DATA TYPE</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_CHAR</td>
<td>Signed char</td>
<td>MPI_INTEGER</td>
</tr>
<tr>
<td>MPI_SHORT</td>
<td>signed short int</td>
<td>MPI_REAL</td>
</tr>
<tr>
<td>MPI_INT</td>
<td>signed int</td>
<td>MPI_REAL8 or DOUBLE PRECISION</td>
</tr>
<tr>
<td>MPI_LONG</td>
<td>signed long int</td>
<td>MPI_REAL16</td>
</tr>
<tr>
<td>MPI_UNSIGNED_CHAR</td>
<td>unsigned char</td>
<td>MPI_REAL16 (16-byte)</td>
</tr>
<tr>
<td>MPI_UNSIGNED_SHORT</td>
<td>unsigned short int</td>
<td>MPI_COMPLEX[8]</td>
</tr>
<tr>
<td>MPI_UNSIGNED</td>
<td>unsigned int</td>
<td>MPI_COMPLEX*16 (8-byte, 8-byte)</td>
</tr>
<tr>
<td>MPI_UNSIGNED_LONG</td>
<td>unsigned long int</td>
<td>MPI_DOUBLE_COMPLEX</td>
</tr>
<tr>
<td>MPI_FLOAT</td>
<td>float</td>
<td>MPI_LOGICAL</td>
</tr>
<tr>
<td>MPI_DOUBLE</td>
<td>double</td>
<td>MPI_CHARACTER</td>
</tr>
<tr>
<td>MPI_LONG_DOUBLE</td>
<td>long double</td>
<td>MPI_BYTE</td>
</tr>
<tr>
<td>MPI_BYTE</td>
<td>NONE</td>
<td>MPI_PACKED</td>
</tr>
<tr>
<td>MPI_PACKED</td>
<td>NONE</td>
<td></td>
</tr>
</tbody>
</table>

Within MPI communication calls specify MPI DATA TYPES
POINT-TO-POINT COMMUNICATION: SEND

MPI_SEND(buf, count, datatype, dest, tag, comm, ierr)

Sends a message to dest in blocking mode

buf  IN  starting address of send buffer. Special value: MPI_BOTTOM. MPI_BOTTOM is a special address and, typically, used when the message length is zero

count  IN  (integer) number of MPI_Datatype contiguous elements to send to dest process

datatype  IN  the MPI_Datatype of the elements in the send buffer

dest  IN  (integer) process id of destination process Special constant: MPI_PROC_NULL

tag  IN  (integer) message tag (integer). Adds specificity to a message. Wildcards:
   MPI_ANY_TAG. (0 =< tag <= MPI_TAG_UB)

comm  IN  (integer in Fortan; MPI_Comm in C) communicator

The process that executes this call causes count contiguous elements of type MPI_Datatype to be sent, starting from buf’s address, to the process specified by dest. The message sent by MPI_SEND can be received by either MPI_RECV or MPI_IRECV. It must be emphasized that MPI_SEND does not return (i.e., it blocks) until it is safe to use the send buffer, buf, again. For many MPI implementations that means that MPI_SEND does not return until the message in buf has been copied to a system buffer.
POINT-TO-POINT COMMUNICATION: RECV

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

Receives a message from source in blocking mode

buf OUT address of receive buffer

count IN (integer) maximum size of the receive buffer

datatype IN the MPI_Datatype of elements in the receive buffer

source IN (integer) process id of source process. Wildcards: MPI_ANY_SOURCE

tag IN (integer) message tag. Wildcards: MPI_ANY_TAG

comm IN (integer in Fortan; MPI_Comm in C) communicator

status(*) OUT (integer in Fortran; MPI_Status in C) status object.

Status(*) provides a way of retrieving actual source, tag, and, indirectly, message length information. In C it is a structure typically passed by reference, in Fortan it is an integer array of size, MPI_STATUS_SIZE.
POINT-TO-POINT COMMUNICATION: RECV

Fortran

   integer status(MPI_STATUS_SIZE)
   :
   :
   source_id = status(MPI_SOURCE)
   tag = status(MPI_TAG)

C

   MPI_Status status;
   :
   :
   source_id = status.MPI_SOURCE
   tag = status.MPI_TAG

Attempts to receive in blocking mode a message with the specified envelope (source, tag, comm). When the message arrives, elements of the specified MPI_Datatype are placed into the buffer in contiguous locations, starting at address buf. The buffer starting at buf is assumed preallocated and has capacity for count MPI_Datatype elements. If the length of the message exceeds the capacity of the buffer, MPI_RECV will report an error. MPI_RECV can receive a message sent by either MPI_SEND or MPI_ISEND. Agreement in MPI_Datatype and tag between the send and the receive is also required.

MPI_RECV returns only after the message has been copied into buf. The actual size of the received message can be extracted by using the MPI_GET_COUNT routine which uses the status object as an input parameter.
Process 0: User Mode

Call mpi_send
(sendbuf, dest=1)
(blocked)
Time line progress
Now sendbuf can be reused

Receive data from src to sysbuf
Copying data from sysbuf to recvbuf

Process 1: User Mode

Call mpi_recv
(recbuf, scr=0)
(blocked)
Now recvbuf contains valid data

Copying data from sysbuf to recvbuf
program Send_RecV
USE MPI
CHARACTER (LEN=19) FN      ! File Name strg
CHARACTER (LEN=34) fmtstr ! write fmt strg
integer value, rank, np, i, ierr, status(MPI_STATUS_SIZE)
!
call MPI_INIT( ierr )
call MPI_COMM_RANK( MPI_COMM_WORLD, rank, ierr )
call MPI_COMM_SIZE( MPI_COMM_WORLD, np, ierr )
!
FN = 'mpi_send_recv'//CHAR(48+rank)//'.dat'
OPEN(UNIT=rank+10, FILE=FN, FORM='formatted')
!
do
  if ( rank == 0 ) then
    print *,'Enter integer to transmit: (<0 quits)'
    read(*,fmt='(i10)') value
    call MPI_Send(value, 1, MPI_INT, rank + 1, 0, &
                   MPI_COMM_WORLD,ierr)
  else
    call MPI_Recv(value, 1, MPI_INT, rank - 1,0, &
                   MPI_COMM_WORLD, status,ierr)
    !
    fmtstr = 'Process ',I2,X,'got value ',I3'
    WRITE(UNIT=rank+10, FMT=fmtstr) rank, value
  !
  if (rank < np - 1) then
    call MPI_Send(value, 1,MPI_INT,rank +1, 0, &
                   MPI_COMM_WORLD, ierr)
  end if
end do
!
close(rank+10)
!
call MPI_FINALIZE(ierr) stop end

K2% f90 -o mpi_send_recv.exe mpi_send_recv.f90 -lmpi
K2% mpirun -np 6 mpi_send_recv.exe 
   Enter integer to transmit: (<0 quits) 666
   Enter integer to transmit: (<0 quits) -1
K2% cat mpi_send_recv[0-5].dat
Process  1 got value 666
Process  1 got value  -1
Process  2 got value 666
Process  2 got value  -1
Process  3 got value 666
Process  3 got value  -1
Process  4 got value 666
Process  4 got value  -1
Process  5 got value 666
Process  5 got value  -1

if ( value < 0 ) exit     ! All procs exit
POINT-TO-POINT NONBLOCKING COMMUNICATION: ISEND

MPI_ISEND(buf, count, datatype, dest, tag, comm, request)

Send a message to dest in nonblocking mode

buf IN address of send buffer

count IN (integer) number of MPI_Datatype elements to send

datatype IN the MPI_Datatype of send buffer

dest IN (integer) process id of destination process; or MPI_PROC_NULL

tag IN (integer) message tag. Wildcards: MPI_ANY_TAG

comm IN (integer in Fortran; MPI_Comm in C) Communicator

request OUT integer in Fortran; MPI_Request in C) The communication request handle
Similar semantics as MPI_SEND except that the sending operation executes in nonblocking mode. But now when this routine starts a send, it returns immediately a request handle, which can be used for testing/quering whether a specific communication has completed (using MPI_TEST or MPI_WAIT).

The send buffer, buf, cannot be accessed or modified correctly before the request has been completed. The messages sent by MPI_ISEND can be received by either MPI_RECV or MPI_IRECV.

Used correctly, the mpi_isend and mpi_irecv routines can speedup execution because, between their issue and completion, one can carry out useful computation that does not involve the communication buffers. They also make point-to-point transfers more secure by not depending on the size of system buffers.
POINT-TO-POINT NONBLOCKING COMMUNICATION : IRECV

MPI_IReCV (buf, count, datatype, source, tag, comm, request)

Receive a message from source in nonblocking mode

buf OUT address of receive buffer

count IN (integer) maximum size of the receive buffer

datatype IN the MPI_Datatype of receive buffer

source IN (integer) process id of source process; or MPI_ANY_SOURCE

tag IN (integer) message tag. Wildcards: MPI_ANY_TAG

comm IN (integer in Fortran; MPI_Comm in C) communicator

request OUT (integer in Fortran; MPI_Request in C) The communication request handle

Starts a nonblocking receive operation and returns a request handle. This can subsequently be used in an MPI_WAIT call.

All other semantics & operational features are completely analogous to MPI_ISEND.
ANCILLARY ROUTINES FOR SEND & RECEIVE

**MPI_TEST**  
Used to determine whether a specific request made with an MPI_ISEND (or an MPI_IRECV) has completed delivery (or receipt) of a message

**MPI_PROBE/MPI_IProbe**  
Used to determine whether messages have arrived and need to be received that match a source and a tag

**MPI_GET_COUNT**  
Used to determine the actual size of messages given the status returned by MPI_RECV, MPI_PROBE, MPI_WAIT, or MPI_TEST

**MPI_WAIT**  
Waits for and MPI_ISEND or an MPI_IRECV to complete

**MPI_WAITALL**  
Waits for the completion of all the issued MPI_ISEND's and MPI_IRECV's
MPI_TEST( request, flag, status, ierr )

Determines completion status of an MPI_ISEND or MPI_IRECV Request

IN Request handle returned by MPI_ISEND or MPI_RECV

OUT (logical in Fortran; Int in C) .TRUE. or .FALSE. For successful completion

status(*) OUT Status object. Same as in MPI_RECV, ...
Example:

** Fortran **
integer comm, count, request
integer status(MPI_STATUS_SIZE), ierr
logical flag
:
call MPI_Irecv (recv_buf, count, ..., &
comm, request, ierr)
do
     call MPI_Test(request, flag, status, &
ierr)
     if ( flag == .FALSE. ) then
          ... computation not using recv_buf ...
     else
         exit
     end if
end do

Example:

** C **
MPI_Comm comm;
MPI_Request request;
MPI_Status status;
int flag;
:
ierr = MPI_Irecv (rec_buf, count, ..., 
comm, &request)
do {
ierr = MPI_Test( &request, &flag,&status)
If ( flag !=0 ) {
           ... computation not using rec_buf ...
         }
else break;
}
while ( 1 )
MPI_PROBE( source, tag, comm, status, ierr)
MPI_IPROBE(source, tag, comm, flag, status, ierr)

Blocking and non-blocking versions for determining whether messages have arrived (that need to be received locally) that match a source, tag, and communicator combination.

source IN (integer) source process id of the expected message
tag IN (integer) tag of expected message
comm IN communicator
flag OUT (logical) .TRUE, for successful arrival; .FALSE. for an unsuccessful one
status(*) OUT status object
GET_COUNT

MPI_GET_COUNT( status, datatype, count, ierr)

Returns the actual size of message that has arrived

status(*) IN Status object returned by an MPI_RECV, MPI_PROBE, MPI_IPROBE,
MPI_WAIT, or an MPI_TEST

datatype IN the MPI_Datatype of the elements in the message

count OUT (integer) number of MPI_Datatype elements involved in a message that
has arrived (or pending by first using MPI_PROBE)

Comment:
In both MPI_RECV and MPI_Irecv the receiving buffer is preallocated. It may happen
that the size of this buffer is smaller or larger than the size of the actual message
to be received. When the message size is larger than the buffer size, calls to
MPI_PROBE and MPI_GET_COUNT prior to posting a blocking receive will give the correct
message size, thus enabling the allocation of enough buffer space prior to issuing the
actual receive—otherwise, an error will occur.
**WAIT**

MPI_WAIT(request, status, ierror)

Waits for a non-blocking send or receive to complete

request IN Specifies the request (handle) supplied by isend or irecv

status(*) OUT Status object.

Can be used with non-blocking operations to overlap computation with communication, as well as avoid deadlock.

call mpi_irecv(buf, ..., ireq, ...) :
    computation that does not use buf :
    call mpi_Wait(ireq, ...)

WAITALL

MPI_WAITALL( count, array_of_requests, array_of_statuses, ierr)

Waits for the completion of the specified requests

count IN Number of requests (MPI_ISEND & MPI_IRECV to wait for

array_of_requests IN Array of request handles for MPI_ISENDs
(integer) and MPI_IRECVs

array_of_statuses OUT Array of status objects. See example

Example:

integer status_array(MPI_STATUS_SIZE, 4), req(4), ierr :
  call MPI_IRECV( ..., req(1), ierr)
call MPI_ISEND( ..., req(2), ierr)
call MPI_IRECV( ..., req(3), ierr)
call MPI_ISEND( ..., req(4), ierr)
  :
  :
call MPI_WAITALL(4, req, status_array, ierr)
BARRIER

MPI_BARRIER(comm, ierr)

comm IN communicator
ierr OUT return code

A call to barrier blocks until all processes within comm have called/entered it (Once all processes in comm have entered it, it is unpredictable which process will return first). All processes in comm must call this routine.

WTIME

MPI_WTIME() (double precision fn) Returns the elapsed wall-clock time in seconds

When called by a process, mpi_wtime() returns the elapsed wall-clock time in seconds since some arbitrary point in the past that is guaranteed not to change during the lifetime of the process. Finds typical use in timing various computation segments.

Example
REAL (KIND=8) t1, t2, elapsed_tm
:
call MPI_Barrier(MPI_COMM_WORLD, ierr)
t1 = MPI_Wtime()
:
    Computation
:
t2 = MPI_Wtime()
elapsed_tm = t2 - t1
Example: Isend & Irecv

PROGRAM Isend_Irecv
    ! Nonblocking Send/Recv
    ! Not to be run for more than 9 processes
USE MPI
CHARACTER (LEN=21) FN           ! File Name strg
CHARACTER (LEN=54) fmtstr        ! Write format strg
INTEGER VALUE, RANK, NP, I, IREQ, IERR, ISTATUS(MPI_STATUS_SIZE)

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

    FN = 'mpi_isend_irecv'//CHAR(48+RANK)//'.dat'      ! Proc-specific file name

    OPEN(UNIT=RANK+10, FILE=FN, FORM='formatted', STATUS='unknown')

    fmtstr = '("Process ",I2,X,"got value ",I4," from process ",I2)'
    do
        if ( rank == 0 ) then
            print *, 'Enter integer to transmit: (<0 quits)'
            read(*,fmt='(i10)') value
            call MPI_ISend(value, 1, MPI_INT, rank + 1, 0, MPI_COMM_WORLD, ireq, ierr)
        else
            call MPI_IRecv(value, 1, MPI_INT, rank - 1, 0, MPI_COMM_WORLD, ireq, ierr)
            call MPI_WAIT(ireq, istatus, ierr)
            WRITE(UNIT=rank+10, FMT=fmtstr) rank, value, istatus(MPI_SOURCE)
            if (rank < np - 1) then
                call MPI_ISend(value, 1, MPI_INT, rank + 1, 0, MPI_COMM_WORLD, ireq, ierr)
            end if
        end if
    end do
    if (value < 0) exit
end do
close(rank+10)     !
Close each opened file
     !
call MPI_FINALIZE(ierr)
stop
end

f90 -o mpi_isend_irecv.exe \
     mpi_isend_irecv.f90 -lmpi

> mpirun -np 6 mpi_isend_irecv.exe
Enter integer to transmit: (<0 quits) 111
Enter integer to transmit: (<0 quits) 99
Enter integer to transmit: (<0 quits) 888
Enter integer to transmit: (<0 quits) -1

> cat mpi_isend_irecv[0-5].dat

Process  1 got value  111 from process  0
Process  1 got value  99 from process  0
Process  1 got value  888 from process  0
Process  1 got value   -1 from process  0
Process  2 got value  111 from process  1
Process  2 got value  99 from process  1
Process  2 got value  888 from process  1
Process  2 got value   -1 from process  1
Process  3 got value  111 from process  2
Process  3 got value  99 from process  2
Process  3 got value  888 from process  2
Process  3 got value   -1 from process  2
Process  4 got value  111 from process  3
Process  4 got value  99 from process  3
Process  4 got value  888 from process  3
Process  4 got value   -1 from process  3
Process  5 got value  111 from process  4
Process  5 got value  99 from process  4
Process  5 got value  888 from process  4
Process  5 got value   -1 from process  4
DEADLOCKS IN BIDIRECTIONAL COMMUNICATION

Two or more processes exchanging data can cause each other to stall, either because of making the send and recv calls in the incorrect order, or because of the limited size of the system buffer(s).

The order by which both (rank = 0 and rank = 1) processes call the send and receive routines presents three major cases.

Case 1 Both processes first call the send routine, and the receive routine next.

Case 2 Both processes first call the receive routine, and the send routine next.

Case 3 One process first calls send and then receive, while the other calls them in the opposite order.
**DEADLOCKS IN BIDIRECTIONAL COMMUNICATION - CASE 1**

Case 1: Send first and Then Receive

```fortran
if (myrank==0) then
    call mpi_send(sendbuf, ...)
    call mpi_recv(recvbuf, ...)
elseif (myrank==1) then
    call mpi_send(sendbuf, ...)
    call mpi_recv(recvbuf, ...)
endif
```

If the size of `sendbuf` for both processes is smaller than the system buffer, then the above code completes correctly. When `sendbuf` of, say, the rank 0 process is larger than the system buffer, then the `mpi_send` call by rank 0 does not return, and, hence, nor can the `mpi_recv` call of rank 1. A deadlock ensues.

