Programming Issues

Interconnection Network Architecture and

Issues

Memory System Architecture and Programming

Basic Shared Memory Programming Paradigm

Basic Message Passing Programming Paradigm

What's Next?
Distributed Memory Machines

- Each processor has local memory with local address space
- Only way to exchange data is using explicit message passing
- Time taken for message depends on the relative locations of the source and destination processors (How strongly depends on system)
- Performance of a parallel program determined by how well the location of data matches its use
- Detailed analysis of initial data location and subsequent motion is required.
small compared to memory.

local memory sizes, i.e., a cache is assumed to be very
with much longer time constants and much larger
Simulations to cache problem of data locality but

preferred version?

partitioning or reshuffle data to get the library's
you design routines with suboptimal initial
that can be called in many different situations. Do

This can be problematic when designing libraries


amongst users

In some machines subpartitions are also time shared

machine to execute their program

shared, i.e., users obtain a subpartition of the

typically, distributed memory machines are space

Programming Models
do based on local data including processor ID
processor IDs, i.e., each processor figures out what to
same program, computation is shared based on
the SPMD model, every processor executes the

Data (SPMD) programming model popular

These factors make the Single Program Multiple
user at the same time
the processors in the subpartition are available to the

However, such sharing uses same scheduling, i.e., all
True distributed computing is MPMD

- model

Have to take the MPMD model using the SPMD

- systems do not support the MPMD model directly

- Most of the current distributed memory operating

- programs and share the computation

- Under this model, processors execute different

- MPMD model

- MPMD machine is the Multiple Program Multiple Data

- Another programming model for distributed memory

-
combination of both often occurs in practice •

or "task" and data circulates
each processor performs one type of "command"

control parallel •

same computation

– data partitioned across processors and each does
data parallel •

– two typical ways of looking at this programming:
Interprocessor Communication

- Communicating processes form arbitrary graphs
  - In unstructured communications
    - Processes form a regular structure such as a ring
- In structured communication, communicating processes, possibly all other processors communicate with many with a small set of neighbors; in global
- In local communications, processors communicate amongst themselves when they share computation
- Processors usually need to communicate data
Local and structured is preferred.

- computed at runtime and may be highly variable
- dynamic communications may be determined by data
  - In contrast, the identity of communication partners in
  - In static communication, the identity of
  - network of the system
  - relationship to the underlying interconnection

Structured communication need not have any
Synchronous at some level

static, and structured communication (often
most often used communication paradigm is local;

synchronous at the clock cycle level of coupling
applied at several levels; e.g., systolic arrays are
synchronous/asynchronous distinction can be

simultaneously (or even at all in some cases)
require the communication processors to co-operate
contrast, asynchronous communication does not
processors execute in a coordinated fashion; in
synchronous communication, communication


Relationship between processors

- SEND-RECV pairs imply a producer-consumer

- The destination

- The RECV operation initiates message reception at

- The source

- The SEND operation initiates message transmission

- Interprocessor communication

- We often use matched SEND-RECV pairs for
blocking

- Typically SENDs are non-blocking and RECVs are

- Specified to assess choice.

- Exactly what "complete" means has to be

- Computation

  - Operation to complete and proceeds with further

    - Non-blocking: processor does not wait for

      - Completed

    - Blocking: processor waits until operation has

      - The SEND/RECV operations can be:
correct data and proceed with other computation it. On the other hand, the producer can just send the for the correct data from the producer before using the reason for this is that the consumer must wait processes involved implicit synchronization between the pair of A SEND/RECV pair used in this manner enforces
control of a processor.

the operating system is swapping tasks in and out of

arguments for non-blocking receives to avoid involving

emulating several virtual tasks on one processor

Also using the worker task point of view which is

receives.

asynchronous relaxation arguments for non-blocking

partial dependence satisfaction seen for example in

and non-blocking. (We will discuss this later.

this, for example, to what MPI considers blocking

addressed in real message passing libraries. Compare

non-blocking. It ignores details that must be

This is a general definition of blocking and

...
still be needed.

multiple versions for performance portability may
so

The standard defines function NOT performance

Development of the parallel software industry

set of routines to implement efficiently

- Providing hardware vendors with a well-defined

- Portability and ease-of-use

- Standard needed for:

  PICT

  Follow-on to portable libraries PVM, PA, EXPRESS,

  MPI - standard for explicit message passing

  Introduction to Message-Passing Interface (MPI)
Web support is extensive.

- Environmental query functions
- Support for application topologies
- Support for communication contexts
- Support for process groups
- Collective communication
- Point-to-point message passing

MPI provides: •
MPI\texttt{\_RECV}: Receive a message

MPI\texttt{\_SEND}: Send a message

MPI\texttt{\_COMM\_RANK}: Determine my process identifier

MPI\texttt{\_COMM\_SIZE}: Determine number of processes

MPI\texttt{\_FINISH}:

MPI\texttt{\_INITIALIZE}:

MPI\texttt{\_TEST}\,\texttt{\_success}:

to get started on writing useful codes.

