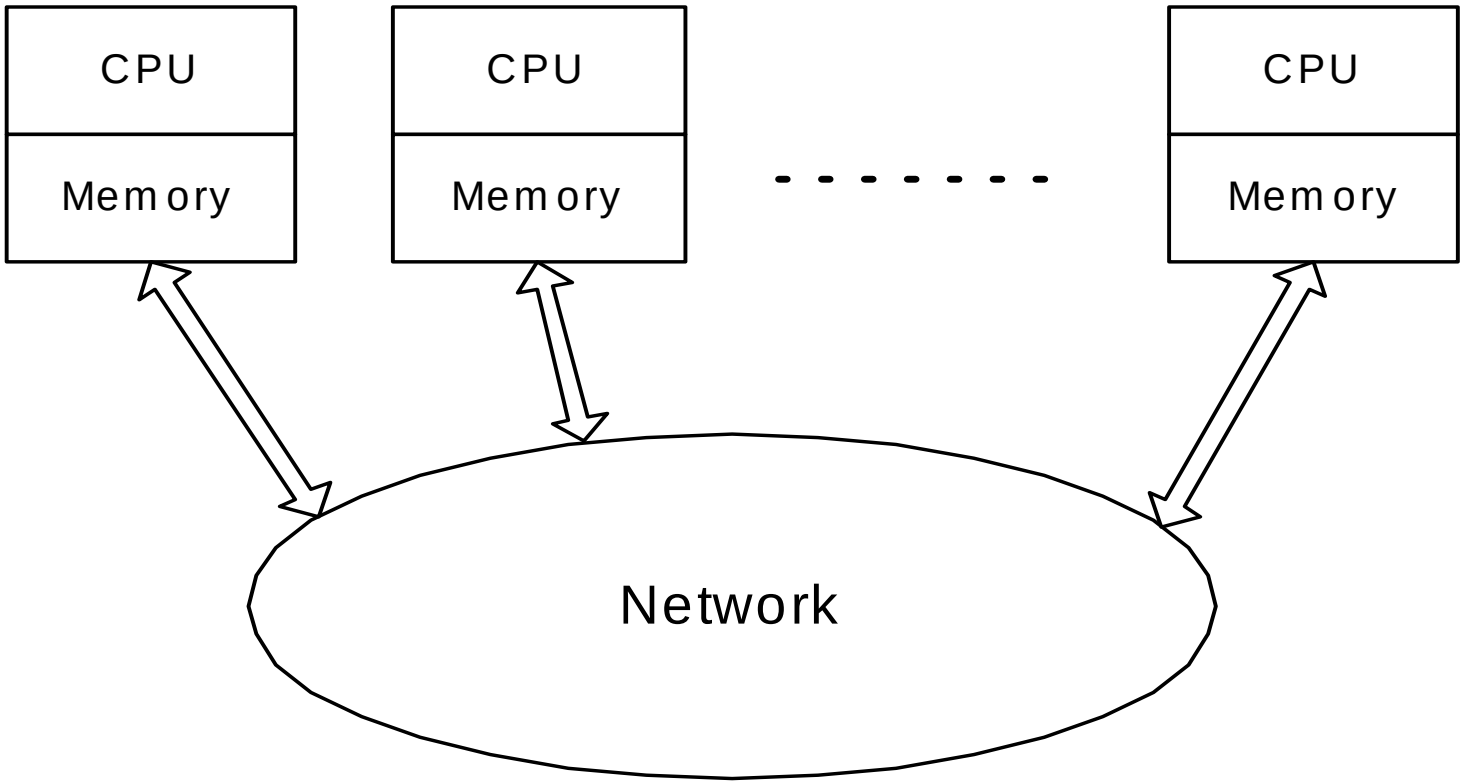
Distributed Memory Multiprocessors

Distributed Memory Multiprocessor



The MPP architecture

Programming Model

- Primary programming model
  - Parallel *processes*
    - each running on a separate processor
    - using message passing to communicate with the others
- Why send messages?
  - Some processes may use data computed by other processes
    - the data must be delivered from processes producing the data to processes using the data
  - The processes may need to synchronize their work
    - sending messages is used to inform the processes that some event has happened or some condition has been satisfied