The situation does not change when in above code we replace:

```fortran
call mpi_send with call mpi_isend call mpi_wait
```

On the other hand, the following code is free from deadlock, basically because the program immediately returns from `mpi_isend` and starts receiving data from the other process:

```fortran
if (myrank==0) then
    call mpi_isend(sendbuf, ..., ireq, ...)
    call mpi_recv(recvbuf, ...)
    call mpi_Wait(ireq, ...)
elseif (myrank==1) then
    call mpi_isend(sendbuf, ..., ireq, ...)
    call mpi_recv(recvbuf, ...)
    call mpi_Wait(ireq, ...)
endif
```

The above works even when `sendbuf` is larger than the associated system buffer.
Case 2: Receive first and then send

The following code will always cause deadlock no matter how large the system buffer is,

if (myrank==0) then
    call mpi_recv(recvdbuf, ...)
    call mpi_send(sendbuf, ...)
elseif (myrank==1) then
    call mpi_recv(recvdbuf, ...)
    call mpi_send(sendbuf, ...)
endif

The following code is free from deadlock,

if (myrank==0) then
    call mpi_irecv(recvdbuf, ..., ireq, ...)
    call mpi_send(sendbuf, ...)
    call mpi_Wait(ireq, ...)
elseif (myrank==1) then
    call mpi_irecv(recvdbuf, ..., ireq, ...)
    call mpi_send(sendbuf, ...)
endif

Case 3: One process sends and receives; the other receives and sends

In the following arrangements, the first one may encounter a problem, other things being equal, if the sent message overflows the system buffers. The second runs no such risk.

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

A safe and perhaps a better performer,

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, ...)
COLLECTIVE COMMUNICATION CALLS

MPI_BCAST - Broadcast the same message from root to all processes in comm

MPI_GATHER - Collect individual messages in comm at the root process

MPI_GATHERV - Collect individual messages in comm of variable size and displacement at the root process

MPI_ALLGATHER - Collects individual messages of the same size from each process in comm and distributes the resulting message to all processes

MPI_ALLGATHERV - Same as MPI_ALLGATHER except the messages can be of variable size

MPI_REDUCE - Applies a reduction operation to the vector of elements in sendbuf over the group of processes specified in comm and places the result in recvbuf on root

MPI_ALLREDUCE - Same as MPI_REDUCE except the result is placed in recvbuf on all the processes in comm

MPI_SCATTER - Distribute individual messages from root to each process in comm

MPI_SCATTERV - A SCATTER capable of variable length messages and displacements

MPI_OP_CREATE - Creates a user-defined reduction operation and binds it to an op handle

DOCUMENTATION
For the latest available information on any of the above consult its man page; e.g., man mpi_reduce.

Note: The above is not a complete list of the bulk communication routines available in MPI.

* ALL PROCESSES IN A COMMUNICATOR MUST CALL
**BROADCAST**

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

Broadcast same data from root to all procs in comm

- inbuf: **IN/OUT** address of input buffer
- incnt: **IN** number of intype elements in inbuf
- intype: **IN** the MPI_Datatype of input buffer
- root: **IN** process id of root process
- comm: **IN** communicator

This routine broadcasts/sends the same message from root's send buffer, inbuf, to the receive buffer (above also called inbuf) of all processes in comm. The root process may return after performing its function even before any other process has called mpi_bcast. Each receiving process blocks until the message sent by root has arrived in its buffer.

**IMPORTANT:** All processes in comm must call this routine.

**REMARKS**

Many times the send buffer of the root process and the receive buffer of the other processes are referenced by the same name. You may want, at times, to use distinct names of the send buffer and the receive buffer, as illustrated below:

```
IF (myrank == 0) then
  call MPI_BCAST(sendbuf, ... )
ELSE
  call MPI_BCAST(recvbuf, ... )
ENDIF
```
Distribute the computation of the dot products to any available CPU. Each dot product is Independent of the other.

Suppose the master process (rank=0) "owns" A and b, and collects c = A * b, the result vector.

Master Process:
1) Broadcast b to each worker (rank > 0)
2) Using tag to signify row number, send a row to each worker
3) Loop: receive a dot product entry from a worker_i
   - send a new row to worker_i from whom we just received a dot product
   until all dot products received

Worker Process:
Receive b
Loop: receive a new row from master
   - compute dot product
   - send dot product to master
   until master sends termination notice to each task using tag=0

SELF SCHEDULING is called the work load distribution scheme whereby a "master" process hands work to an idle worker process.
MATRIX-VECTOR MULTIPLICATION

program matvec90 ! A*b = c; Uses Self-Scheduling
!
! Taken from Gropp's (& others) book "Using MPI", with minor mods.
!
!=====================================================================
!
USE MPI
!
integer myid, master, numprocs, ierr, status(MPI_STATUS_SIZE)
integer rows, cols, row, row_num, i, j, numsent, sender, workerid
real(kind=8), allocatable :: A(:,,:), b(:,), c(:,), c2(:,), row_buf(:)
real(kind=8) ans
!=====================================================================
!
call MPI_INIT( ierr )
call MPI_COMM_RANK( MPI_COMM_WORLD, myid, ierr )
call MPI_COMM_SIZE( MPI_COMM_WORLD, numprocs, ierr )
!
master = 0
!
if ( myid == master ) then
write(6,*), 'Enter array column size - '; read(5,*), cols
end if
!
MATRIX-VECTOR MULTIPLICATION

call MPI_BCAST(cols, 1, MPI_INTEGER, master, MPI_COMM_WORLD, ierr)
!
! Allocate space per process
allocate(row_buf(1:cols), b(1:cols))  ! for b & a row's worth
!
rows = cols
! We're dealing with sq mat's
!
if ( myid == master ) then
allocate( a(1:cols,1:cols), c(1:cols), c2(1:cols) )
call random_number(a); call random_number(b)  ! Fill in #'s
!
! send b to each slave process
call MPI_BCAST(b, cols, MPI_REAL8, master, MPI_COMM_WORLD, ierr)

numsent = 0
! Keep a count
!
! First, send one row to each slave process; tag with row #
!
do i = 1, min(numprocs-1,rows)  ! Avoid numprocs>rows case
  row_buf = a(i,1:cols)
  workerid = i
  call MPI_SEND(row_buf, cols, MPI_REAL8, workerid,&
                i, MPI_COMM_WORLD, ierr)
  numsent = numsent+1
end do
!
MATRIX-VECTOR MULTIPLICATION

do i = 1, rows
!
call MPI_RECV(ans, 1, MPI_REAL8, MPI_ANY_SOURCE, &
               MPI_ANY_TAG, MPI_COMM_WORLD, status, ierr
!
sender = status(MPI_SOURCE)  ! Id who just delivered a result
row_num = status(MPI_TAG)     ! row# is tag value
!
C(row_num) = ans
!
if (numsent < rows) then  ! then send another row
    row_buf = a(numsent+1,1:cols)
call MPI_SEND(row_buf, cols, MPI_REAL8, sender, &
              numsent+1, MPI_COMM_WORLD, ierr)
numsent = numsent+1
else  ! Tell sender that there is no more work
call MPI_SEND(MPI_BOTTOM, 0, MPI_REAL8, sender, &
              0, MPI_COMM_WORLD, ierr)
endif
!
end do
!
c2 = matmul(a,b)  ! F90 intrinsic: c2=A*b
write(6,fmt='(/" *** ",I4," x",I4, & ! Confirm Correctness
         " Matrix-Vector Multiplication ***",/)' ) cols,rows
write(6,fmt='(" -- MATMUL -- -- MATVEC90 --")' )
write(6,fmt='(4X,F9.4,6X,F9.4)') (c2(i), c(i), i=1, cols)
dallocate( a, c , c2)
else  
   ! **** slave section begins: slaves receive b, a row, compute dot_prods, send them
   ! to master until completion msg received by each of them
   !
   call MPI_BCAST(b, cols, MPI_REAL8, master,&
       MPI_COMM_WORLD, ierr)
   if (myid <= rows) then
      call MPI_RECV(row(buf), cols, MPI_REAL8, master,&
       MPI_ANY_TAG, MPI_COMM_WORLD, status, ierr)
      do while ( status(MPI_TAG) > 0 )
         row = status(MPI_TAG)
         ans = dot_product(row(buf), b)    ! F90 intrinsic func
      !
         call MPI_SEND(ans, 1, MPI_REAL8, master, row, &
              MPI_COMM_WORLD, ierr)
         call MPI_RECV(row(buf), cols, MPI_REAL8, master,&
              MPI_ANY_TAG, MPI_COMM_WORLD, status, ierr)
      end do
   end if
endif
deallocate(row(buf), b)
call MPI_FINALIZE(ierr)
stop
end
K2% f90 -o mpi_matvec.exe mpi_matvec.f90 -l mpi
K2% mpirun -np 6 mpi_matvec.exe
Enter array column size - 1000

*** 1000 x1000 Matrix-Vector Multiplication ***

-- MATMUL -- -- MATVEC90 --
246.5021   246.5021
257.0403   257.0403
256.4652   256.4652
242.5871   242.5871
253.9197   253.9197
247.6066   247.6066
248.4027   248.4027
244.4161   244.4161
...        ...
...        ...
249.9407   249.9407
236.3081   236.3081
249.2095   249.2095
248.8639   248.8639
245.2100   245.2100
245.5322   245.5322
248.4311   248.4311
254.1251   254.1251
MATRIX-MATRIX MULTIPLICATION

program matmat90  ! A*B = C; Uses Self-Scheduling
!=================================================================
! An extension of matvec90. This implementation of A*B aims to
! illustrate the basic practice of MPI. It is not at all intended
! as an efficient version of things.
!
! Spiros Vellas@tamu.edu
!
!=================================================================
!
USE MPI
!
integer myid, master, numprocs, ierr, status(MPI_STATUS_SIZE)
integer rows, cols, row, row_num, i, j, k, numsent, sender
real(kind=8), allocatable :: A(:,,:), B(:,,:), C(:,,:), C2(:,,:)
real(kind=8), allocatable :: ans(:,), row_buf(:)
!=================================================================
!
call MPI_INIT( ierr )
call MPI_COMM_RANK( MPI_COMM_WORLD, myid, ierr )
call MPI_COMM_SIZE( MPI_COMM_WORLD, numprocs, ierr )
!
master = 0
!
if ( myid == master ) then
    write(6,*),'Enter array column size - '; read(5,*) cols
end if
!
call MPI_BCAST(cols, 1, MPI_INTEGER, master, MPI_COMM_WORLD, ierr)
!
allocate(row_buf(1:cols), ans(1:cols), B(1:cols,1:cols))
rows = cols  ! We're dealing with sq mat's
MATRIX-MATRIX MULTIPLICATION

! if ( myid == master ) then
  allocate( A(1:cols,1:cols), C(1:rows,1:cols), C2(1:rows,1:cols) )
  call random_number(A); call random_number(B) ! Fill in rand #'s
!
! send B to each slave process
  call MPI_BCAST(B, rows*cols, MPI_REAL8, master, &
                MPI_COMM_WORLD, ierr)
  numsent = 0
!
! First, send one row to each slave process; tag with row #
  do i = 1, min(numprocs-1,rows)
    row_buf = A(i,1:cols) ! i-th row
    call MPI_SEND( row_buf, cols, MPI_REAL8, i, &
                   i, MPI_COMM_WORLD, ierr)
    numsent = numsent+1
  end do
!
  do i = 1, rows
    call MPI_RECV(ans, cols, MPI_REAL8, MPI_ANY_SOURCE, &
                  MPI_ANY_TAG, MPI_COMM_WORLD, status, ierr)
    sender = status(MPI_SOURCE)
    row_num = status(MPI_TAG) ! row# is tag value
    C( row_num, 1:cols ) = ans
  end do
!
  if (numsent < rows) then ! then send another row
    row_buf = A(numsent+1,1:cols)
    call MPI_SEND( row_buf, cols, MPI_REAL8, sender, &
                  numsent+1, MPI_COMM_WORLD, ierr)
    numsent = numsent+1
  else ! Tell sender that there is no more work
    call MPI_SEND(MPI_BOTTOM, 0, MPI_REAL8, sender, &
                  0, MPI_COMM_WORLD, ierr)
  endif
end do
MATRICES-MATRIX MULTIPLICATION

\[
C_2 = \text{matmul}(A,B) \text{ ! F90 intrinsic: } C_2 = A \times B
\]

Confirm correctness

\[
\text{write}(6, \text{fmt}='(/''***''\ I4\ '' \times\ '','I4, \\
& '' \ Matrix-Matrix\ Multiplication\ ***''),' )\ )\ cols,\rows
\]

\[
k = \text{cols}/2
\]

\[
\text{write}(6, \text{fmt}='(9X, ''\ >>\ \text{Listing\ only}\ ''\ I4, ''-kth\ Column<<''/ )\ )\ k
\]

\[
\text{write}(6, \text{fmt}='('\ --\ MATMUL\ --\ --\ MATMAT90\ --''),')
\]

\[
\text{write}(6, \text{fmt}=('\ 4X,F9.4, 6X,F9.4')\ )\ (C_2(i,k), C(i,k), i=1, \rows)
\]

else

**** slave section begins: slaves receive B, compute
 **** a row of C at a time, send them to master until
 **** completion msg received by each of them

\[
call\ \text{MPI\_BCAST}(B, \text{rows*cols}, \text{MPI\_REAL8}, \text{master,} &\ \\
\text{MPI\_COMM\_WORLD, ierr})
\]

\[
\text{if}\ (\text{myid} \leq \text{rows})\ \text{then}\ \\
\text{call}\ \text{MPI\_RECV}(\text{row\_buf, cols, MPI\_REAL8, master,} &\ \\
\text{MPI\_COMM\_WORLD, status, ierr})
\]

\[
do\ \text{while}\ (\ \text{status(MPI\_TAG)} > 0 )
\]

\[
\text{row} = \text{status(MPI\_TAG)}
\]

\[
do\ \text{j=1, rows} \text{ ! ans = C(row,1:cols)}
\]

\[
\text{ans(j)} = \text{dot\_product(row\_buf(1:cols), B(1:rows,j))}
\]

end do

\[
call\ \text{MPI\_SEND}(\text{ans, rows, MPI\_REAL8, master, row,} &\ \\
\text{MPI\_COMM\_WORLD, ierr})
\]

\[
call\ \text{MPI\_RECV}(\text{row\_buf, cols, MPI\_REAL8, master,} &\ \\
\text{MPI\_ANY\_TAG, MPI\_COMM\_WORLD, status, ierr})
\]

end do

end if

endif

!
call\ \text{MPI\_FINALIZE(ierr)}
stop
end
MATRIX–MATRIX MULTIPLICATION

K2% f90 -o mpi_matmat.exe mpi_matmat.f90 -l mpi
K2% mpirun -np 6 mpi_matmat.exe
   Enter array column size -
   1000

*** 1000 x 1000 Matrix–Matrix Multiplication ***
   >> Listing only 500-kth Column <<

-- MATMUL --   -- MATMAT90 --
  251.7982       251.7982
  258.7576       258.7576
  259.2238       259.2238
  245.7377       245.7377
  252.4468       252.4468
  255.0469       255.0469
  243.7969       243.7969
  251.2136       251.2136
   ...            ...
   ...            ...
  252.9531       252.9531
  241.1260       241.1260
  251.3991       251.3991
  252.0089       252.0089
  242.5807       242.5807
  250.4379       250.4379
  252.4090       252.4090
  258.2735       258.2735
GATHER

MPI_GATHER(inbuf, incnt, intype, outbuf, outcnt, outtype, root, comm, ierr)

inbuf        IN    address of send buffer
incnt        IN    number of elements that each process sends
intype       IN    MPI_Datatype of input buffer elements
outbuf       OUT   address of receive buffer
outcnt       IN    number of elements received from each process.
                  Significant at the root process only
outtype       IN    MPI_Datatype of receive buffer elements
root         IN    process id of root (receiving) process
comm         IN    communicator

When MPI_GATHER is activated, all processes in comm send data located in their inbuf to
root. Root places the arriving data from each process in rank order, and for each process
in contiguous locations in outbuf. The size and type of each message sent must be the same;
hence, size(outbuf) = np*size(inbuf). Also, intype and outtype must be the same. All
processes in comm must call this routine.
GATHER

Process Rank

0
inbuf
11
outbuf
11

1
inbuf
2
comm={0,1,2}

2
inbuf
114

11

2

114
GATHERV

MPI_GATHERV(inbuf, incnt, inctype, outbuf, outcnt, offsets, outtype, root, comm, ierr)

- **inbuf** `IN` address of send buffer
- **incnt** `IN` number of elements that each process sends
- **inctype** `IN` MPI_Datatype of send buffer elements
- **outbuf** `OUT` address of receive buffer
- **outcnt(*)** `IN` INTEGER array whose entries contain the size of the received message by the root from each process. The size of outcnt is equal to the number of processes in comm. This argument is significant only at root.
- **offsets(*)** `IN` INTEGER array whose i-th entry specifies the offset (relative to the start) into outbuf at which to place the incoming data from process i. The size of offsets is equal to the number of processes in comm. This argument is significant only to the root.
- **outtype** `IN` MPI_Datatype of receive buffer elements. This argument is significant only to the root.
- **root** `IN` process id of root process
- **comm** `IN` communicator

Gathers individual messages from each process in comm at the root process. When activated, all processes within comm, including root, send data located in inbuf to root. root receives and stores the data in outbuf, placing them in the order specified by the offsets array. Messages can have varying sizes based on the values in the outcnt(*) array. All processes in comm MUST call MPI_GATHERV.
GATHERV EXAMPLE

\( \text{comm} = \{0, 1, 2\} \)

<table>
<thead>
<tr>
<th>Process rank</th>
<th>0</th>
<th>1</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>inbuf</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>114</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>22</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>101</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

\(0 = \text{offsets}(0); 1 = \text{outcnt}(0)\)
\(1 = \text{offsets}(1); 2 = \text{outcnt}(1)\)
\(3 = \text{offsets}(2); 3 = \text{outcnt}(2)\)

outbuf

N.B. The specification of the offsets is entirely dependent on the discretion of the programmer.
Example Ex-gatherV

DESCRIPTIOM: Different processes compute/init different blocks of consecutive columns. In turn, these blocks (=alocals below) are consolidated into a complete array, $A$, at the master process. The master then calls the DGESV routine to solve the $Ax=b$ problem (-l scs needed to link the SCSL math lib). !