MPI has over 125 functions, but you need to know only 6

\textbf{Basics of MPI}
The MPI SEND operation:

\[
\text{(handle)} \quad \text{datatype is the datatype of send buffer elements}
\]

\[
\text{count is the number of elements to send (integer)}
\]

\[
\text{but is the address of the send buffer}
\]

\[
\text{MPI-SEND}(\text{buf, count, datatype, dest, tag, comm})
\]

The MPI SEND call syntax:
only one context specified by MPI_COMM_WORLD.

For now we have

• comm identifies a group of processes
• tag is a message tag to identify the message
• dest is the process id of destination process
•
is actually a blocking send in the MPI sense.

- The data can be modified after the return. (This
  returns to processing the user program.

- Processor

availability of a message for the destination

- Uses the tag and dest arguments to record the
  space

- Copies the contents of the buffer into the system

- Allocates system space for the contents of the

- Allocates a system with buffering:

On the initiation of an MPI SEND, the source
tag is a message tag to identify the message (integer)

(source is the process id of source process (integer)

(handle)

data_type is the datatypes of send buffer elements

(count is the number of elements to receive (integer)

(buf is the address of the receive buffer

(tag, comm, status)

MPI_RECV(buf, count, data_type, source, 

The MPI_RECV call syntax:

The MPI_RECV operation
the status.

count of the message received can be retrieved from

status is status of the message. The source, tag and

communication context (handle)

common identifiers a group of processes and a
- Returns to processing the user program
- The user's buffer
  - The arrival of the message before copying it into
  - Message into the user's buffer; else it waits until
  - If the message has been received, it copies the
    availability of a message from the source processor
- Uses the tag and source arguments to check the
  processor:
  - On the initiation of an MPI_RECV, the destination
resolve this ordering problem. Determine the source of a received message and

Therefore, the status record can be used to

single destination.

(ordinate of two messages from single source to a

related to when they were sent. MPI preserves

arrive at the common destination in an order that is

Messengers from two different processors may not

Sources and Tags
decisions are driven by the messages received.
the message processing, i.e., the local code’s control
This assumes, of course, a case-based organization of
•
to allow the receive to proceed.
The source field can be set to MPI ANY SOURCE
•
and could be processed.
second message has arrived from a different source
if there is local work that could be done or if a
•
source arrives. This can have performance penalties
specific waiting until a message from a particular
the source argument in a receive can be used to
arrives.

specifics waiting until a message of a particular type
message types. The tag argument in a receive
Tags can be associated with messages to distinguish
not be produced.

exception condition or error condition might or might
example, some sort of message indicating an
known that a message will definitely be sent. For
a non-deterministic order. Also it may not ever be
produced the message may arrive at the sender in
receiver does not know a priori since the data needed
same type and may be produced in an order that the
Messages from a single source may not all be of the
Hexability to the simple SEND/RECV pair:

- Clearly, specific values or wild cards in the source receive to proceed.
- This can also be set to MPI-ANY-TAG to allow the...
comm (integer)

- and size is the number of processes in the group
- where comm is the communicator

MPI.COMM_SIZE(comm, size)

Obtaining size of partition:

Useful Auxiliary Primitives
(integer)

and \texttt{pid} is the process id in the group \texttt{comm}.

where \texttt{comm} is the communicator.

\texttt{MPI-COMM-RANK(comm, pid)}

\texttt{number}: Obtaining process IDs (not OS identification)
data is used as would be needed in a receive.

wild card type of receive. Note no information about
and return when satisfied. This is a more general
queue for a specific source/tag comm combination

MPI-PROBE{source,tag,comm,stats} query local

combination and return logical value.

query local queue for a specific source/tag comm

MPI-PROBE{source,tag,comm,stats} query local
message queues.

source and tag wild cards in receives. Best way is to

Implementing asynchronous communciation was aided by
Example in C

```c
{
    MPI_Init(NULL, NULL);
    /* Initialize MPI with no arguments. */

    for (int i = 0; i < 100; i++)
        MPI_Sendrecv(A[i], 100, MPI_FLOAT, 0, 0, MPI_COMM_WORLD, 10, 100, MPI_COMM_WORLD);
    /* Send and receive data between processes. */

    MPI_Finalize();
    /* Clean up MPI. */
}

main()

MPI_Init(&argc, &argv);
/* Initialize MPI with command line arguments. */

/* Local process ID */
int myid;
/* Number of processes */
int count;

float a[100];
/* Float array */

int b[100];
/* Integer array */

# include <stdio.h>
```
Same Example in FORTRAN
Note space is laid out for all processes.

Work determined by process ID

andFortran

Note slight differences in capitalization between C

Note need for explicit indication of call by address in

C

; MPI_REAL in Fortran

; MPI_datatypes - MPI_Status, MPI_FLOAT

Note MPI datatypes - MPI_Status, MPI_FLOAT


C compilation: cc code.c -lmpl

Fortran compilation: f90 -O3 code.f -lmpl

We will use the SGI O200S for now.

SGI Compilation
MPI-GET-COUNT (status, datatype, count, ierr)

The number of elements received is given by:

MPI-STAT-TYPE

In Fortran status is an integer array of size

– The message tag

– The source process for the message

provides information about:

Return Status Objects

Return Status Objects