MPP Architecture

- MPP architecture
  - Provides more parallelism than SMPs
    - SMPs rarely have more than 32 processors
    - MPPs may have hundreds and even thousands processors
- How significant is the performance potential of the MPP architecture?
  - Due to modern communication technologies, MPPs are a *scalable* parallel architecture
    - (p+1)-processor configuration executes "normal" message-passing programs faster than p-processor one for practically arbitrary p

MPP Architecture (ctd)

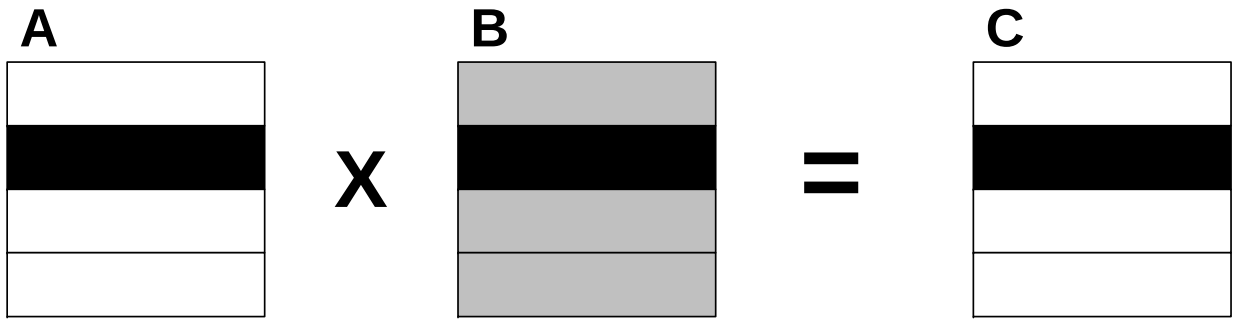
- MPP communication network
  - Must provide a communication layer that would be
    - fast
    - well balanced with the number and performance of processors
      - should be no degradation in communication speed even when all processors of the MPP simultaneously perform intensive data transfer operations
    - homogeneous
      - ensure the same speed of data transfer between any two processors of the MPP

MPP Architecture (ctd)

- Implementation of the MPP architecture
  - Parallel computer
  - Dedicated cluster of workstations
  - A real MPP implementation the ideal MPP architecture
    - a compromise between the cost and quality of its communication network
- Illustration of the scalability of the MPP architecture
  - Parallel multiplication of two dense *nxn* matrices on an ideal *p*-processor MPP: $\mathbf{C = A \times B}$

MPP Architecture (ctd)

- The A, B, and C matrices are evenly (and identically) partitioned into p horizontal slices
  - For the sake of simplicity, we assume that n is a multiple of p
  - There is one-to-one mapping between these slices and the processors
  - Each processor is responsible for computing its C slice
- To compute its C slice, each processor requires all elements B
  - Receives from each of p-1 other processors n^2/p matrix elements



MPP Architecture (ctd)

- Contribution of computation in the total execution time

$$t_{comp} = \frac{t_{proc} \times n^3}{p}$$

- The cost of transfer of a single horizontal slice between two processors

$$t_{slice} = t_s + t_e \times \frac{n^2}{p}$$

MPP Architecture (ctd)

- Assume
  - A double port model
  - A processor sends its slice to other processors in *p*-1 sequential steps
- Then, the per-processor communication cost

$$t_{comm} = (p-1) \times t_{slice} \approx t_s \times p + t_e \times n^2$$

- Assume that communications and computations do not overlap

$$t_{total} \approx t_{proc} \times \frac{n^3}{p} + t_s \times p + t_e \times n^2$$

MPP Architecture (ctd)

- What restrictions must be satisfied to ensure scalability?
  - *Firstly*, there must be speedup when upgrading the MPP from 1- to 2-processor configuration, that is,

$$t_{proc} \times n^3 - (t_{proc} \times \frac{n^3}{2} + t_s + t_e \times \frac{n^2}{2}) = t_{proc} \times \frac{n^3}{2} - t_s - t_e \times \frac{n^2}{2} > 0$$

or

$$n^3 > 2 \times \frac{t_s}{t_{proc}} + \frac{t_e}{t_{proc}} \times n^2$$

MPP Architecture (ctd)