USE MPI

integer ierr, info, n, nrhs, rank
integer ncols, mysiz, np, L1, L2, master
integer, allocatable :: ipiv(:), recvcnt(:), offsets(:)
real (kind=8), allocatable :: A(:, :), alocal(:,:), b(:), x_lapack(:)

call MPI_INIT(ierr)
call MPI_COMM_RANK(MPI_COMM_WORLD, rank, ierr)
call MPI_COMM_SIZE(MPI_COMM_WORLD, np, ierr)
master = 0

if ( rank == master ) then
    write(6,*) 'Enter dimension size (<100 stops)- '; read(5,*) n
    if ( n < 100 ) stop
end if

call MPI_BCAST(n, 1, MPI_INTEGER, master, MPI_COMM_WORLD, ierr)
call BLOCK_MAP(1, n, np, rank, L1, L2) ! Block-Map columns

ALLOCATE( alocal(1:n, L1:L2), recvcnt(0:np-1), offsets(0:np-1) )
GATHERV (Program Ex_gatherv – 2)

do j=L1, L2                     ! Set tridiagonal locations; other zero
   alocal(1:n, j) = 0.0
   alocal(j,j) = 2.0     ! diag elems = -2.0; above & below it, -1.0
   if ( j>1 .and. j<n ) then
      alocal(j+1, j) = -1.0
      alocal(j-1, j) = -1.0
   elseif ( j == 1) then
      alocal(j+1,1) = -1.0
   elseif ( j == n) then
      alocal(j-1,n) = -1.0
   end if
end do

ncols = L2 - L1 + 1; mysiz = ncols*n
recvcnt(0:np-1) = 0; offsets(0:np-1) = 0

! ***Collect all xfer sizes at the master's recvcnt in rank order
!
call MPI_GATHER( mysiz, 1, MPI_INT, recvcnt, 1, MPI_INT, master, &
                          MPI_COMM_WORLD, ierr)

if ( rank == master ) then
   do j=1, np-1         ! Construct offsets array in rank-order
      do i=0, j-1       ! for use in gatherv
         offsets(j) = offsets(j) + recvcnt(i)
      end do
   end do
end if

if ( rank == master ) then
   ALLOCATE(A(1:n,1:n), x_lapack(1:n), b(1:n), ipiv(1:n) )
end if

!                       Consolidate into A, at master, all alocal's
!                                         ///
call MPI_GATHERV(alocal, mysiz, MPI_REAL8, A, recvcnt(rank), &
                      offsets(rank), MPI_REAL8, master, MPI_COMM_WORLD, ierr)
GATHERV (Program Ex_gatherv - 3)

if ( rank == master ) then  ! Master computes: Solve Ax=b
  b = 0.0; b(1) = 1.0; b(n) = 1.0  ! RHS: b = [ 1,0,0,...,0,0,1 ]
  x_lapack = b; nrhs = 1
  call DGESV(n, nrhs, A, n, ipiv, x_lapack, n, info) ! Lapack solver
  print *, " -- LAPACK's DGESV Solver Gives This Solution --"
  write(6, fmt='(6X,F9.6)') (x_lapack(i), i=1, n)
end if
!
call MPI_FINALIZE(ierr)
stop
End

! Run on cosmos

$ ifort -o ex_gather.exe ex_gather.f90 -lmpi -lscs

$ mpirun -np 4 ex_gather.exe
Enter dimension size (<100 stops)- 800

-- LAPACK's DGESV Solver Gives This Solution --
  1.000000
  1.000000
  1.000000
  1.000000
  1.000000
  1.000000
  :  
  :  
  1.000000
  1.000000
  1.000000
  1.000000
GATHERV

COMMENTS

In general terms, this example aims to illustrate the case where each process is assigned a part of the computation, and all such parts are consolidated into a large structure at a root process, which, in turn, performs a specialized task using this structure as input. In specific terms, the program assigns to each process the computation of a block of columns (alocal(1:n, L1:L2)), the consolidation of which into an array A, allocated at the root, is carried out by MPI_GATHERV. In this example, A is none other than the well known tridiagonal matrix with diagonal entries equal to 2 and bidiagonal ones equal to -1. At the root, A is now the coefficient matrix for solving $Ax=b$ via a call to LAPACK's linear solver, DGESV.

Carefully observe how the "block" assignment of columns to processes is closely tied with the use of the first MPI_GATHER in the formation of the recvcnt array at root, and the subsequent computation of the offsets array for rank-order placement of the alocal's into buffer A. All of these combine for the correct the formation of A.
**ALLGATHER**

MPI_ALLGATHER(sendbuf, sendcnt, sendtype, recvbuf, recvcnt, recvtype, comm, ierr)

sendbuf IN address of send buffer
sendcnt IN number of elements sent by each process in comm
sendtype IN MPI_Datatype of send buffer elements
recvbuf OUT address of receive buffer
recvctcnt IN number of elements to be received from each process in comm
recvtype IN MPI_Datatype of receive buffer elements
comm IN communicator

MPI_ALLGATHER is similar to MPI_GATHER except now all processes in comm receive the result instead of just the root. The amount of data gathered from each process is the same. In addition, the block of data sent by process j is received by each process (in comm) and placed in the j-th block of the buffer recvbuf. There must also be agreement between sendtype and recvtype. All processes in comm call this routine.
ALLGATHER

Comm={0,1,2}

Process rank

0

11

sendbuf

recvbuf

11

2

114

1

2

114

sendbuf

recvbuf

sendbuf

recvbuf

2

114

11

2

114

recvbuf

recvbuf

recvbuf
SCATTER

MPI_SCATTER(inbuf, incnt, intype, outbuf, outcnt, outtype, root, comm, ierr)

inbuf IN address of the send buffer (significant at root only)
incnt IN number of elements sent to each process
intype IN MPI_Datatype of the elements in inbuf
outbuf OUT address of receive buffer. inbuf & outbuf must not overlap
outcnt IN number of elements to be received
outtype IN MPI_Datatype of the elements in outbuf
root IN the rank of the sending process
comm IN communicator
ierr OUT Fortran return code

When MPI_SCATTER is activated, root sends the same amount (incnt) of data to each process in comm. The first contiguous incnt elements are sent to process 0, the next to process 1, etc. Each process receives data from root in its outbuf. Now we have, size(inbuf) = np*size (outbuf). All processes in comm must call this routine.
SCATTER

Process rank 0

| 33.3 |
| 11.1 |
| 0.97 |

inbuf

- 33.3
- 11.1
- 0.97

outbuf

comm = {proc_0, proc_1, proc_2}

Process rank 1

| 11.1 |

outbuf

Process rank 2

| 0.97 |

outbuf


SCATTERV

MPI_SCATTERV(inbuf, incnts, offsets, intype, outbuf, outcnt, outtype, root, comm, ierr)

inbuf       IN     address of the root's send buffer
incnts(*)   IN     integer array of size=group_size(comm) whose elements specify the size of
                   the data transfer to their corresponding process. Entry i specifies the size
                   of the data transfer to process i from inbuf.
                   SIGNIFICANT ONLY AT ROOT
offssets(*) IN     integer array of size=group_size(comm). Entry i specifies the offset into
                   inbuf from which to send the outgoing data to process i.
                   SIGNIFICANT ONLY AT ROOT
intype      IN     MPI_Datatype of the elements in inbuf.
                   SIGNIFICANT ONLY AT ROOT
outbuf      OUT    address of receive buffer. inbuf & outbuf must not overlap
outcnt      IN     number of elements to be received
outtype     IN     MPI_Datatype of the elements in outbuf
root        IN     the rank of the sending process
comm        IN     communicator
ierr        OUT    Fortran return code

Distributes individual messages from root to each process in comm. Messages can have different sizes and
offsets. The entries in the incnts array can be set to different values. The offsets array specifies where
the data can be taken from the root's inbuf. There must be pair-wise agreement between the amount and type of
the data sent by root on the one hand, and that received by each process on the other. All processes in comm
must call this routine.
SCATTERV

Process rank = 0

<table>
<thead>
<tr>
<th>33.3</th>
</tr>
</thead>
<tbody>
<tr>
<td>11.1</td>
</tr>
<tr>
<td>0.97</td>
</tr>
<tr>
<td>0.00</td>
</tr>
<tr>
<td>-22.03</td>
</tr>
</tbody>
</table>

Offsets(0) = 0
Incnts(0) = 1

Offsets(1) = 1
Incnts(1) = 2

Offsets(2) = 3
Incnts(2) = 2

Process rank = 1

<table>
<thead>
<tr>
<th>33.3</th>
</tr>
</thead>
</table>

Process rank = 2

<table>
<thead>
<tr>
<th>11.1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.97</td>
</tr>
<tr>
<td>0.00</td>
</tr>
<tr>
<td>-22.03</td>
</tr>
</tbody>
</table>

inbuf

outbuf

outbuf
PROGRAM SCATTERV_SAMPLE ! Matches diagram
USE MPI
INTEGER np, myid, ierr
INTEGER RCNT, INCNTS(0:2), OFFSETS(0:2)
DATA INCNTS/1, 2, 2/, OFFSETS/0, 1, 3/
REAL (KIND=8) INBUF(5), OUTBUF(2)
DATA INBUF/33.3, 11.1, 0.97, 0.00, -22.03/
!
call MPI_INIT(ierr)
call MPI_COMM_RANK(MPI_COMM_WORLD, myid, ierr)
call MPI_COMM_SIZE(MPI_COMM_WORLD, np, ierr)
!
RCNT = 1+MYID; IF (MYID == 2) RCNT = RCNT-1
OUTBUF = 0.0 ! Initialize for each proc
!
call MPI_SCATTERV(INBUF, INCNTS, OFFSETS, &
  MPI_DOUBLE_PRECISION, OUTBUF, RCNT, &
  MPI_DOUBLE_PRECISION, 0, MPI_COMM_WORLD, ierr)
!
PRINT '(I1,X,2(F6.2, X))', MYID, OUTBUF
call MPI_FINALIZE(ierr)
STOP
END

---------------------------------------------------
> f90 -o mpi_scatterv.exe mpi_scatterv.f90 -lmpl
> mpirun -np 3 mpi_scatterv.exe

0  33.30   0.00
1  11.10   0.97
2   0.00 -22.03

Note that the 0.00 value listing under the rank=0 process is not one that was put there as a result of the scatter process, but was generated by the initialization statement, OUTBUF = 0.0, and was executed by each process.

This example is meant to be run on three processor only.
REDUCE/ALLREDUCE

MPI_REDUCE(inbuf, outbuf, count, type, op, root, comm, ierr)
MPI_ALLREDUCE(inbuf, outbuf, count, type, op, comm, ierr)

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>inbuf</td>
<td>IN</td>
<td>address of send buffer</td>
</tr>
<tr>
<td>outbuf</td>
<td>OUT</td>
<td>address of receive buffer. inbuf and outbuf must not overlap at root</td>
</tr>
<tr>
<td>count</td>
<td>IN</td>
<td>number of intype elements in inbuf</td>
</tr>
<tr>
<td>type</td>
<td>IN</td>
<td>the MPI_Datatype of send buffer</td>
</tr>
<tr>
<td>op</td>
<td>IN</td>
<td>operation (e.g., +,-,*, etc)</td>
</tr>
<tr>
<td>root</td>
<td>IN</td>
<td>process id of root process</td>
</tr>
<tr>
<td>comm</td>
<td>IN</td>
<td>communicator</td>
</tr>
</tbody>
</table>

The MPI_REDUCE applies a reduction operation to a vector of elements in the inbuf of each process in comm and places the result in outbuf, the receive buffer, of the root process. The size and type of inbuf and outbuf must be the same. If inbuf for each process contains a vector of elements, then the reduction operation is carried out element-wise.

The MPI_ALLREDUCE is similar to MPI_REDUCE except the result is received by all processes in comm. No root process here.
## REDUCE / ALLREDUCE

### PREDEFINED OPERATIONS

<table>
<thead>
<tr>
<th>Operation 1</th>
<th>Operation 2</th>
<th>Data Types</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_SUM,</td>
<td>MPI_PROD</td>
<td>MPI_REAL, MPI_INTEGER, MPI_DOUBLE_PRECISION, MPI_COMPLEX</td>
</tr>
<tr>
<td>MPI_MAX,</td>
<td>MPI_MIN</td>
<td>MPI_INTEGER, MPI_REAL, MPI_DOUBLE_PRECISION</td>
</tr>
<tr>
<td>MPI_MAXLOC,</td>
<td>MPI_MINLOC</td>
<td>MPI_2REAL, MPI_2INTEGER, MPI_2DOUBLE_PRECISION</td>
</tr>
<tr>
<td>MPI_LAND,</td>
<td>MPI_LOR, MPI_LXOR</td>
<td>MPI_LOGICAL</td>
</tr>
<tr>
<td>MPI_BAND,</td>
<td>MPI_BOR, MPI_BXOR</td>
<td>MPI_INTEGER, MPI_BYTE</td>
</tr>
</tbody>
</table>
### A REDUCE EXAMPLE FOR ARRAYS

\[ \text{comm} = \{0, 1, 2\} \]

<table>
<thead>
<tr>
<th>Process rank</th>
<th>0</th>
<th>1</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>inbuf</td>
<td>11</td>
<td>2</td>
<td>114</td>
</tr>
<tr>
<td></td>
<td>7</td>
<td>3</td>
<td>22</td>
</tr>
<tr>
<td></td>
<td>10</td>
<td>1</td>
<td>101</td>
</tr>
</tbody>
</table>

\[ \text{Op} = + \]

\[
\begin{align*}
11 + 2 + 114 &= 127 \\
7 + 3 + 22 &= 32 \\
10 + 1 + 101 &= 112
\end{align*}
\]

\[ \text{outbuf} \]

\[ \text{N.B. size(inbuf)} &= \text{size(outbuf)} \]
AN ALLREDUCE EXAMPLE FOR ARRAYS

comm = \{0,1,2\}

\begin{align*}
\text{Process rank 0} & & \text{Process rank 1} & & \text{Process rank 2} \\
\text{inbuf} && \text{inbuf} && \text{inbuf} \\
\begin{array}{c}
11 \\
7 \\
10 \\
\end{array} & & \\
\begin{array}{c}
2 \\
3 \\
1 \\
\end{array} & & \\
\begin{array}{c}
114 \\
22 \\
101 \\
\end{array} & & \\
\end{align*}

\begin{align*}
\text{outbuf} & & \text{outbuf} & & \text{outbuf} \\
\begin{array}{c}
11 + 2 + 114 = 127 \\
7 + 3 + 22 = 32 \\
10 + 1 + 101 = 112 \\
\end{array} & & \\
\begin{array}{c}
11 + 2 + 114 = 127 \\
7 + 3 + 22 = 32 \\
10 + 1 + 101 = 112 \\
\end{array} & & \\
\begin{array}{c}
11 + 2 + 114 = 127 \\
7 + 3 + 22 = 32 \\
10 + 1 + 101 = 112 \\
\end{array} & & \\
\end{align*}

N.B. \(\text{size}(\text{inbuf}) = \text{size}(\text{outbuf})\)
MPI_OP_CREATE

MPI_OP_CREATE(func, commute, op, ierr)

func IN EXTERNAL, user-defined reduction function

commute IN LOGICAL. Specifies whether the function is commutative. Value is true if commutative; false otherwise

op OUT the reduction operation handle

ierr OUT specifies the Fortran return code value for successful completion

Binds a user-defined reduction operation to an op handle which you can then use in MPI_REDUCE, MPI_ALLREDUCE, MPI_REDUCE_SCATTER, and MPI_SCAN, and their non-blocking equivalents. The user-defined operation is assumed to be associative. If commute == .TRUE., then the operation must be both commutative and associative. If commute == .FALSE., then the order of the operation is fixed. The order is defined in ascending process rank order and begin with process zero. The user-defined function, func, must have the following argument interface:

SUBROUTINE func(invec(*), inoutvec(*), len, type)
<type> invec(len), inoutvec(len)
INTEGER len, type
 :
MPI_OP_CREATE

comm = {0, 1, 2}

Process rank 0

\[
\begin{array}{c}
(1,1) \\
(2,2)
\end{array}
\]

\[\text{inbuf} \rightarrow + \rightarrow \text{outbuf} \]

\[
\begin{array}{c}
(33,33) \\
(36,36)
\end{array}
\]

Process rank 1

\[
\begin{array}{c}
(11,11) \\
(12,12)
\end{array}
\]

\[\text{inbuf} \rightarrow + \rightarrow \text{outbuf} \]

Process rank 2

\[
\begin{array}{c}
(21,21) \\
(22,22)
\end{array}
\]

\[\text{inbuf} \rightarrow + \rightarrow \text{outbuf} \]

\[\text{Op} = + \] ------ add double precision complex numbers
MPI_OP_CREATE (Program Opcreate – 1)

PROGRAM OPCODECREATE ! Taken from ref. [5] & modified a bit
! Illustrates sum reduction with complex #'s
! which is not included in the default ops
USE MPI
EXTERNAL cmplx_sum
INTEGER MYRANK, NUMPROCS, IERR, CSUM_OP
COMPLEX (KIND=8) C(2), CSUM(2)
!
call MPI_INIT( ierr )
call MPI_COMM_RANK( MPI_COMM_WORLD, myrank, ierr )
!
call MPI_OP_CREATE(cmplx_sum, .TRUE., csum_op, ierr)
!
C(1) = CMPLX(myrank*10 + 1, myrank*10 + 1, KIND=8)
C(2) = CMPLX(myrank*10 + 2, myrank*10 + 2, KIND=8)
!
call MPI_REDUCE(C, CSUM, 2, MPI_DOUBLE_COMPLEX, &
                csum_op, 0, MPI_COMM_WORLD, ierr)
IF (MYRANK == 0) PRINT *, 'CSUM =', CSUM
!
call MPI_FINALIZE(ierr)
END

SUBROUTINE cmplx_sum(CIN, CINOUT, LEN, ITYPE)
COMPLEX (KIND=8) CIN(*), CINOUT(*)
DO I=1, LEN
    CINOUT(I) = CINOUT(I) + CIN(I)
END DO
END

-----------------------------------------------------
> f90 -o mpi_opcreate.exe mpi_opcreate.f90 -lmpi
> mpirun -np 3 mpi_opcreate.exe
  CSUM = (33.,33.), (36.,36.)
Program Mpi_Compute_Pi

PROGRAM MPI_COMPUTE_PI
! Computes pi using trapezoidal rule. Slightly modified
! version of Gropp's USING MPI book
USE MPI
REAL (KIND=8), PARAMETER :: PI_MACH = 3.141592653589793238462643
REAL (KIND=8) mypi, pi, h, sum, x, f, a
INTEGER n, myid, numprocs, i, ierr
f(a) = 4.0 / (1.0 + a*a) ! function to integrate
!
call MPI_INIT( ierr )
call MPI_COMM_RANK( MPI_COMM_WORLD, myid, ierr )
call MPI_COMM_SIZE( MPI_COMM_WORLD, numprocs, ierr )
!
do while ( 1 == 1 )
  if ( myid .eq. 0 ) then
    write(6, &
   fmt='("Enter the number of intervals: (0 quits)" )' )
    read(5,fmt='(i10)') n
  endif
  call MPI_BCAST(n,1,MPI_INTEGER,0,MPI_COMM_WORLD,ierr)
  if ( n .le. 1 ) exit ! Exit loop/terminate

  h = 1.0/n ! integration interval increment
  sum = 0.0

  do i = myid+1, n, numprocs ! Cyclic Distribution
    x = h * (dble(i) - 0.5)
    sum = sum + f(x)
  end do
  mypi = h * sum ! Contribution from myid
!
  collect all the partial sums
  call MPI_REDUCE(mypi, pi, 1, MPI_DOUBLE_PRECISION, &
                 MPI_SUM, 0, MPI_COMM_WORLD,ierr)
  if (myid .eq. 0) print *,"pi =",pi," Error =",abs(pi-PI_MACH)
end do
Program Mpi_Compute_Pi - output

! call MPI_FINALIZE(ierr) stop end

Øf90 -o mpi_pi.exe mpi_pi.f90 -lmpi

> mpirun -np 6 mpi_pi.exe

Enter the number of intervals: (0 quits) 100
pi ~= 3.141600942219454  Error = 8.2012068807202354E-6

Enter the number of intervals: (0 quits) 1000
pi ~= 3.1415928319180302  Error = 9.09054569397937939E-8

Enter the number of intervals: (0 quits) 0
Program mpi_scatterv2-1

!---------------------------------------------------------------------!
! Programmer: Spiros Vellas@Texas A&M
!
! Illustrates use of MPI_SCATTERV in carrying matrix-vector multiplication. Strips of consecutive columns of matrix A are distributed to all processes. Each of these carries out a part of the linear vector sum:

\[ A*b = b_1*a_1 + b_2*a_2 + \ldots + b_n*a_n = [ c_1, c_2, \ldots, c_n ]^T \]

where the a's are the columns of \( A = [ a_1, a_2, \ldots, a_n ] \) and the b's are the (scalar) components of the b column vector

\[ b = [ b_1, b_2, \ldots, b_n ]^T \]

!---------------------------------------------------------------------!

Program mpi_scatterv2
USE MPI
integer id, ierr, n, rank, ncols, np, L1, L2, root
integer, allocatable :: sendcnt(:), offsets(:)
logical convergent
real (kind=8), allocatable :: A(:, :), alocal(:, :)
real (kind=8), allocatable :: b(:), ab(:), c(:), c2(:)
real (kind=8) tol, e_tm
!--------------------------------------------------------------
call MPI_INIT(ierr)
call MPI_COMM_RANK(MPI_COMM_WORLD, rank, ierr)
call MPI_COMM_SIZE(MPI_COMM_WORLD, np, ierr)
!
root = 0
!
if ( rank == root ) then
    write(6,*) 'Enter dimension size - '; read(5,*) n
    ALLOCATE( A(1:n,1:n), b(1:n), c(1:n), c2(1:n), sendcnt(0:np-1), &
              offsets(0:np-1) )
    call random_number(A); call random_number(b) ! Fill them with rands
!
do id=0, np-1
    call BLOCK_MAP(1, n, np, id, L1, L2) ! Block-Map n cols across np procs
    sendcnt(id) = ( L2 - L1 + 1 ) * n    ! Space for n-element cols for proc id
    if ( id == 0 ) then
        offsets(0) = 0
    else
        offsets(id) = offsets(id-1) + sendcnt(id-1)
    end if
end do
endif
Program mpi_scatterv2 - 3

call MPI_BCAST(n, 1, MPI_INTEGER, root, MPI_COMM_WORLD, ierr)
If ( rank /= root ) ALLOCATE( b(1:n) )
call MPI_BCAST(b, n, MPI_REAL8, root, MPI_COMM_WORLD, ierr)
!
call BLOCK_MAP(1, n, np, rank, L1, L2) ! Block-Map iters
ncols = L2 - L1 + 1
!
ALLOCATE( alocal(1:n, 1:ncols), ab(1:n) ); alocal = 0.0
!
call mpi_scatterv(A, sendcnt, offsets, MPI_REAL8, alocal, &
   n*ncols, MPI_REAL8, root, MPI_COMM_WORLD, ierr)
ab = 0.0
do i=L1, L2
   ab = ab + b(i) * alocal(1:n, i-L1+1)
end do
!
call MPI_REDUCE(ab, c, n, MPI_REAL8, MPI_SUM, root, MPI_COMM_WORLD, ierr)
!
if ( rank == root ) then
   c2 = matmul(A,b) ! F90 intrinsic: c2=A*b
   Confirm correctness
   write(6,fmt='(/" *** ",I4," x",I4, &
      " Matrix-Vector Multiplication ***",/)' ) n,n
   write(6,fmt='(" -- MATMUL -- -- SCATTERV2 --")')
   write(6,fmt=('(4X,F9.4,6X,F9.4)') (c2(i), c(i), i=1, n)
end if
!
call MPI_FINALIZE(ierr)
stop
end
Program mpi_scatterv2-4

```bash
$ f90 -o mpi_scatterv2.exe mpi_scatterv2.f90 -l mpi

$ mpirun -np 7 mpi_scatterv2.exe
Enter dimension size - 1004

*** 1004 x1004 Matrix-Vector Multiplication ***

```

<table>
<thead>
<tr>
<th>MATMUL</th>
<th>SCATTERV2</th>
</tr>
</thead>
<tbody>
<tr>
<td>263.4424</td>
<td>263.4424</td>
</tr>
<tr>
<td>260.9878</td>
<td>260.9878</td>
</tr>
<tr>
<td>255.7887</td>
<td>255.7887</td>
</tr>
<tr>
<td>255.8765</td>
<td>255.8765</td>
</tr>
<tr>
<td>252.9694</td>
<td>252.9694</td>
</tr>
<tr>
<td>256.2698</td>
<td>256.2698</td>
</tr>
<tr>
<td>260.4828</td>
<td>260.4828</td>
</tr>
<tr>
<td>255.9019</td>
<td>255.9019</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>MATMUL</th>
<th>SCATTERV2</th>
</tr>
</thead>
<tbody>
<tr>
<td>256.7112</td>
<td>256.7112</td>
</tr>
<tr>
<td>256.7890</td>
<td>256.7890</td>
</tr>
<tr>
<td>250.4080</td>
<td>250.4080</td>
</tr>
<tr>
<td>263.3795</td>
<td>263.3795</td>
</tr>
<tr>
<td>259.0245</td>
<td>259.0245</td>
</tr>
<tr>
<td>256.7393</td>
<td>256.7393</td>
</tr>
<tr>
<td>256.7123</td>
<td>256.7123</td>
</tr>
<tr>
<td>254.9661</td>
<td>254.9661</td>
</tr>
</tbody>
</table>
Program Allreduce - 1

program Allreduce
    use mpi
    INTEGER, allocatable :: sendbuf(:), recvbuf(:)
    INTEGER myrank, numprocs, i, ierr
    CHARACTER (LEN=2) itoa*2, FN*20, fmtstr*16
!
    call MPI_INIT( ierr )
    call MPI_COMM_RANK( MPI_COMM_WORLD, myrank, ierr )
    call MPI_COMM_SIZE( MPI_COMM_WORLD, numprocs, ierr )
!
    FN = 'mpi_allreduce'//CHAR(48+myrank)//'.dat'
    OPEN(UNIT=myrank+10,FILE=FN,FORM='formatted',STATUS='unknown')
    allocate(sendbuf(0:numprocs-1), recvbuf(0:numprocs-1))
!
    do i=0, numprocs-1
            sendbuf(i) = i * (myrank+1)
    end do
!
    size(sendbuf) = numprocs
    call MPI_ALLREDUCE(sendbuf, recvbuf, numprocs, MPI_INTEGER, &
                       MPI_SUM, MPI_COMM_WORLD, ierr)
    fmtstr = '('//ITOA(numprocs)//'(I3,X,/))'
    write(unit=myrank+10, fmt=fmtstr) recvbuf
    deallocate( sendbuf, recvbuf)
    call MPI_FINALIZE(ierr)
    stop
end

> f90 -o mpi_allreduce.exe mpi_allreduce.f90 -lmmpi
> mpirun -np 4 mpi_allreduce.exe
> cat mpi_allreduce[0-3].dat
  0
  10
  20
  30
  0
  10a
  20
  30
EXAMPLE: SOLVING the x-y POISSON EQUATION

\[
\frac{d^2u}{dx^2} + \frac{d^2u}{dy^2} = f(x,y) \quad u(x,y) = g(x,y) \text{ on boundary}
\]

x, y in [0,1]

5-point finite difference stencil approximation

\[
u_{i-1,j} + u_{i,j+1} + u_{i,j-1} + u_{i+1,j} - 4u_{i,j} = h^2 f_{i,j}
\]

Note: here i and j correspond to (Spatial) coordinates x and y.
EXAMPLE: SOLVING the x-y POISSON EQUATION

5-point finite difference stencil approximation

$k+1$ Jacobi iteration step at $x_i = i \cdot h; \ y_j = j \cdot h; \ i, j = 1: n$

$$u_{k+1}^{i,j} = \frac{1}{4} (u_{k}^{i-1,j} + u_{k}^{i,j+1} + u_{k}^{i,j-1} + u_{k}^{i+1,j} - h^2 \cdot f_{i,j})$$

Boundary Conditions (Stay fixed):

$u(x_i, 0) = \sin(\pi \cdot x_i)$
$u(x_i, 1) = \sin(\pi \cdot x_i) \cdot \exp(x_i) \quad i = 0: n + 1$
$u(0, y_j) = 0.0 \quad j = 0: n + 1$
$u(1, y_j) = 0.0$

Jacobi iteration across all points:

```plaintext
do j=1, n
    do i=1, n
        unew(i, j) = 0.25*( (u(i-1, j)+u(i, j+1)+u(i, j-1)+u(i+1, j)) - &
                          f(i, j)*h^2 )
    end do
end do
```
EXAMPLE: SOLVING the x-y POISSON EQUATION (Fortran)

program s_jacobi1d  ! SERIAL VERSION
************************************************************************
! s_jacobi1d.f - a serial solution to the 2-d Poisson problem is effected
! using the Jacobi iteration with a 5-point finite difference stencil.
!
! Problem: --- + --- = f(x,y)  x,y in [ 0, 1 ]
!           dx   dy
!
! Programmer: Spiros Vellas
!
! The Jacobi iteration is run until the change in successive
! elements is < eps or a maximum number of iterations is reached.
************************************************************************

integer, parameter :: maxn = 1000
real (kind=8)  u(0:maxn+1,0:maxn+1), unew(0:maxn+1,0:maxn+1), &
                f(maxn,maxn)
integer n, iter, max_iter
real (kind=8) diff, diffnorm

print *, 'Enter Problem (Square Mesh) Size    - '; read *, n
if ( n < 100 .OR. n > maxn ) STOP
print *, "Enter Maximum Number of Iterations - "; read *, max_iter
program s_jacobi1d

if ( max_iter < 1 ) STOP !
call init2d_domain( u, unew, f, n ) ! Init the rhs, f(x), of Poisson eq'n
! and set bdry & the initial solution guess
eps = 1.0d-5; diffnorm = 1.0d+2; iter = 0 !
do while ( iter < max_iter .and. diffnorm > eps ) !
    call sweep2d_domain( u, f, n, unew )
call sweep2d_domain( unew, f, n, u ) !
    diffnorm= diff( u, unew, n )
    iter = iter + 1 end do
if ( diffnorm < eps ) then
  print *, 'CONVERGENCE: ', iter, ' Iters '
  open (unit=9, file='s_jacobi.dat', form='unformatted', status='UNKNOWN')
  write(9) u(0:n+1,0:n+1)
else
  print *, 'NO CONVERGENCE: ', iter endif !
stop end

subroutine init2d_domain( u, unew, f, n )
integer n, i, j
real (kind=8) u(0:n+1, 0:n+1), unew(0:n+1, 0:n+1),
          f(0:n+1, 0:n+1)
real (kind=8) h, pi, sin_hx, exp_hx !
h = 1.0d0/(n + 1); pi = 3.1415926535897932d0 !
do j=0, n+1
  do i=0, n+1
    u(i,j) = 0.0d0
    unew(i,j) = 0.0d0
    f(i,j) = 0.0d0
  end do
  sin_hx = sin(pi * h*i)
  u(i,j) = sin_hx ! x-axis, y=0: sin(pi*x)
  unew(i,j) = sin_hx
  exp_hx = exp( h*i)
  u(i,nx+1) = sin_hx * exp_hx ! x-axis, y=1
  unew(i,nx+1) = sin_hx * exp_hx
end do
return end
program s_jacobi1d

!----------------------------------------------
! Does one full Jacobi iteration sweep across the
! Whole square 2-d mesh
!----------------------------------------------
integer n, i, j
real (kind=8) u(0:n+1,0:n+1), unew(0:n+1,0:n+1), f(0:n+1,0:n+1)
real (kind=8) h
!
  h = 1.0d0 / (n+1)
do j=1, n
do i=1, n
    unew(i,j) = 0.25 * (&
      ( u(i-1,j)+u(i,j+1)+u(i,j-1)+u(i+1,j) ) - &
      h * h * f(i,j) )
endo
doendo
return
end
!

real (kind=8) function diff( u, unew, n )
integer n
real (kind=8) u(0:n+1,0:n+1), unew(0:n+1,0:n+1)
real (kind=8) sum
integer i, j
!
  sum = 0.0d0
do j=1,n
do i=1,n
    sum = sum + (u(i,j) - unew(i,j)) ** 2
endo
doendo
!
diff = sum
return
end
EXAMPLE: SOLVING the x-y POISSON EQUATION (in C)

/*******************************************************************************/
*  s_jacob1d.c - a serial solution to the 2-d Poisson problem is
*  effected using the Jacobi iteration.
*
*  _ 2  _ 2
*  d u  d u
*  Problem: --- + --- = f(x,y)  x,y in [ 0, 1 ]
!  _ 2  _ 2
*  dx  dy
*
*  Programmer: Nagesh, Texas A&M University
*
*  The Jacobi iteration is run until the change in successive
*  elements is < eps or a maximum number of iterations is reached.
*******************************************************************************/

#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#define MAXN 1000

void init2d_domain (double u[][MAXN], double unew[][MAXN], double f[][MAXN], int n);
void sweep2d_domain (double u[][MAXN], double f[][MAXN], int n, double unew[][MAXN]);
double diff_U (double u[][MAXN], double unew[][MAXN], int n);

int main() {

double u[MAXN][MAXN], unew[MAXN][MAXN], f[MAXN][MAXN];
double diff, diffnorm, eps;
int iter, max_iter, i, j, n;
FILE *fp;

printf ("Enter problem (Square mesh) size: ");
scanf ("%d", &n);

if (n < 100 || n > MAXN) {
    printf ("Specify n between 100 & 500\n");
    exit(1);
}

printf ("Enter max. number of Iterations: ");
scanf ("%d", &max_iter);
if (max_iter < 1) {
    printf ("Enter max. #iterations > 1\n");
    exit(1);
}

while (iter < max_iter & & diffnorm > eps) {
    sweep2d_domain (u, f, n, unew);
    sweep2d_domain (unew, f, n, u);
    diffnorm = diff_U (u, unew, n);
    ++iter;
}

if (diffnorm < eps) {
    printf ("Convergence: %d Iterations\n", iter);
    fp = fopen ("s_jacobi.dat", "w+");
    for (i=0; i <= n+1; i++)
        for (j=0; j <= n+1; j++)
            fprintf (fp, "%lf", U[i][j]);
    fclose (fp);
}

return 1;
void init2d_domain (double u[][MAXN], double new[][MAXN], double f[][MAXN], int n) {
    int i, j;
    double h = 1.0 / (n+1);
    double pi = 3.141592535897932;
    double sin_hx, exp_hx;

    for (i=0; i <= n+1; i++)
        for (j=0; j <= n+1; j++) {
            u[i][j] = 0.0;
            new[i][j] = 0.0;
            f[i][j] = 0.0;
        }