– As typically $\frac{t_s}{t_{proc}} \sim 10^3$ $\frac{t_e}{t_{proc}} \sim 10^1$

the above inequality will be comfortably satisfied if $n>100$

– *Secondly*, t_{total} must be a monotonically decreasing function of p, that is,

$$\frac{\partial t_{total}}{\partial p} = t_s - t_{proc} \times \frac{n^3}{p^2} < 0 \qquad \text{or} \qquad \frac{t_s}{t_{proc}} \times \left(\frac{p}{n}\right)^2 \times \frac{1}{n} < 1$$

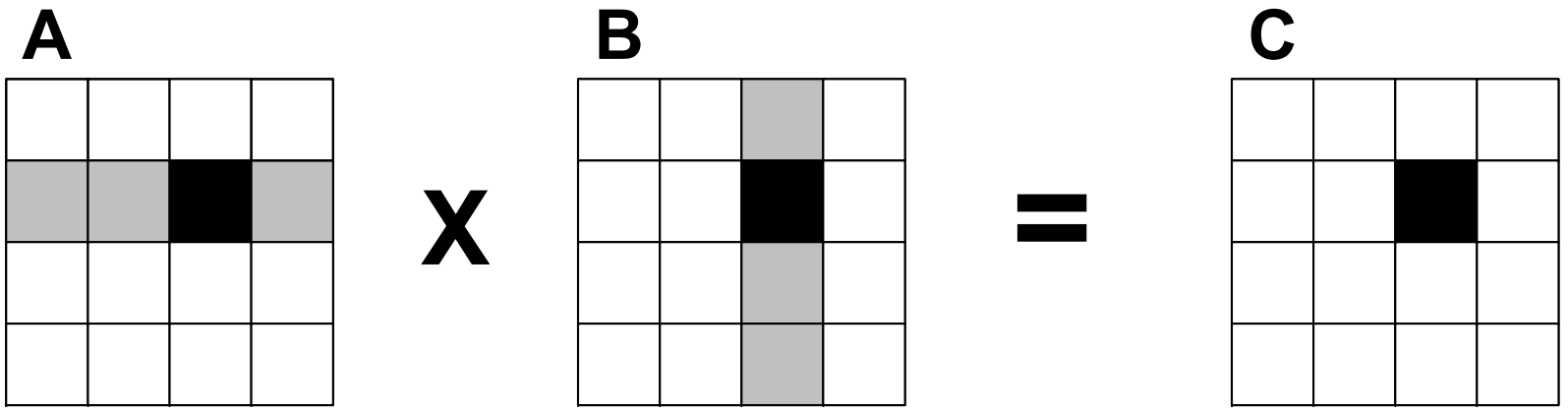
This will be true, if n is reasonably larger than p

MPP Architecture (ctd)

- The MM algorithm can be improved by using a 2-dimensional decomposition, when matrices A, B, and C are identically partitioned into p equal $\frac{n}{\sqrt{p}} \times \frac{n}{\sqrt{p}}$ squares
  - For the sake of simplicity, we assume that p is a square number, and that n is a multiple of $\sqrt{p}$
  - Each row and each column contain $\sqrt{p}$ squares
  - 1-to-1 mapping between the squares and the processors
  - Each processor is responsible for computing its C square

MPP Architecture (ctd)

- To compute its C slice, each processor requires the corresponding row of squares of the A matrix and column of squares of the B matrix
  - Receives from each of $\sqrt{p}-1$ horizontal and $\sqrt{p}-1$ vertical neighbours n^2/p matrix elements



MPP Architecture (ctd)

- The total per-processor communication cost

$$t_{comm} = 2 \times (\sqrt{p} - 1) \times (t_s + t_e \times \frac{n^2}{p}) \approx 2 \times t_s \times \sqrt{p} + 2 \times t_e \times \frac{n^2}{\sqrt{p}}$$

- The total execution time of that parallel MM algorithm

$$t_{total} \approx t_{proc} \times \frac{n^3}{p} + 2 \times t_s \times \sqrt{p} + 2 \times t_e \times \frac{n^2}{\sqrt{p}}$$

- This is considerably less than in the 1D algorithm

MPP Architecture (ctd)

- Some further improvements can be made to achieve
  - Overlapping communications and computations
  - Better locality of computations
  - The resulting carefully designed 2D algorithm will be efficient and scalable, practically, for any reasonable task size and number of processors
- The ideal MPP is scalable when executing carefully designed and highly efficient parallel algorithms
- Under quite weak, reasonable and easily satisfied restrictions even very straightforward parallel algorithms make the MPP architecture scalable

Performance Models

- We used 3 parameters t_{proc}, t_s, and t_e, and a straightforward linear communication model to describe an MPP, called the *Hockney model*
- The model is satisfactory
  - To demonstrate scalability
  - For performance analysis of *coarse-grained* parallel algorithms with
    - simple structure of communications
    - mainly long messages rarely sent during the communications

Performance Models (ctd)

- The accuracy of this model is unsatisfactory
  - To predict performance of message-passing algorithms with
    - non-trivial communication structure
    - frequent communication operations
    - mainly short messages
    - communications prevailing over computations

LogP

- LogP
  - A more realistic model of the MPP architecture
  - Still simple but sufficiently detailed
    - Allows accurate prediction of performance of message-passing algorithms with fine-grained communication structure
  - Under the model the processors communicate by point-to-point short messages
  - The model specifies the performance characteristics of the interconnection network
    - does not describe the structure of the network

LogP (ctd)

- The main parameters of the LogP model
  - *L*: An upper bound on the *latency*
    - The delay, incurred in sending a message from its source processor to its target processor
  - *o*: The *overhead*
    - The length of time that a processor is engaged in the transmission or reception of each message; during this time the processor cannot perform other operations

  => Point-to-point communication time *L*+*2*x*o*

LogP (ctd)

- *g*: The *gap* between messages
  - The minimum time interval between consecutive message transmissions or consecutive message receptions at a processor
    - $1/g$ corresponds to the available per-processor communication bandwidth for short messages
  - Needed to model collective communications
- *P*: The number of processors
- Unit time (called *processor cycle*) is assumed for local operations
  - *L*, *o*, and *g* are measured as multiples of the unit

$\lfloor L/g \rfloor$

LogP (ctd)

- The LogP network has a finite capacity
  - At most L/g messages can be in transit from any processor or to any processor at any time
- The LogP model is asynchronous
  - Processors work asynchronously
  - The latency experienced by any message is unpredictable
    - but is bounded above by L in the absence of stalls
  - Because of variations in latency, the messages directed to a given target processor may not arrive in the same order as they are sent

$\lfloor L/g \rfloor$

LogP (ctd)

- LogP can predict the execution time of communication operations
  - Broadcasting a single data unit from one processor to P-1 others using the flat tree topology
    - $L+2xo+(P-2)xg$, if $g>o$
    - $L+Pxo$, otherwise

LogP (ctd)

- Not all parameters are always equally important
  - Often it is possible to ignore one or more parameters and work with a simpler model
  - The bandwidth and capacity limits
    - can be ignored in algorithms that communicate data infrequently
  - The latency
    - may be disregarded in algorithms where messages are sent in long streams, which are pipelined through the network
      - message transmission time is dominated by inter-message gaps
  - The gap
    - can be eliminated for MPPs with the overhead dominating the gap

LogP(ctd)

- Example 1. Optimal broadcasting a single data unit from one processor to P-1 others
  - All processors that have received the data unit transmit it as quickly as possible
  - No processor receives more than one message
  - The root begins transmitting the data unit at time 0
    - The first data unit
      - enters the network at time o
      - takes L cycles to arrive at the destination
      - received by the processor at time $L+2xo$