    /* Boundary Conditions */
    for (i=0; i <= n+1; i++)
        for (j=0; j <= n+1; j++) {
            u[i][j] = 0.0;
            new[i][j] = 0.0;
            f[i][j] = 0.0;
        }

    /* Boundary Conditions */
    for (i=0; i <= n+1; i++)
        for (j=0; j <= n+1; j++) {
            u[i][j] = 0.0;
            new[i][j] = 0.0;
            f[i][j] = 0.0;
        }

    double diff_U (double u[][MAXN], double new[][MAXN], int n) {
        int i,j;
        double sum = 0.0;

        for (i=1; i <= n; i++)
            for (j=1; j <= n; j++)
                sum += pow ((u[i][j] - new[i][j]), 2);

        return sum;
    }

void sweep2d_domain (double u[][MAXN], double f[][MAXN], int n, double new[][MAXN]) {
    /* Does one full jacobi iteration sweep across the whole square 2D mesh */
    int i, j;
    double h = 1.0/(n+1);

    for (i=1; i <= n; i++)
        for (j=1; j <= n; j++)
            new[i][j] = 0.25 *
            ( (u[i-1][j]+u[i][j+1]+u[i][j-1]+u[i+1][j])
                - h*h*f[i][j] );

    double diff_U (double u[][MAXN], double new[][MAXN], int n) {
    int i,j;
    double sum = 0.0;

    for (i=0; i <= n; i++)
        for (j=0; j <= n; j++)
            sum += pow ((u[i][j] - new[i][j]), 2);

    return sum;
}
COLUMN-WISE BLOCK DISTRIBUTION: POISON PROBLEM

** PARALLEL CASE 1 **

Showing border columns that will have to be copied into the address space of a neighbor process during a (5-point) finite-difference iteration scheme (jacobi) used in the p_jacobild program to solve the x-y Poisson equation.
COLUMN-WISE BLOCK DISTRIBUTION: POISSON PROBLEM
Column Exchanges Needed for Parallel 5-point FDM Jacobi Iteration

Showing exchanges of border columns per process. A border column from one process is copied into a "ghost" column of a neighbor's address space.
Benchmarking p_jacobild_v1.exe

> f90 -o p_jacobild_v1.exe p_jacobild_v1.f90 -lmpi

> mpirun -np 2 p_jacobild_v1.exe

Enter Problem (Square Mesh) Size   - 100
Enter Maximum Number of Iterations - 120000
CONVERGENCE:  Iters= 65326  Seconds= 73.14

> mpirun -np 4 p_jacobild_v1.exe

Enter Problem (Square Mesh) Size   - 100
Enter Maximum Number of Iterations - 120000
CONVERGENCE:  Iters= 65326  Seconds= 39.00

> mpirun -np 6 p_jacobild_v1.exe

Enter Problem (Square Mesh) Size   - 100
Enter Maximum Number of Iterations - 120000
CONVERGENCE:  Iters= 65326  Seconds= 28.09

> mpirun -np 8 p_jacobild_v1.exe

Enter Problem (Square Mesh) Size   - 100
Enter Maximum Number of Iterations - 120000
CONVERGENCE:  Iters= 65326  Seconds= 22.14
EXAMPLE: SOLVING the x-y POISSON EQUATION IN PARALLEL (MPI - C)

/****************************************************************************
* jacobi_1d.c *************************************************************/
* Computes a solution to the two dimensional Poisson problem using
* Jacobi iteration on a 1-d domain decomposition. The domain de-
* composition is carried out along the y-axis, i.e., along the
* rows of arrays u and unew.
* __ 2 __ 2
* d u d u
* x,y in [ 0, 1 ]
* Problem: --- + --- = f(x,y) = (x^2 + y^2) * cos(x*y) - cos(pi * x)
* __ 2 __ 2
* dx dy Boundary Conditions
* y = 0: u(x, 0)=\(\frac{\cos(\pi x) - \pi \pi}{\pi \pi}\)
* y = 1: u(x, nx +1) = \(\frac{\cos(\pi x) - \pi \pi \cos(x)}{\pi \pi}\)
* x = 0: u(0,y) = 1/(\pi \pi) - 1
* x = 1: u(1,y) = -(1/(\pi \pi) + \cos(y))
* Programmer: Julian Panetta
* The following source file is a modification of source files,
* at the intermediate level, that accompany the Using MPI textbook
* (ref. 2) by Gropp and others.
* The Laplacian and the boundary conditions used in this problem
* were taken from section 12.1 exercise 3f of the fifth edition
* of Numerical Analysis by Burden and Faires.
* The size of the domain is read by processor 0 and broadcast to
* all other processors. The Jacobi iteration is run until the
* change in successive elements is small or a maximum number of
* iterations is reached. The results from each processor are
* output to separate files in the format output[#].txt where
* # is the cpu's rank.
****************************************************************************/

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

void **alloc2d(int nRows, int nColumns, int size);
void printRow(double *uRow, int nColumns, FILE *aFile);
void block_map(int n1, int n2, int numProcs, int myId, int *s, int *e);
void init2d_domain(double **u, double **unew, double **f, int nx, int ny, int s, int e);
void exhcn1d(double **u, int nx, int s, int e, MPI_Comm comm1d, int aboveId, int belowId);
void sweep1d(double **u, double **f, int n, int s, int e, double **unew);
double diff(double **u, double **f, int n, int s, int e, double **unew);

int main(int argc, char *argv[])
{
  double **u, **unew, **f;
  int myid, aboveid, belowid, numprocs, my_num_rows, nx, ny, s, e, maxIter, iter;
  double eps = 1.0e-5, diffNorm, mydiff, t1, t2;

  MPI_Init(&argc, &argv);
  MPI_Comm_size(MPI_COMM_WORLD, &numprocs);
  MPI_Comm_rank(MPI_COMM_WORLD, &myid);

  if (myid == 0) {
    do {
      printf("Enter problem (Square mesh) size:\n");
      scanf("%d", &nx);
      if (nx < 100) printf("--You must enter a size greater than 100.--\n");
    } while (nx < 100);

    do {
      printf("Enter maximum number of iterations:\n");
      scanf("%d", &maxIter);
      if (maxIter < 1) printf("--You must enter a positive number of iterations!--\n");
    } while (maxIter < 1);
  }
}
MPI_Barrier(MPI_COMM_WORLD);

t1 = MPI_Wtime();
MPI_Bcast(&nx, 1, MPI_INT, 0, MPI_COMM_WORLD);
MPI_Bcast(&maxIter, 1, MPI_INT, 0, MPI_COMM_WORLD);
ny = nx;
block_map(1, nx, numprocs, myid, &s, &e);
/*******************************************************************************/
/* Allocate only the chunk of the array needed to hold the processor's */
/* points and two ghost rows for the next/previous processor (or */
/* boundary row vectors in the case of the top/bottom processors) */
*******************************************************************************/
my_num_rows = e - s + 1;
u = (double **)alloc2d(my_num_rows + 2, ny + 2, sizeof(double));
unew = (double **)alloc2d(my_num_rows + 2, ny + 2, sizeof(double));
f = (double **)alloc2d(my_num_rows + 2, ny + 2, sizeof(double));

aboveid = (myid == 0) ? MPI_PROC_NULL : myid - 1; /*no neighbor above the top cpu*/
belowid = (myid == numprocs - 1) ? MPI_PROC_NULL : myid + 1; /*no neighbor below the bottom cpu*/
init2d_domain(u, unew, f, nx, ny, s, e);
iter = 0;
do {
    double **tempPtr;
    exchng1d(u, ny, s, e, MPI_COMM_WORLD, aboveid, belowid);
    sweep1d(u, f, ny, s, e, unew);

    mydiff = diff(u, unew, ny, s, e);
    MPI_Allreduce( &mydiff, &diffNorm, 1, MPI_DOUBLE, MPI_SUM, MPI_COMM_WORLD);

    tempPtr = unew;
    unew = u;
    u = tempPtr;
} while (++iter < maxIter && diffNorm > eps);
MPI_Barrier(MPI_COMM_WORLD);
t2 = MPI_Wtime();
if (diffNorm <= eps) {
    char path_string[32];
    int i, end_print_row, start_print_row;
    FILE *output_file;
    /* Print top boundary if we are root cpu */
    /* Print bottom boundary if we are bottom cpu*/
    start_print_row = (myid == 0) ? 0 : 1;
    end_print_row = (myid == numprocs - 1) ? my_num_rows+1 : my_num_rows;
    snprintf(path_string, 32, "output[%i].txt", myid);
    output_file = fopen(path_string, "w+");
    for (i = start_print_row; i <= end_print_row; i++)
        printRow(u[i], ny+2, output_file);

    if (myid == 0) {
        FILE *control_file;
        printf("Convergence: Procs = %i, Size = %ix%i, Iterations = %d, WC seconds=%f\n",
                numprocs, nx, ny, iter, t2 - t1);

        control_file = fopen("control.txt", "w+");
        for(i = 0; i < numprocs; i++)
            fprintf(control_file, "%i\n", i);  /* decomposition was along rows */
        printf("Results written to 'output[*].txt' and 'control.txt'\n");
        fclose(control_file);
    }
    fclose(output_file);
} else if (myid == 0)
    printf("No Convergence: Iterations = %d, DiffNorm = %lf, WC seconds=%f\n",
                iter, diffNorm, t2 - t1);

free(u);
free(unew);
free(f);
MPI_Finalize();
return 0;
/***************************** allocate2d ****************************************************************************/
* Allocates memory for a 2 dimensional array of size nRows * nColumns.
* Allocates all the memory in one large chunk to ensure the last element
* of a row is contiguous with the first of the next.
*******************************************************************************/

void **alloc2d(int nRows, int nColumns, int size)
{
    int i;
    void **anArray = (void **) malloc(nRows * nColumns * size + nRows * sizeof(void *));
    anArray[0] = (void *)((long int)anArray + nRows * sizeof(void *));
    for (i = 1; i < nRows; i++)
        anArray[i] = (void *)((long int)anArray[i-1] + nColumns * size);
    return anArray;
}

/**************************** printRow *************************************/
* Writes one row of the processor's u array to aFile.
*******************************************************************************/

void printRow(double *uRow, int nColumns, FILE *aFile)
{
    int j;
    for (j = 0; j < nColumns; j++)
        fprintf(aFile, "%f", uRow[j], (j == nColumns-1) ? "\n" : " , ");
}

/**************************** block_map *************************************/
* Assigns process myid a chunk of u to work on between s and e.
*******************************************************************************/

void block_map(int n1, int n2, int numprocs, int myid, int *s, int *e)
{
    int elements = n2 - n1 + 1, chunk = elements / numprocs, res = elements % numprocs;
    if (myid < res) {
        chunk++;
        *s = n1 + myid * chunk;
    } else
        *s = n1 + myid * chunk + res;
    *e = (*s + chunk - 1);
    *e = (*e > n2) ? n2 : *e;
}
void init2d_domain( double **u, double **unew, double **f, int nx, int ny, int s, int e) {
    double pi = 3.1415926535897932, h = 1.0/(ny + 1);
    double cos_pix, x, y, pi_squared = pi*pi, pi_squared_inv = 1/pi_squared;
    int i, j;
    /* Make "initial guess" of zero
    Also, set f(x,y) = (x^2 + y^2) * cos(x*y) - cos(pi * x). */
    for (i = s-1; i <= e + 1; i++) {
        for (j = 0; j <= ny + 1; j++) {
            u[i-s+1][j] = unew[i-s+1][j] = 0.0;
            x = i * h;
            y = j * h;
            f[i-s+1][j] = (x*x+y*y)*cos(x*y) - cos(pi*x);
        }
    }
    /*************************** init2d_domain ***************************
    * Initializes all u and unew points to 0.0, and sets the boundary
    * conditions and the Laplacian as follows:
    * y = 0: u(x, 0)=(cos(pi*x) - pi*pi)/(pi*pi)
    * y = 1: u(x, nx+1) = (cos(pi*x) - pi*pi*cos(x))/(pi*pi)
    * x = 0: u(0,y) = 1/(pi*pi) - 1
    * x = 1: u(1,y) = -(1/(pi*pi) + cos(y))
    * f(x,y) = (x^2 + y^2) * cos(x*y) - cos(pi * x)
    ********************************************************************/
/** Initialize boundaries along y-axis (rows in u and unew): **/
* x = 0: u(0,y) = 1/(pi*pi) - 1
* x = 1: u(1,y) = -(1/(pi*pi) + cos(y))
*******************************************************************************/
if (s == 1)  /* if our processor borders the first row */
  for (j = 0; j <= ny+1; j++)
    u[0][j] = unew[0][j] = pi_squared_inv - 1;
if (e == nx)  /* if our processor borders the last row */
  for (j = 0; j <= ny+1; j++)
    u[e-s+2][j] = unew[e-s+2][j] = -(pi_squared_inv + cos(j * h));
}
*******************************************************************************/

void exchngcd(double **u, int ny, int s, int e, MPI_Comm commld, int aboveid, int belowid)
{
  MPI_Status status_array[4];
  MPI_Request request[4];
  MPI_Irecv (&u[0][1], ny, MPI_DOUBLE, aboveid, 0, commld, &request[0]);
  MPI_Irecv (&u[e-s+2][1], ny, MPI_DOUBLE, belowid, 0, commld, &request[1]);
  MPI_Isend (&u[e-s+1][1], ny, MPI_DOUBLE, belowid, 0, commld, &request[2]);
  MPI_Isend (&u[1][1], ny, MPI_DOUBLE, aboveid, 0, commld, &request[3]);
  MPI_Waitall (4, request, status_array);
}
/**
* Perform a 2-dim (x-y) Jacobi sweep for a 1-d (column) decomposition
* of the x-y domain. Sweep/update from u into unew.
**/

void sweep1d(double **u, double **f, int ny, int s, int e, double **unew)
{
    int i, j;
    double h = 1.0/(ny+1);
    for (i=1; i <= e - s + 1; i++) {
        for (j=1; j <= ny; j++)
            unew[i][j] = 0.25 * (u[i-1][j]+u[i][j+1]+u[i][j-1]+u[i+1][j]-h*h*f[i][j]);
    }
}

/**
* Computes the convergence "norm" (the sum of the squares of the
* differences between the old and the new approximations).
**/

double diff(double **u, double **unew, int ny, int s, int e)
{
    double sum = 0.0;
    int i, j;

    for (i = 1; i <= e - s + 1; i++) {
        for (j = 1; j <= ny; j++) {
            double diff = u[i][j] - unew[i][j];
            sum += diff*diff;
        }
    }
    return sum;
}
Plotting Results in Matlab

Once execution completes, the solver will output:

Convergence: Procs = 4, Size = 100x100, Iterations = 5133, WC seconds=1.354822
Results written to 'output[*].txt' and 'control.txt'

The following Matlab program can be run to import the results for plotting:

```
function results = import_data

control = csvread('control.txt');
[rows,cols] = size(control);

results = [];
for i = 1:rows
    theRow = [];
    for j = 1:cols
        procResults = csvread(sprintf('output[%i].txt',control(i,j)));
        theRow = cat(2, theRow, procResults);  %concatenate along columns
    end
    results = cat(1, results, theRow);  %concatenate along rows
end
results = transpose(results);  %jacobi output was the transpose of the physical problem
```

This script assumes the control.txt and output[*].txt files are in the working directory, and it returns an array constructed from these files.
Example:
```matlab
results = import_data;
[x,y] = meshgrid(0:1/101:1);
mesh(x,y,results);
```
Benchmarking jacobi_1d.exe on Hydra

3000x3000 Size Problem – \( \epsilon = 1 \times 10^{-5} \) – 140082 Iterations

<table>
<thead>
<tr>
<th>Num Procs</th>
<th>WC Time (Seconds)</th>
<th>Theoretical Speedup</th>
<th>Actual Speedup</th>
<th>Wallclock Time (Minutes)</th>
<th>CPU Time (Minutes)</th>
<th>CPU Time (Hours)</th>
<th>Proc. Grid</th>
<th>CPU Subgrid Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>21647.761</td>
<td>1.000</td>
<td>1.000</td>
<td>360.796</td>
<td>360.796</td>
<td>6.013</td>
<td>1x1</td>
<td>3000x3000</td>
</tr>
<tr>
<td>2</td>
<td>11452.919</td>
<td>2.000</td>
<td>1.890</td>
<td>190.882</td>
<td>381.764</td>
<td>6.363</td>
<td>2x1</td>
<td>1500x3000</td>
</tr>
<tr>
<td>4</td>
<td>5673.304</td>
<td>4.000</td>
<td>3.816</td>
<td>94.555</td>
<td>378.220</td>
<td>6.304</td>
<td>2x2</td>
<td>1500x1500</td>
</tr>
<tr>
<td>8</td>
<td>2319.830</td>
<td>8.000</td>
<td>9.332</td>
<td>38.664</td>
<td>309.311</td>
<td>5.155</td>
<td>2x4</td>
<td>1500x750</td>
</tr>
<tr>
<td>16</td>
<td>1152.166</td>
<td>16.000</td>
<td>18.789</td>
<td>19.203</td>
<td>307.244</td>
<td>5.121</td>
<td>4x4</td>
<td>750x750</td>
</tr>
<tr>
<td>24</td>
<td>1103.765</td>
<td>24.000</td>
<td>19.613</td>
<td>18.396</td>
<td>441.506</td>
<td>7.358</td>
<td>4x6</td>
<td>750x500</td>
</tr>
<tr>
<td>28</td>
<td>1025.374</td>
<td>28.000</td>
<td>21.112</td>
<td>17.090</td>
<td>478.508</td>
<td>7.975</td>
<td>4x7</td>
<td>750x428.571</td>
</tr>
</tbody>
</table>