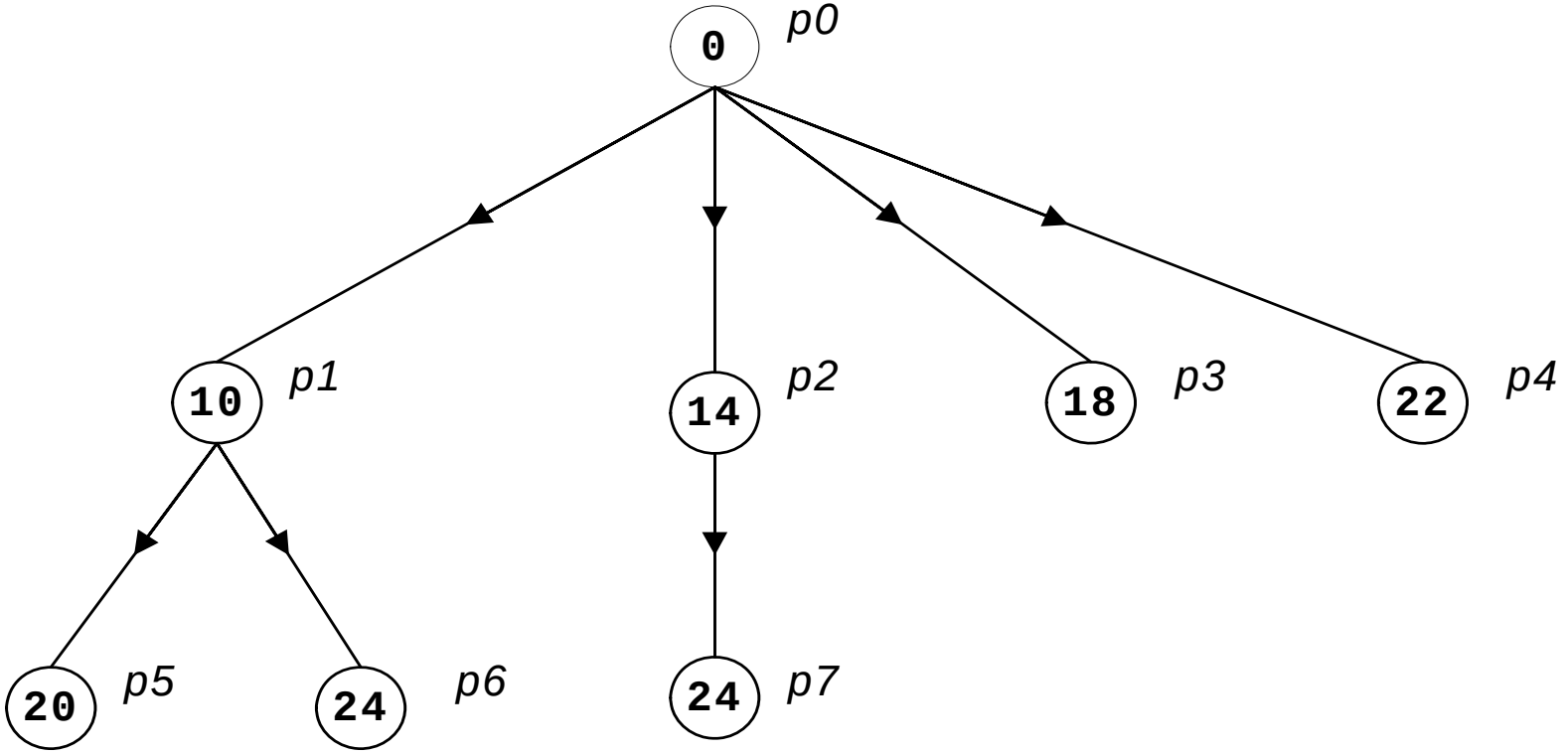
LogP (ctd)

- Meanwhile, the root initiates transmission to other processors
  - at time g, $2xg$,… ($o \leq g$ is assumed)
  - each processor acts as the root of a smaller broadcast tree

– The optimal broadcast tree for P processors is unbalanced with the fan-out at each node determined by the relative values of L, o, and g

LogP (ctd)

The optimal broadcast tree for $P=8$, $L=6$, $o=2$, $g=4$



LogP(ctd)

- Example 2. Optimal parallel summation
  - Summation of as many values as possible within a fixed amount of time *T*
  - The communication pattern is a tree
    - Each processor sums a set of the elements and then transmits the result to its parent
      - The set to be summed include original elements stored in its memory and partial results received from its children in the communication tree

LogP(ctd)

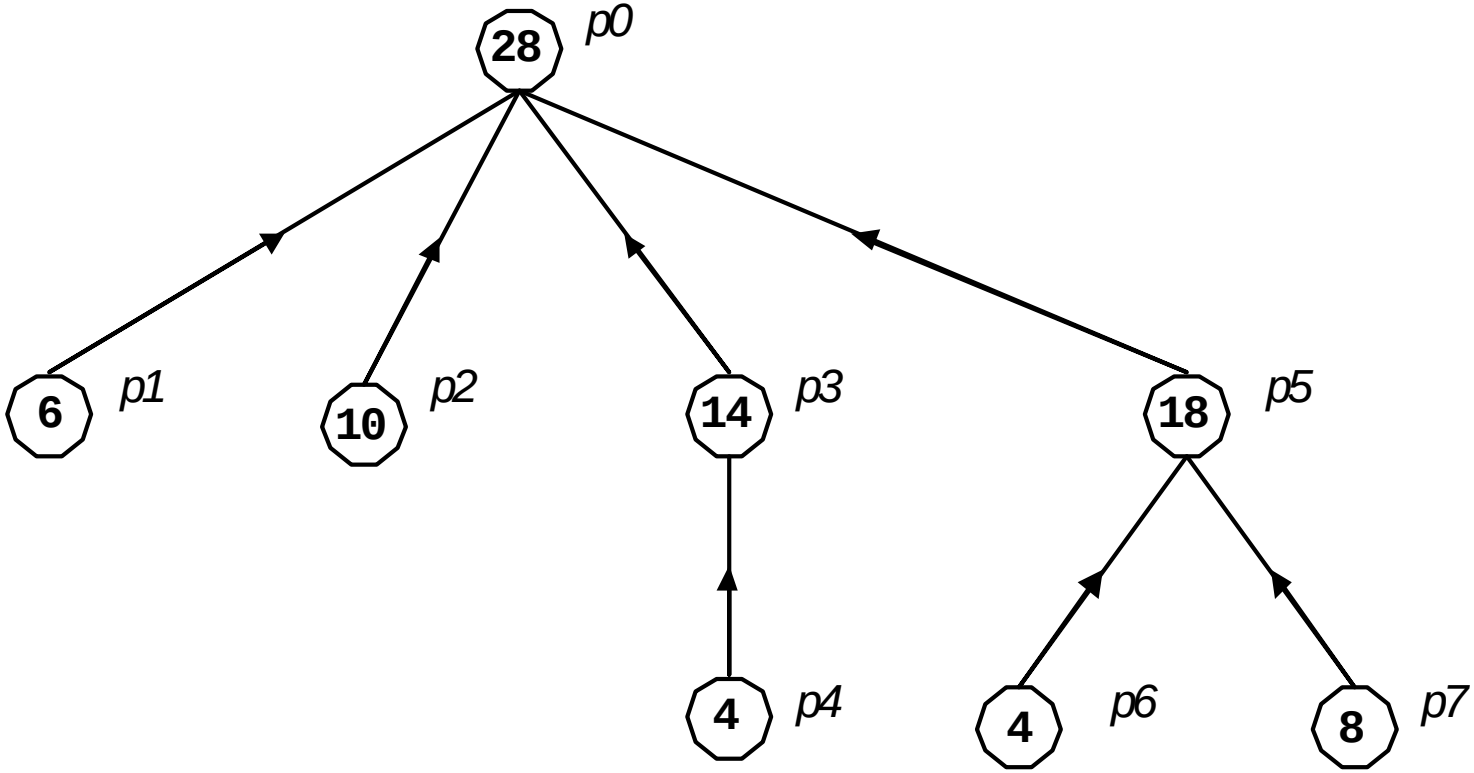
- Example 2. Optimal parallel summation (ctd)
  - Optimal schedule of communication events
    - If $T \leq L+2xo$
      - Sum $T+1$ values on a single processor
    - Else
      - At time $T-1$, the root adds a value computed locally to a value received from another processor
      - This remote processor
        - » must have sent the value at time $T-1-L-2xo$
        - » is the root of an optimal summation tree within time $T-1-L-2xo$

LogP(ctd)

- Example 2. Optimal parallel summation (ctd)
  - The local value must be produced at time $T–1–o$
  - The root can receive a message every g cycles => its children should complete their summations at times $T–(2xo+L+1)$, $T–(2xo+L+1+g)$, $T–(2xo+L+1+2xg)$, …
  - The root can perform $g–o–1$ additions between messages
  - Since a processor invests o cycles in receiving a partial sum from a child, all transmitted partial sums must represent at least o additions

LogP (ctd)

The optimal summation tree for $T=28$, $L=5$, $o=2$, $g=4$



Performance Models (ctd)