Wallclock Time vs. CPUs

Speedup vs. CPUs

Theoretical Speedup
Actual Speedup
Two-Dimensional BLOCK DISTRIBUTION: POISON PROBLEM

Showing border columns and rows that will have to be copied into the address space of a neighbor process during a (5-point) finite-difference iteration scheme (jacobi). Gray squares represent boundary/ghost elements that are neither transmitted nor computed. White squares are elements computed by the processor that holds them.
Two-Dimensional BLOCK DISTRIBUTION: POISSON PROBLEM
Column & Row Exchanges Needed for Parallel 5-point FDM Jacobi Iteration

Showing exchanges of border columns and rows per process. A border column/row from one process is copied into a "ghost" column/row of a neighbor’s address space. Shaded empty cells are not transmitted, as they do not fall within the stencil used for the jacobi iteration.
EXAMPLE: SOLVING the x-y POISSON EQUATION IN PARALLEL (MPI - C)

Two-Dimensional Decomposition With MPI Cartesian Routines

/***************************************************************************/
/* Computes a solution to the two dimensional Poisson problem using */
/* Jacobi iteration on a 2-d domain decomposition. The domain de- */
/* composition is carried out along the x- and y-axis, i.e., along the */
/* rows and columns of arrays u and unew. Cartesian MPI routines */
/* are used to create the processor grid. A general multidimensional */
/* dynamic array allocation routine is used. */
/* */
/* _ 2 _ 2 */
/* x,y in [ 0, 1 ] */
/* d u d u */
/* Problem: --- + --- = f(x,y) = (x^2 + y^2) * cos(x*y) - cos(pi * x) */
/* _ 2 _ 2 */
/* dx dy */
/* Boundary Conditions */
/* y = 0: u(x, 0)=(cos(pi*x) - pi*pi)/(pi*pi) */
/* y = 1: u(x, nx +1) = (cos(pi*x) - pi*pi*cos(x))/(pi*pi) */
/* x = 0: u(0,y) = 1/(pi*pi) - 1 */
/* x = 1: u(1,y) = -(1/(pi*pi) + cos(y)) */
/* */
/* Programmer: Julian Panetta */
/* */
/* The following source file is a modification of source files, */
/* at the intermediate level, that accompany the Using MPI textbook */
/* (ref. 2) by Gropp and others. */
/* */
/* The Laplacian and the boundary conditions used in this problem */
/* were taken from section 12.1 exercise 3f of the fifth edition */
/* of Numerical Analysis by Burden and Faires. */
/* */
/* The size of the domain is read by processor 0 and broadcast to */
/* all other processors. The Jacobi iteration is run until the */
/* change in successive elements is small or a maximum number of */
/* iterations is reached. The results from each processor are */
/* output to separate files in the format output[#].txt where */
/* # is the cpu's rank. */
/***************************************************************************/

#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <mpi.h>
#define X 0
#define ROW 0
#define Y 1
#define COL 1
#define LEFT 0
#define RIGHT 1
#define ABOVE 2
#define BELOW 3
#define YES 1
#define NO 0

void **allocMultidim(int n, int dims[], int type_size);

void printRow(double *uRow, int nColumns, FILE *aFile);

void construct_pgrid(int numprocs, int pgrid_size[], int *myid, int my_position[],
    int neighbor_ids[], MPI_Comm *comm2d);

void block_map(int n[], int pgrid_size[], int my_position[], int s[], int e[]);

void init2d_domain(double **u, double **unew, double **f, int *n, int *s, int *e);

void exchng2d(double **u, int *n, int *s, int *e, MPI_Comm comm2d, int *neighbor_ids);
void sweep2d(double **u, double **f, int *n, int *s, int *e, double **unew);
double diff(double **u, double **unew, int *n, int *s, int *e);

int main(int argc, char *argv[])
{
    double **u, **unew, **f;
    int myid, numprocs, my_num_rows, my_num_cols, my_array_dims[2];
    int maxiter, iter;
    int n[2], s[2], e[2], pgrid_size[2];
    int neighbor_ids[4], my_position[2];
    double eps = 1.0e-5, diffNorm, mydiff, t1, t2;
    MPI_Comm comm2d;

    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &numprocs);
    MPI_Comm_rank(MPI_COMM_WORLD, &myid);
if (myid == 0) {
    do {
        printf("Enter problem (Square mesh) size:\n");
        scanf("%d", &n[X]);
        if (n[X] < 100)
            printf("--You must enter a size greater than 100.--\n");
    } while (n[X] < 100);
    do {
        printf("Enter maximum number of iterations:\n");
        scanf("%d", &maxiter);
        if (maxiter < 1)
            printf("--You must enter a positive number of iterations!--\n");
    } while (maxiter < 1);
}

MPI_Barrier(MPI_COMM_WORLD);

t1 = MPI_Wtime();

MPI_Bcast(&n[X], 1, MPI_INT, 0, MPI_COMM_WORLD);
MPI_Bcast(&maxiter, 1, MPI_INT, 0, MPI_COMM_WORLD);

n[Y] = n[X];

/*Note: construct_pgrid updates myid to the processor's rank within
 * the new cartesian communicator so that it can reflect reordering.
 * This means a new cpu may become the root processor!*/
construct_pgrid(numprocs, pgrid_size, &myid, my_position, neighbor_ids, &comm2d);

block_map(n, pgrid_size, my_position, s, e);

/**************************************************************************
 ** Allocate only the chunk of the array needed to hold the processor's
 ** points and ghost rows and columns for the adjacent processors (or
 ** boundary row and column vectors in the case of the edge processors).
**************************************************************************/
my_num_rows = e[X] - s[X] + 1;
my_num_cols = e[Y] - s[Y] + 1;

my_array_dims[ROW] = my_num_rows + 2;
my_array_dims[COL] = my_num_cols + 2;

u = (double **) allocMultidim(2, my_array_dims, sizeof(double));
unew = (double **) allocMultidim(2, my_array_dims, sizeof(double));
init2d_domain(u, unew, f, n, s, e);
iter = 0;

do {
    double **tempPtr;
    exchng2d(u, n, s, e, comm2d, neighbor_ids);
    sweep2d(u, f, n, s, e, unew);

    mydiff = diff(u, unew, n, s, e);
    MPI_Allreduce(&mydiff, &diffNorm, 1, MPI_DOUBLE, MPI_SUM, comm2d);

    tempPtr = unew;
    unew = u;
    u = tempPtr;
} while (++iter < maxiter && diffNorm > eps);

MPI_Barrier(MPI_COMM_WORLD);
t2 = MPI_Wtime();
if (diffNorm <= eps) {
    char path_string[32];
    int i, j, end_print_row, start_print_row;
    int coords[2];
    int end_print_col, start_print_col;
    FILE *output_file;

    start_print_row = (my_position[X] == 0) ? 0 : 1;
    end_print_row = (my_position[X] == pgrid_size[X] - 1) ? my_num_rows+1 : my_num_rows;
    start_print_col = (my_position[Y] == 0) ? 0 : 1;

    snprintf(path_string, 32, "output[%i].txt", myid);
    output_file = fopen(path_string, "w+");

    for (i = start_print_row; i <= end_print_row; i++)
        printRow(&u[i][start_print_col], end_print_col - start_print_col + 1, output_file);
if (myid == 0) {
    FILE *control_file;
    printf("Convergence: Procs = %i, Size = %ix%i, Iterations = %d, WC seconds=%f\n",
           numprocs, n[X], n[Y], iter, t2 - t1);

    control_file = fopen("control.txt", "w+");

    for(coords[ROW] = 0; coords[ROW] < pgrid_size[X]; coords[ROW]++) {
        for(coords[COL] = 0; coords[COL] < pgrid_size[Y]; coords[COL]++) {
            int cart_rank;
            MPI_Cart_rank(comm2d, coords, &cart_rank);
            fprintf(control_file, "%i%s",
                    cart_rank,
                    (coords[COL] < pgrid_size[Y] - 1) ? ", " : "\n");
        }
    }
    printf("Results written to 'output[*].txt' and 'control.txt'\n");
    fclose(control_file);
}

fclose(output_file);

} else if (myid == 0) {
    printf("No Convergence: Iterations = %d, DiffNorm = %lf, WC seconds=%f\n",
           iter, diffNorm, t2 - t1);
}

free(u);
free(unew);
free(f);

MPI_Finalize();
return 0;
/************************** allocMultidim ***************************/
* Allocate memory for a n-dimensional array, with dimensions given in *
* dims. Allocate all the memory in one large chunk to ensure (in the *
* 2d case) the last element of a row is contiguous with the first of *
* the next.

void **allocMultidim(int n, int dims[], int type_size)
{
    int nAbove = 1, indexEntries = 0, columnVectors=1, i, j, p=0;
    void **anArray;

    for(i = 0; i < n-1; i++)
    {
        columnVectors *= dims[i];
        indexEntries += columnVectors;
    }
    anArray = (void **)malloc(indexEntries*sizeof(void *) + columnVectors*dims[n-1]*type_size);

    for(i = 0; i < n - 1; i++)
    {
        /* build index */
        anArray[p] = (void *)((long)&anArray[p] + nAbove * dims[i]*sizeof(void *));
        for(j = p + 1; j < p + dims[i] * nAbove; j++)
            anArray[j] = (void *)((long)anArray[j-1] + dims[i+1] * ((i < n-2) ? sizeof(void *) : type_size));
        nAbove *= dims[i];
        p += nAbove;
    }
    return anArray;
}

/**************************** printRow ******************************/
* Writes one row of the processor's u array to aFile. *

void printRow(double *uRow, int nColumns, FILE *aFile)
{
    int j;
    for (j = 0; j < nColumns; j++)
        fprintf(aFile, "%f%s", uRow[j], (j == nColumns-1) ? "\n" : ",");
}
/************ construct_pgrid *************/ * Arranges nprocs cpus into a grid that is as squarelike as possible. +* If nprocs is a square, a true square grid will be achieved, but if +* it is prime the decomposition will be equivalent--but slightly less +* efficient--than the 1d decomposition. */    +--+--+ * [0 |1 | This is an example of the resultant processor grid +--+--+ when nprocs = 6. Because we call MPI_Cart_create + |2 |3 | with reorder=NO, however, whether the processors +--+--+ are arranged row-wise as shown depends on the + |4 |5 | MPI implementation. +--+--+ */    */    **************************** block_map ****************************/ * Assigns process myid a rectangular chunk of u to work on bounded * by sx, ex, sy, and ey. * It implements a "block" assignment of n total * amount of work. */    void block_map(int n[], int pgrid_size[], int *myid, int my_position[], int s[], int e[]) { }
/************************** init2d_domain ***************************
* Initializes all u and unew points to 0.0, and sets the boundary
* conditions and the laplacian as follows:
*  
y = 0: u(x, 0)=(cos(pi*x) - pi*pi)/(pi*pi)
*  
y = 1: u(x, nx +1) = (cos(pi*x) - pi*pi*cos(x))/(pi*pi)
*  
x = 0: u(0,y) = 1/(pi*pi) - 1
*  
x = 1: u(1,y) = -(1/(pi*pi) + cos(y))
*  
f(x,y) = (x^2 + y^2) * cos(x*y) - cos(pi * x)
******************************************************************/

void init2d_domain( double **u, double **unew, double **f, int *n, int *s, int *e)
{
    double pi = 3.1415926535897932;
    double h = 1.0/(n[Y] + 1);
    double x, y;
    double pi_squared = pi*pi;
    double pi_squared_inv = 1/pi_squared;
    int i, j;
    int my_num_rows = e[X] - s[X] + 1;
    int my_num_cols = e[Y] - s[Y] + 1;
    /* Make "initial guess" of zero
    Also, set f(x,y) = (x^2 + y^2) * cos(x*y) - cos(pi * x). */
    for (i = 0; i <= my_num_rows+1; i++)
    {
        for (j = 0; j <= my_num_cols+1; j++)
        {
            u[i][j] = unew[i][j] = 0.0;
            x = (i + s[X] - 1) * h;   /* Compute x[i] and y[j], where i,j are relative */
            y = (j + s[Y] - 1) * h;   /* to processor's s[X] and s[Y] offsets*/
            f[i][j] = (x*x+y*y)*cos(x*y) - cos(pi*x);
        }
    }
    /* Initialize boundary along the x-axis (columns in u and unew)
    y=0: u(x, 0)=(cos(pi*x) - pi*pi)/(pi*pi)
    y=1: u(x, nx +1) = (cos(pi*x) - pi*pi*cos(x))/(pi*pi)
    Each processor only initializes the portion of these boundries it will
    need for the stencil we are using.
    *************************************************/
    if (s[Y] == 1) /*if our processor borders the first column it must be initialized*/
    {
        for (i = 1; i <= my_num_rows; i++)
        {
            x = (i + s[X] - 1) * h;
            u[i][0] = unew[i][0] = (cos(pi * x) - pi_squared) * pi_squared_inv;
        }
    }
    if (e[Y] == n[Y])
    {
        for (i = 1; i <= my_num_rows; i++)
        {
            x = (i + s[X] - 1) * h;
            u[i][my_num_cols + 1] = unew[i][my_num_cols + 1] = (cos(pi * x) -
                                           - pi_squared * cos(x)) * pi_squared_inv;
        }
    }
/* Initialize boundaries along y-axis (rows in u and unew):
* x = 0: u(0,y) = 1/(pi*pi) - 1
* x = 1: u(1,y) = -(1/(pi*pi) + cos(y))
*/

if (s[X] == 1)  /* if our processor borders the first row */
  for (j = 0; j <= my_num_cols + 1; j++)
    u[0][j] = unew[0][j] = pi_squared_inv - 1;

if (e[X] == n[X])  /* if our processor borders the last row */
  for (j = 0; j <= my_num_cols + 1; j++)
    u[my_num_rows + 1][j] = unew[my_num_rows + 1][j] = -(pi_squared_inv + cos((j + s[Y] - 1) * h));

}  /* *************** exchng2d *************** */

/* Each process executes founds sends and two receives. It sends its border
* rows and columns to its neighbors above, below, and to the left
* and right. The same process also receives four rows and columns. One from
* each of its cardinal neighbors. These rows and columns fit in the
* "ghost" rows s[X]-1 and e[X]+1 and "ghost" columns s[Y]-1 and e[Y]+1.
* Because column elements are not consecutive in memory, we pack them
* into a column vector array to send.
*/

void exchng2d(double **u, int *n, int *s, int *e, MPI_Comm comm2d, int *neighbor_ids)
{
  int my_num_rows = e[X] - s[X] + 1;
  int my_num_cols = e[Y] - s[Y] + 1;
  MPI_Status status_array[8];
  MPI_Request request[8];
  MPI_Datatype column_type;
  /*my_num_rows entries of one double separated by my_num_cols+2 in a column vector*/
  MPI_Type_vector(my_num_rows, 1, my_num_cols+2, MPI_DOUBLE, &column_type);
  MPI_Type_commit(&column_type);

  MPI_Irecv(&u[0][1], my_num_cols, MPI_DOUBLE, neighbor_ids[ABOVE], 0, comm2d, &request[0]);
  MPI_Irecv(&u[my_num_rows+1][1], my_num_cols, MPI_DOUBLE, neighbor_ids[BELOW], 0, comm2d, &request[1]);
  MPI_Irecv(&u[1][0], 1, column_type, neighbor_ids[LEFT], 0, comm2d, &request[2]);
  MPI_Irecv(&u[1][my_num_cols+1], 1, column_type, neighbor_ids[RIGHT], 0, comm2d, &request[3]);

  MPI_Isend(&u[my_num_rows][1], my_num_cols, MPI_DOUBLE, neighbor_ids[BELOW], 0, comm2d, &request[4]);
  MPI_Isend(&u[1][1], my_num_cols, MPI_DOUBLE, neighbor_ids[ABOVE], 0, comm2d, &request[5]);
  MPI_Isend(&u[1][1], 1, column_type, neighbor_ids[LEFT], 0, comm2d, &request[6]);
  MPI_Isend(&u[1][my_num_cols], 1, column_type, neighbor_ids[RIGHT], 0, comm2d, &request[7]);

  MPI_Waitall(8, request, status_array);
  MPI_Type_free(&column_type);
}
/********************** sweep2d ************************************
* Perform a 2-dim (x-y) Jacobi sweep for a 1-d (column) decomposition
* of the x-y domain. Sweep/update from u into unew.
****************************** diff ****************************************
* Computes the convergence "norm" (the sum of the squares of the
* differences between the old and the new approximations).
***************************************************************************/
void sweep2d(double **u, double **f, int *n, int *s, int *e, double **unew)
{
    int i, j;
    double h = 1.0/(n[Y]+1);
    for (i=1; i <= e[X] - s[X] + 1; i++) {
        for (j=1; j <= e[Y] - s[Y] + 1; j++)
            unew[i][j] = 0.25 * (u[i-1][j]+u[i][j+1]+u[i][j-1]+u[i+1][j]-h*h*f[i][j]);
    }
}

double diff(double **u, double **unew, int *n, int *s, int *e)
{
    double sum = 0.0;
    int i, j;
    for (i = 1; i <= e[X] - s[X] + 1; i++) {
        for (j = 1; j <= e[Y] - s[Y] + 1; j++) {
            double diff = u[i][j] - unew[i][j];
            sum += diff*diff;
        }
    }
    return sum;
}
**Benchmarking jacobi_2d_cart.exe on Hydra**

3000x3000 Size Problem - \( \epsilon = 1 \times 10^{-5} \) - 140082 Iterations

<table>
<thead>
<tr>
<th>Num Procs</th>
<th>WC Time (Seconds)</th>
<th>Theoretical Speedup</th>
<th>Actual Speedup</th>
<th>Wallclock Time (Minutes)</th>
<th>CPU Time (Minutes)</th>
<th>CPU Time (Hours)</th>
<th>Proc. Grid</th>
<th>CPU Subgrid Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>21647.761</td>
<td>1.000</td>
<td>1.000</td>
<td>360.796</td>
<td>360.796</td>
<td>6.013</td>
<td>1x1</td>
<td>3000x3000</td>
</tr>
<tr>
<td>2</td>
<td>11452.919</td>
<td>2.000</td>
<td>1.890</td>
<td>190.882</td>
<td>381.764</td>
<td>6.363</td>
<td>2x1</td>
<td>1500x3000</td>
</tr>
<tr>
<td>4</td>
<td>5673.304</td>
<td>4.000</td>
<td>3.816</td>
<td>94.555</td>
<td>378.220</td>
<td>6.304</td>
<td>2x2</td>
<td>1500x1500</td>
</tr>
<tr>
<td>8</td>
<td>2319.830</td>
<td>8.000</td>
<td>9.332</td>
<td>38.664</td>
<td>309.311</td>
<td>5.155</td>
<td>2x4</td>
<td>1500x750</td>
</tr>
<tr>
<td>16</td>
<td>1152.166</td>
<td>16.000</td>
<td>18.789</td>
<td>19.203</td>
<td>307.244</td>
<td>5.121</td>
<td>4x4</td>
<td>750x750</td>
</tr>
<tr>
<td>24</td>
<td>1103.765</td>
<td>24.000</td>
<td>19.613</td>
<td>18.396</td>
<td>441.506</td>
<td>7.358</td>
<td>4x6</td>
<td>750x500</td>
</tr>
<tr>
<td>28</td>
<td>1025.374</td>
<td>28.000</td>
<td>21.112</td>
<td>17.090</td>
<td>478.508</td>
<td>7.975</td>
<td>4x7</td>
<td>750x428.571</td>
</tr>
</tbody>
</table>

**Wallclock Time vs. CPUs**

**Speedup vs. CPUs**
EXAMPLE: SOLVING the x-y POISSON EQUATION IN PARALLEL (MPI - C)

Two-Dimensional Decomposition With Manual Processor Grid

/*****************************************************************************
* jacobi_2d.c *****************************
* Computes a solution to the two dimensional Poisson problem using
* Jacobi iteration on a 2-d domain decomposition. The domain de-
* composition is carried out along the x- and y-axis, i.e., along the
* rows and columns of arrays u and unew. The processor grid is created
* manually, and a specific 2d dynamic array allocation routine is used.
*
* _ 2 _ 2                          x,y in [ 0, 1 ]
* d u   d u
* Problem: --- + --- = f(x,y) =      (x^2 + y^2) * cos(x*y) - cos(pi * x)
*                  _ 2 _ 2                  Boundary Conditions
*                  dx   dy
*      y = 0: u(x, 0)=(cos(pi*x) - pi*pi)/(pi*pi)
*      y = 1: u(x, nx +1) = (cos(pi*x) - pi*pi*cos(x))/(pi*pi)
*      x = 0: u(0,y) = 1/(pi*pi) - 1
*      x = 1: u(1,y) = -(1/(pi*pi) + cos(y))
* Programmer: Julian Panetta
* The following source file is a modification of source files,
* at the intermediate level, that accompany the Using MPI textbook
* (ref. 2) by Gropp and others.
* The Laplacian and the boundary conditions used in this problem
* were taken from section 12.1 exercise 3f of the fifth edition
* of Numerical Analysis by Burden and Faires.
* The size of the domain is read by processor 0 and broadcast to
* all other processors. The Jacobi iteration is run until the
* change in successive elements is small or a maximum number of
* iterations is reached. The results from each processor are
* output to separate files in the format output[#].txt where
* # is the cpu's rank.
*****************************************************************************
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <mpi.h>
#define X 0
#define ROW 0
#define Y 1
#define COL 1
#define LEFT 0
#define RIGHT 1
#define ABOVE 2
#define BELOW 3

void **alloc2d(int nRows, int nColumns, int size);
void printRow(double *uRow, int nColumns, FILE *aFile);

void construct_pgrid(int myid, int numprocs, int pgrid_size[], int my_position[], int neighbor_ids[]);
void block_map(int n[], int pgrid_size[], int my_position[], int s[], int e[]);
void init2d_domain(double **u, double **unew, double **f, int *n, int *s, int *e);

void exchng2d(double **u, int *n, int *s, int *e, MPI_Comm comm2d, int *neighbor_ids);
void sweep2d(double **u, double **f, int *n, int *s, int *e, double **unew);
double diff(double **u, double **f, int *n, int *s, int *e, double **unew);

int main(int argc, char *argv[])
{
    double **u, **unew, **f;
    int myid, numprocs, my_num_rows, my_num_cols;
    int maxiter, iter;
    int n[2], s[2], e[2], pgrid_size[2];
    int neighbor_ids[4], my_position[2];
    double eps = 1.0e-5, diffNorm, mydiff, t1, t2;

    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &numprocs);
    MPI_Comm_rank(MPI_COMM_WORLD, &myid);
    if (myid == 0) {
        do {
            printf("Enter problem (Square mesh) size:\n");
            scanf("%d", &n[X]);
            if (n[X] < 100)
                printf("--You must enter a size greater than 100.--\n");
        } while (n[X] < 100);
        do {
            printf("Enter maximum number of iterations:\n");
            scanf("%d", &maxiter);
            if (maxiter < 1)
                printf("--You must enter a positive number of iterations!--\n");
        } while (maxiter < 1);
    }
}
MPI_Barrier(MPI_COMM_WORLD);
t1 = MPI_Wtime();
MPI_Bcast(&n[X], 1, MPI_INT, 0, MPI_COMM_WORLD);
MPI_Bcast(&maxiter, 1, MPI_INT, 0, MPI_COMM_WORLD);

n[Y] = n[X];

construct_pgrid(myid, numprocs, pgrid_size, my_position, neighbor_ids);

block_map(n, pgrid_size, my_position, s, e);

/*****************************************************************
* Allocate only the chunk of the array needed to hold the processor's
* points and ghost rows and columns for the adjacent processors (or
* boundary row and column vectors in the case of the edge processors).
*****************************************************************/
my_num_rows = e[X] - s[X] + 1;
my_num_cols = e[Y] - s[Y] + 1;

u = (double **) alloc2d(my_num_rows + 2, my_num_cols + 2, sizeof(double));
unew = (double **) alloc2d(my_num_rows + 2, my_num_cols + 2, sizeof(double));
f = (double **) alloc2d(my_num_rows + 2, my_num_cols + 2, sizeof(double));

init2d_domain(u, unew, f, n, s, e);
iter = 0;

do {
  double **tempPtr;
  exchng2d(u, n, s, e, MPI_COMM_WORLD, neighbor_ids);
  sweep2d(u, f, n, s, e, unew);

  mydiff = diff(u, unew, n, s, e);
  MPI_Allreduce( &mydiff, &diffNorm, 1, MPI_DOUBLE, MPI_SUM, MPI_COMM_WORLD);

  tempPtr = unew;
  unew = u;
  u = tempPtr;
} while (++iter < maxiter && diffNorm > eps);
MPI_Barrier(MPI_COMM_WORLD);
t2 = MPI_Wtime();
if (diffNorm <= eps) {
    char path_string[32];
    int i, j, end_print_row, start_print_row;
    int end_print_col, start_print_col;
    FILE *output_file;
    /*Processors on the edges and corners of the grid should print the boundaries*/
    start_print_row = (my_position[X] == 0) ? 0 : 1;
    end_print_row = (my_position[X] == pgrid_size[X] - 1) ? my_num_rows + 1 : my_num_rows;
    start_print_col = (my_position[Y] == 0) ? 0 : 1;

    snprintf(path_string, 32, "output[%i].txt", myid);
    output_file = fopen(path_string, "w+");
    for (i = start_print_row; i <= end_print_row; i++)
        printRow(&u[i][start_print_col], end_print_col - start_print_col + 1, output_file);

    if (myid == 0) {
        FILE *control_file;
        printf("Convergence: Procs = %i, Size = %ix%i, Iterations = %d, WC seconds=%f\n",
                numprocs, n[X], n[Y], iter, t2 - t1);

        control_file = fopen("control.txt", "w+");
        for (i = 0; i < pgrid_size[X]; i++)
            for (j = 0; j < pgrid_size[Y]; j++)
                fprintf(control_file, "%i%s", i * pgrid_size[Y] + j,
                        (j < pgrid_size[Y] - 1) ? "," : ":\n");
        printf("Results written to 'output[*].txt' and 'control.txt'\n");
        fclose(control_file);
    }
    fclose(output_file);
} else if (myid == 0) {
    printf("No Convergence: Iterations = %d, DiffNorm = %lf, WC seconds=%f\n",
            iter, diffNorm, t2 - t1);
}
free(u);
free(unew);
free(f);
MPI_Finalize();
return 0;
allocate2d

Allocate memory for a 2 dimensional array of size nRows * nColumns.
Allocate all the memory in one large chunk to ensure the last element of a row is contiguous with the first of the next.

```c
void **alloc2d(int nRows, int nColumns, int size)
{
    int i;
    void **anArray = (void **) malloc(nRows * nColumns * size + nRows * sizeof(void *));
    anArray[0] = (void *)((long int)anArray + nRows * sizeof(void *));
    for (i = 1; i < nRows; i++)
        anArray[i] = (void *)((long int)anArray[i-1] + nColumns * size);
    return anArray;
}
```

printRow

Write one row of the processor’s u array to aFile.

```c
void printRow(double *uRow, int nColumns, FILE *aFile)
{
    int j;
    for (j = 0; j < nColumns; j++)
        fprintf(aFile, "%f%s", uRow[j], (j == nColumns-1) ? "\n" : ",");
}
```

construct_pgrid

Arranges nprocs cpus into a grid that is as squarelike as possible.
If nprocs is a square, a true square grid will be achieved, but if it is prime the decomposition will be equivalent—but slightly less efficient—than the 1d decomposition.

```c
void construct_pgrid(int myid, int nprocs, int pgrid_size[], int my_position[], int neighbor_ids[])
{
    for (pgrid_size[ROW] = ceil(sqrt(nprocs)); numprocs % pgrid_size[ROW]; pgrid_size[ROW]--);
    pgrid_size[COL] = numprocs/pgrid_size[ROW];
    my_position[ROW] = myid / pgrid_size[COL];
    my_position[COL] = myid % pgrid_size[COL];
    neighbor_ids[LEFT] = (my_position[COL] == 0) ? MPI_PROC_NULL : myid-1;
    neighbor_ids[RIGHT] = (my_position[COL] == pgrid_size[COL] - 1) ? MPI_PROC_NULL : myid + 1;
    neighbor_ids[ABOVE] = (my_position[ROW] == 0) ? MPI_PROC_NULL: myid - pgrid_size[COL];
    neighbor_ids[BELOW] = (my_position[ROW] == pgrid_size[ROW] - 1) ? MPI_PROC_NULL : myid + pgrid_size[COL];
}
```
void block_map(int n[], int pgrid_size[], int my_position[], int s[], int e[])
{
    int chunk, res, i;
    for (i = 0; i < 2; i++) { /* Loop over x and y dimensions */
        chunk = n[i] / pgrid_size[i];
        res = n[i] % pgrid_size[i];
        if (my_position[i] < res) {
            chunk++;
            s[i] = 1 + my_position[i] * chunk;
        } else {
            s[i] = 1 + my_position[i] * chunk + res;
        }
        e[i] = (s[i] + chunk - 1);
        e[i] = (e[i] > n[i]) ? n[i] : e[i];
    }
}

void init2d_domain( double **u, double **unew, double **f, int *n, int *s, int *e) {
    double pi = 3.1415926535897932;
    double h = 1.0/(n[Y] + 1);
    double x, y;
    double pi_squared = pi*pi;
    double pi_squared_inv = 1/pi_squared;
    int i, j;
    int my_num_rows = e[X] - s[X] + 1;
    int my_num_cols = e[Y] - s[Y] + 1;
/* Make "initial guess" of zero
Also, set \( f(x,y) = (x^2 + y^2) \cos(xy) - \cos(\pi x) \). */
for (i = 0; i <= my_num_rows+1; i++)
    for (j = 0; j <= my_num_cols+1; j++)
        u[i][j] = unew[i][j] = 0.0;
        x = (i + s[X] - 1) * h;  /* Compute x[i] and y[j], where i, j are relative */
        y = (j + s[Y] - 1) * h;  /* to processor's s[X] and s[Y] offsets*/
        f[i][j] = (x*x+y*y)*cos(x*y) - cos(pi*x);
}

/*****************************************************************
* Initialize boundary along the x-axis (columns in u and unew)
* y=0: \( u(x, 0) = \left( \cos(\pi x) - \pi^2 \right) / \pi^2 \)
* y=1: \( u(x, nx + 1) = \left( \cos(\pi x) - \pi^2 \cos(x) \right) / \pi^2 \)
* Each processor only initializes the portion of these boundaries it will
* need for the stencil we are using.
******************************************************************/
if (s[Y] == 1) { /*if our processor borders the first column it must be initialized*/
    for (i = 1; i <= my_num_rows; i++)
        x = (i + s[X] - 1) * h;
        u[i][0] = unew[i][0] = (cos(pi * x) - pi_squared) * pi_squared_inv;
}
if (e[Y] == n[Y]) {
    for (i = 1; i <= my_num_rows; i++)
        x = (i + s[X] - 1) * h;
        u[i][my_num_cols + 1] = unew[i][my_num_cols + 1] =
            (cos(pi * x) - pi_squared * cos(x)) * pi_squared_inv;
}

/*****************************************************************
* Initialize boundaries along y-axis (rows in u and unew):
* x = 0: \( u(0,y) = 1/(\pi^2) - 1 \)
* x = 1: \( u(1,y) = -(1/(\pi^2) + \cos(y)) \)
******************************************************************/
if (s[X] == 1) { /* if our processor borders the first row */
    for (j = 0; j <= my_num_cols + 1; j++)
        u[0][j] = unew[0][j] = pi_squared_inv - 1;
}
if (e[X] == n[X]) {
    for (j = 0; j <= my_num_cols + 1; j++)
        u[my_num_rows + 1][j] = unew[my_num_rows + 1][j] =
            -(pi_squared_inv + cos((j + s[Y] - 1) * h));
}
Each process executes founds sends and two receives. It sends its border rows and columns to its neighbors above, below, and to the left and right. The same process also receives four rows and columns. One from each of its cardinal neighbors. These rows and columns fit in the "ghost" rows s[X]-1 and e[X]+1 and "ghost" columns s[Y]-1 and e[Y]+1.

```c
void exchng2d(double **u, int *n, int *s, int *e, MPI_Comm comm2d, int *neighbor_ids) {
    int my_num_rows = e[X] - s[X] + 1;
    int my_num_cols = e[Y] - s[Y] + 1;

    MPI_Status status_array[8];
    MPI_Request request[8];
    MPI_Datatype column_type;

    /*my_num_rows entries of one double separated by my_num_cols+2 in a column vector*/
    MPI_Type_vector(my_num_rows, 1, my_num_cols+2, MPI_DOUBLE, &column_type);
    MPI_Type_commit(&column_type);

    MPI_Irecv(&u[0][1], my_num_cols, MPI_DOUBLE, neighbor_ids[ABOVE], 0, comm2d, &request[0]);
    MPI_Irecv(&u[my_num_rows+1][1], my_num_cols, MPI_DOUBLE, neighbor_ids[BELOW], 0, comm2d, &request[1]);
    MPI_Irecv(&u[1][0], 1, column_type, neighbor_ids[LEFT], 0, comm2d, &request[2]);
    MPI_Irecv(&u[1][my_num_cols+1], 1, column_type, neighbor_ids[RIGHT], 0, comm2d, &request[3]);

    MPI_Isend(&u[my_num_rows][1], my_num_cols, MPI_DOUBLE, neighbor_ids[BELOW], 0, comm2d, &request[4]);
    MPI_Isend(&u[1][1], my_num_cols, MPI_DOUBLE, neighbor_ids[ABOVE], 0, comm2d, &request[5]);
    MPI_Isend(&u[1][1], 1, column_type, neighbor_ids[LEFT], 0, comm2d, &request[6]);
    MPI_Isend(&u[1][my_num_cols], 1, column_type, neighbor_ids[RIGHT], 0, comm2d, &request[7]);

    MPI_Waitall(8, request, status_array);
    MPI_Type_free(&column_type);
}
```
/*************** sweep2d *****************/
* Perform a 2-dim (x-y) Jacobi sweep for a 1-d (column) decomposition
* of the x-y domain. Sweep/update from u into unew.
*******************************************************************************/
void sweep2d(double **u, double **f, int *n, int *s, int *e, double **unew)
{
    int i, j;
    double h = 1.0/(n[Y]+1);
    for (i=1; i <= e[X] - s[X] + 1; i++) {
        for (j=1; j <= e[Y] - s[Y] + 1; j++)
            unew[i][j] = 0.25 * (u[i-1][j]+u[i][j+1]+u[i][j-1]+u[i+1][j]-h*h*f[i][j]);
    }
}

/********************************************************* diff ***********************************/
* Computes the convergence "norm" (the sum of the squares of the
* differences between the old and the new approximations).
******************************************************************************/
double diff(double **u, double **unew, int *n, int *s, int *e)
{
    double sum = 0.0;
    int i, j;
    for (i = 1; i <= e[X] - s[X] + 1; i++) {
        for (j = 1; j <= e[Y] - s[Y] + 1; j++) {
            double diff = u[i][j] - unew[i][j];
            sum += diff*diff;
        }
    }
    return sum;
}
EXAMPLE: SOLVING the $x$-$y$ POISSON EQUATION in Parallel (OpenMP)

program openmp_jacobi1d ! Parallel OpenMP VERSION
!************************************************************************
!   openmp_jacobi1d.f – an openmp solution to the 2-d Poisson problem is
!   Effected using the Jacobi iteration with a 5-point finite difference
!   stencil.
!   _ 2   _ 2
!   d u   d u
!   Problem: --- + --- = $f(x,y)$            x,y in [ 0, 1 ]
!   _ 2   _ 2
!   dx   dy
!
!   Programmer: Nagesh, Texas A&M University
!   (The serial version s_jacobi1d.f90 has been parallelized by inserting
!    OpenMP directives)
!
!   The Jacobi iteration is run until the change in successive
!   elements is < eps or a maximum number of iterations is reached.
!************************************************************************
!
integer, parameter :: maxn = 1000
real (kind=8)  u(0:maxn+1,0:maxn+1), unew(0:maxn+1,0:maxn+1),
real (kind=8)  f(maxn,maxn)
integer n, iter, max_iter
real (kind=8) diff, diffnorm

! Enter Problem (Square Mesh) Size   - '; read *, n
if ( n < 100 .OR. n > maxn ) STOP

! Enter Maximum Number of Iterations - '; read *, max_iter
if ( max_iter < 1 ) STOP
!
call init2d_domain( u, unew, f, n ) ! Init the rhs, $f(x)$, of Poisson eq'n
  and set bdry & the initial solution guess
program OpenMP_jacobi1d

eps = 1.0d-5; diffnorm = 1.0d+2; iter = 0
!do while ( iter < max_iter .and. diffnorm > eps )
!  call sweep2d_domain( u, f, n, unew )
!  call sweep2d_domain( unew, f, n, u )
!  diffnorm= diff( u, unew, n )
  iter = iter + 1
end do
if ( diffnorm < eps ) then
  print *, 'CONVERGENCE: ', iter, ' Iters '
  open (unit=9, file='s_jacobi.dat', form='unformatted', status='UNKNOWN')
  write(9) u(0:n+1,0:n+1)
else
  print *, 'NO CONVERGENCE: ', iter
endif
stop
end

subroutine init2d_domain( u, unew, f, n )
  integer n, i, j
  real (kind=8) u(0:n+1, 0:n+1), unew(0:n+1, 0:n+1), f(0:n+1, 0:n+1)
  real (kind=8) h, pi, sin_hx, exp_hx
  h = 1.0d0/(n + 1); pi = 3.1415926535897932d0
!”$OMP PARALLEL DO PRIVATE(i,j) SHARED(u,unew,f,n)
  do j=0, n+1
    do i=0, n+1
      u(i,j) = 0.0d0
      unew(i,j) = 0.0d0
      f(i,j) = 0.0d0
    end do
  end do
!”$OMP END PARALLEL DO
!”$OMP PARALLEL DO PRIVATE(i,sin_hx,exp_hx) !$OMP FIRSTPRIVATE(h,pi) SHARED(u,unew)
  do i=0, n+1
    sin_hx = sin(pi * h * i)
    u(i,0) = sin_hx
    unew(i,0) = sin_hx
    exp_hx = exp( h * i)
    u(i,n+1) = sin_hx * exp_hx
    unew(i,n+1) = sin_hx * exp_hx
  end do
!”$OMP END PARALLEL DO
!
return
end

program OpenMP_jacobi1d

subroutine sweep2d_domain( u, f, n, unew )
!--------------------------------------------------
! Does a full Jacobi iteration sweep across the
! square 2-d mesh 5-point stencil finite difference
! method used
!--------------------------------------------------
integer n, i, j
real (kind=8) u(0:n+1,0:n+1), unew(0:n+1,0:n+1),
f(0:n+1,0:n+1)
real (kind=8) h
!
 h = 1.0d0 / (n+1)
!$OMP PARALLEL DO PRIVATE(i,j) FIRSTPRIVATE(h)
!$OMP+SHARED(u,unew,f)
do j=1, n
  do i=1, n
    unew(i,j) = 0.25 * ( (u(i-1,j)+u(i,j+1)+ &
    u(i,j-1)+u(i+1,j)) - &
    h * h * f(i,j) )
  end do
end do
!$OMP END PARALLEL DO
!
return
end

real (kind=8) function diff( u, unew, n )
integer n
real (kind=8) u(0:n+1,0:n+1), unew(0:n+1,0:n+1)
real (kind=8) sum
integer i, j
!
sum = 0.0d0
!$OMP PARALLEL DO SHARED(u,unew) PRIVATE(i,j) &
!$OMP+REDUCTION(+:sum)
do j=1,n
  do i=1,n
    sum = sum + (u(i,j) - unew(i,j)) ** 2
  end do
end do
!$OMP END PARALLEL DO
!
diff = sum
return
end
Benchmarking openmp_jacobi1d_f.exe

```bash
ifort -openmp -O3 -o openmp_jacobi1d_f.exe openmp_jacobi1d.f90

> export OMP_NUM_THREADS=1
> time ./openmp_jacobi1d_f.exe

Enter Problem (Square Mesh) Size   - 800
Enter Maximum Number of Iterations - 50000
CONVERGENCE:   Iters= 16475  Seconds= 170

> export OMP_NUM_THREADS=2
> time ./openmp_jacobi1d_f.exe

Enter Problem (Square Mesh) Size   - 800
Enter Maximum Number of Iterations - 50000
CONVERGENCE:   Iters= 16475  Seconds= 140

> export OMP_NUM_THREADS=4
> time ./openmp_jacobi1d_f.exe

Enter Problem (Square Mesh) Size   - 800
Enter Maximum Number of Iterations - 50000
CONVERGENCE:   Iters= 16475  Seconds= 40

> export OMP_NUM_THREADS=8
> time ./openmp_jacobi1d_f.exe

Enter Problem (Square Mesh) Size   - 800
Enter Maximum Number of Iterations - 50000
CONVERGENCE:   Iters= 16475  Seconds= 17
program p_jacobild

! p_jacobild.f90 - a solution to the 2-dim Poisson problem using
! Jacobi iteration on a 1-d domain decomposition. The domain de-
! composition is carried out along the x-axis, i.e., along the
! columns of arrays the u(i*h,j*h) (and unew(i*h,j*h)).
!
! _ 2 _ 2  x,y in [ 0, 1 ]
! d u  d u
! Problem: --- + --- = f(x,y) = (x^2 + y^2) * cos(x*y) - cos(pi * x)
! _ 2 _ 2  *** Boundary Conditions ***
! dx  dy  y = 0: u(x, 0) = (cos(pi*x) - pi^2)/(pi^2)
! y = 1: u(x, 1) = (cos(pi*x) - pi^2 *cos(x))/(pi^2)
! x = 0: u(0, y) = 1/(pi^2) - 1
! x = 1: u(1, y) = -(1/(pi^2) + cos(y))
!
!
!
!
!
!
!

Programmer: Spiros Vellas
!
The following source file is an extensive modification of source files
that accompany the textbook Using MPI by Gropp and others.
!
Compilation & Execution on HYDRA:
mpxl90_r -q64 -qsuffix=f=f90 -qarch=auto -O3 -o p_poisson_ld.exe ¥
p_poisson_ld.f90
!
export OBJECT_MODE=64
!
po e p_poisson_ld.exe -shared_memory yes -procs n -resd [no|yes] ¥
    ... -tasks_per_node # # -euilib us -nodes # #
!
The size of square domain is read by processor 0 and broadcast to
all other processors. The Jacobi iteration is run until the
change in the values of successive iterations is small or a max-
imum number of iterations is reached.

}***********************************************************************
USE MPI

real (kind=8), allocatable :: u(:,,:), unew(:,,:), f(:,:)
integer nx, ny, status(mpi_status_size)
integer myid, numprocs, commld, info, amode, fh, ierr
integer iprevid, inextid, s, e, iter, local_array_size
integer(kind=mpi_offset_kind) fl_offset
real (kind=8) diff, diffnorm, mydiff, t1, t2

! call MPI_INIT( ierr )
call MPI_COMM_RANK( MPI_COMM_WORLD, myid, ierr )
call MPI_COMM_SIZE( MPI_COMM_WORLD, numprocs, ierr )
if (myid == 0) then
  print *, 'Enter Problem (Square Mesh) Size   - '; read *, nx
  if ( nx < 100 ) &
    call MPI_ABORT(MPI_COMM_WOLD, -1, ierr)
  print *, 'Enter Maximum Number of Iterations - '
  read *, max_iter
  if ( max_iter < 100 ) call MPI_ABORT(MPI_COMM_WOLD, -1, ierr)
endif
!
call MPI_BARRIER(MPI_COMM_WORLD, ierr)
t1 = MPI_WTIME()
!
call MPI_BCAST(nx, 1, MPI_INTEGER, 0, MPI_COMM_WORLD, ierr)
call MPI_BCAST(max_iter, 1, MPI_INTEGER, 0, MPI_COMM_WORLD, ierr)
!
ny = nx ! make a square mesh to minimize code clutter
call BLOCK_MAP(1, ny, numprocs, myid, s, e )

inextid = myid + 1 ! right neighbor process
iprevid = myid - 1 ! left neighbor process
if ( myid == numprocs - 1) inextid = MPI_PROC_NULL ! No right neighbor
if ( myid == 0 ) iprevid = MPI_PROC_NULL ! No left neighbor
!
allocate( u(0:nx+1, s-1:e+1), unew(0:nx+1, s-1:e+1), &
         f(1:nx,s:e), stat = ierr )
if ( ierr > 0 ) then
    print *, '--- MPI_ABORT: Allocate Error ---'
    call MPI_ABORT(MPI_COMM_WORLD, -1, ierr)
endif
!
call init1d_domain( u, unew, f, nx, ny, s, e ) ! initialize domain
!
eps = 1.0d-5; iter = 0; diffnorm = 1.0d+2; comm1d = MPI_COMM_WORLD
!
do while ( iter < max_iter .and. diffnorm > eps )
    
call exchng1d( u, nx, s, e, comm1d, iprevid, inextid)
    call sweep1d( u, f, nx, s, e, unew )
    call exchng1d( unew, nx, s, e, comm1d, iprevid, inextid)
    call sweep1d( unew, f, nx, s, e, u )
!
    mydiff = diff( u, unew, nx, s, e )

    call MPI_Allreduce( mydiff, diffnorm, 1, MPI_REAL8, &
                        MPI_SUM, MPI_COMM_WORLD, ierr )
! Each proc now has the same value for diffnorm
    iter = iter + 1
end do
!
call MPI_BARRIER(MPI_COMM_WORLD, ierr)
t2 = MPI_WTIME()
if ( myid == 0 ) then
    if ( diffnorm < eps ) then
        print *, 'CONVERGENCE: ','Procs=',numprocs,' Size=',nx,'x',nx,' Iters=', 2*iter," & WC Seconds=", t2-t1
    else
        print *, 'NO CONVERGENCE: ','Procs=',numprocs,' Size=',nx,'x',nx,' Iters=', 2*iter," Diffnorm="*diffnorm
    endif
endif
endif
amode = ior(MPI_MODE_CREATE, MPI_MODE_WRONLY); info=0
info = MPI_INFO_NULL

! call mpi_file_open(comm, 'poisson1d.dat', amode, info, fh, ierr)
if ( ierr /= 0 ) then
   print *, '--- Error from mpi_file_open, ',ierr,' info=',info,' --'
endif
!
! Store whole mesh, including boundaries
!
local_array_size = (e-s+1) * (nx+2)
if (myid == 0 .OR. myid == numprocs-1) local_array_size = local_array_size + (nx+2)
fl_offset = 0; if (myid > 0) fl_offset = s * (nx+2) * sizeof(u(1,1))

if (myid == 0) then
   call mpi_file_write_at(fh, fl_offset, unew(0,0), local_array_size, &
                          MPI_REAL8, status, ierr)
else
   call mpi_file_write_at(fh, fl_offset, unew(0,s), local_array_size, &
                          MPI_REAL8, status, ierr)
endif
!
if ( ierr /= 0 ) then
   print *, '--- Error from mpi_file_write_at --'
endif
!
call mpi_file_close(fh, ierr)
!
deallocate( u, unew, f )
call MPI_FINALIZE(ierr)
stop
End
!================================================================================ END OF MAIN ================================
subroutine BLOCK_MAP(n1, n2, numprocs, myid, s, e )
!--------------------------------------------------------------------
! Assigns process myid a chunk amount of work between s and e
! It implements a "block" assignment of n total amount of work
!--------------------------------------------------------------------
!
integer n1, n2, numprocs, myid, s, e, chunk, res
!
chunk   = (n2-n1+1) / numprocs
res     = mod(n2-n1+1, numprocs)
if ( myid < res ) then  ! first res-1 id's get one extra
  chunk = chunk + 1
  s = n1 + myid * chunk
else
  s = n1 + res + myid * chunk
endif
e = min(s + chunk - 1, n2)
return
end
!====================================================================

subroutine init1d_domain( u, unew, f, nx, ny, s, e )
! -------------------------------------------------------------------
! Init rhs, f(x,y), and set boundary conditions along the x- and y-axis
!
! f(x,y) = (x^2 + y^2) * cos(x*y) - cos(pi * x)
!
! **** Boundary Conditions ****
! y = 0: u(x, 0) = (cos(pi*x) - pi^2)/(pi^2)
! y = 1: u(x, 1) = (cos(pi*x) - pi^2 *cos(x))/(pi^2)
! x = 0: u(0, y) = 1.0/(pi^2) - 1.0
! x = 1: u(1, y) = -(1.0/(pi^2) + cos(y))
!=====================================================================
integer nx, ny, s, e
real (kind=8) u(0:nx+1, s-1:e+1), unew(0:nx+1, s-1:e+1), f(1:nx,s:e)
real (kind=8) h, pi, pi_sq, x, y
integer i, j
!
  h = 1.0d0/(nx + 1); pi = 3.1415926535897932d0; pi_sq = pi*pi
!
  u(0:nx+1, s-1:e+1) = 0.0d0; unew(0:nx+1, s-1:e+1) = 0.0d0
!
  do j=s, e
    y = j*h
    do i=1, nx
      x = i*h
      f(i,j) = (x*x + y*y) * cos(x*y) - cos(pi*x)
    end do
  end do
!
  if ( s == 1 ) then !init along x-axis; y=0
    do i=0, nx+1
      x = i*h
      u(i, 0) = (cos(pi*x) - pi_sq) / pi_sq
      unew(i, 0) = u(i, 0)
    end do
  end if
!
  if ( e == ny ) then ! init along x-axis; y=1: u(i*h, ny+1)
    do i=0, nx+1
      x = i*h
      u(i, ny+1) = (cos(pi*x) - pi_sq * cos(x)) / pi_sq
      unew(i, ny+1) = u(i, ny+1)
    end do
  end if
!
  u(0,s-1:e+1) = 1.0d0/pi_sq - 1.0d0 ! init along y-axis; x=0
  unew(0,s-1:e+1) = u(0,s-1:e+1)
!
  do j=s-1, e+1 ! init along y-axis; x=1; N.B. min(s)=1
    y = j*h
    u(nx+1,j) = - (1.0d0/pi_sq + cos(y))
    unew(nx+1,j) = u(nx+1,j)
  end do
!
return
end
subroutine exchng1d(u, nx, s, e, comm1d, iprevid, inextid)
!
! Each process executes two sends and two receives. It sends its border
! columns, u(1:nx, s) and u(1:nx, e) correspondingly to its left and
! right neighbors. The same process also receives two columns. One from
! its left and one from its right neighbors. These columns fit in the
! "ghost" columns s-1, and e+1.
! *--------------------------------------------------------------------
!
USE MPI

integer nx, s, e, comm1d, iprevid, inextid
real (kind=8) u(0:nx+1,s-1:e+1)
integer status_array(MPI_STATUS_SIZE,4), ierr, req(4)
!
call MPI_IRECV(u(1,s-1), nx, MPI_REAL8, iprevid, 0, comm1d, req(1), ierr)
call MPI_IRECV(u(1,e+1), nx, MPI_REAL8, inextid, 1, comm1d, req(2), ierr)

call MPI_ISEND(u(1,e), nx, MPI_REAL8, inextid, 0, comm1d, req(3), ierr)
call MPI_ISEND(u(1,s), nx, MPI_REAL8, iprevid, 1, comm1d, req(4), ierr)
!
call MPI_WAITALL(4, req, status_array, ierr)
return
end
!
--------------------------------------------------------------------
subroutine sweep1d(u, f, n, s, e, unew)
!
! Perform a 2-dim (x-y) Jacobi sweep for a 1-d (column) decomposition
! of the x-y domain. Sweep/update from u into unew
! *--------------------------------------------------------------------

integer n, s, e
real (kind=8) u(0:n+1,s-1:e+1), unew(0:n+1,s-1:e+1), f(0:n+1,s-1:e+1)
!
integer i, j
real (kind=8) h
!
h = 1.0d0 / dble(n+1)
do j=s, e
  do i=1, n
    unew(i,j) = 0.25 * ( (u(i-1,j)+u(i,j+1)+u(i,j-1)+u(i+1,j)) - &
                        h * h * f(i,j) )
  end do
end do
return
real (kind=8) function diff( u, unew, n, s, e )
!
! Computes the convergence "norm"
! -------------------------------------------------------------------
integer n, s, e
real (kind=8) u(0:n+1, s-1:e+1), unew(0:n+1, s-1:e+1)
!
real (kind=8) sum
integer i, j
!
sum = 0.0d0
do j=s, e
  do i=1, n
    sum = sum + (u(i,j) - unew(i,j)) ** 2
  end do
end do
!
diff = sum
return
end