- LogP assumes that all messages are small
- LogGP is a simple extension of LogP dealing with longer messages
  - One more parameter *G*
    - the *gap per byte* for long messages
    - The reciprocal of *G* characterizes the available per-processor communication bandwidth for long messages

  => Point-to-point communication time:

  $L+2xo+(m-1)xG$

Estimation of Performance Models

- The models only make sense if their parameters can be accurately estimated
- The overhead o
  - Can be found directly on the sender side by measuring the execution time of sending a message of one byte
  - The value of o is obtained by averaging the results of a sufficient number of tests

Estimation of Performance Models (ctd)

- The latency L
  - Can be found from the average execution time $RTT(1)$ of a roundtrip, consisting of the sending and receiving of a message of 1 byte, given $RTT(1)=2xL+4xo$

Estimation of Performance Models (ctd)

- The gap function $g(m)$
  - Can be found from the execution time $s_n(m)$ of sending without reply a large number n of messages of size m
    - $g(m) \approx s_n(m) / n$
  - $g=g(1)$
  - $G=g(m)$ for a sufficiently large m

Performance Models (ctd)

- LogP and its extensions
  - A good mathematical basis for designing portable parallel algorithms efficiently running on a wide range of MPPs
    - via parametrising the algorithms with the parameters of the models

Optimising Compilers

Optimising Compilers

- MPPs are much far away from the serial scalar architecture than VPs, SPs, or SMPs
  - Optimising C or Fortran 77 compilers for MPPs would have to have intellect
    - To automatically generate an efficient message-passing code using the serial source code as specification of its functional semantics
  - No industrial optimising C or Fortran 77 compilers for MPPs
  - A small number of experimental research compilers
    - PARADIGM
    - Far away from practical use
- Basic programming tools for MPPs
  - Message-passing libraries
  - High-level parallel languages

Message-Passing Libraries

Message-Passing Libraries

- Message-passing libraries directly implement the message-passing parallel programming model
  - The basic paradigm of message passing is the same in different libraries (PARMACS, Chameleon, CHIMP, PICL, Zipcode, p4, PVM, MPI, etc)
  - The libraries just differ in details of implementation
- The most popular libraries are
  - MPI (Message-Passing Interface)
  - PVM (Parallel Virtual Machine)
  - Absolute majority of existing message-passing code is written using one of the libraries

Message-Passing Libraries

- We outline MPI
  - Standardised in 1995 as MPI 1.1
  - Widely implemented in compliance with the standard
    - all hardware vendors offer MPI
    - Free high-quality MPI implementations (Open MPI, MPICH)
  - Supports parallel programming in C and Fortran on all MPP architectures
    - including Unix and Windows NT platforms

Message-Passing Libraries

- MPI 2.0 is a set of extension to MPI 1.1 released in 1997
  - Fortran 90 and C++ bindings, parallel I/O, one-sided communications, etc
  - A typical MPI library used to fully implement MPI 1.1 and optionally supports some features of MPI 2.0
- MPI 2.2 approved in 2009
- We use the C interface to MPI 1.1 to present MPI

MPI

- An MPI program
  - A fixed number of processes
    - Executing their own code in their own address space
      - The codes need not be identical
    - Communicating via calls to MPI communication primitives
  - Does not specify
    - The number of processes
    - The allocation of the processes to physical processors
    - Such mechanisms are external to the MPI program
      - must be provided by particular MPI implementations
  - Uses calls to MPI inquiring operations to determine their total number and identify themselves in the program

MPI (ctd)

- Two types of communication operation
  - Point-to-point
    - Involves two processes, one of which sends a message and other receives the message
  - Collective
    - Involves a group of processes
      - barrier synchronisation, broadcast, etc

MPI (ctd)

- *Process group*
  - An ordered collection of processes, each with a *rank*
  - Defines the scope of collective communication operations
    - No unnecessarily synchronizing uninvolved processes
  - Defines a scope for process names in point-to-point communication operations
    - Participating processes are specified by their rank in the same process group
  - Cannot be build from scratch
    - Only from other, previously defined groups
      - By subsetting and supersetting existing groups
    - The base group of all processes available after MPI is initialised

MPI (ctd)

- *Communicators*
  - Mechanism to safely separate messages
    - Which don't have to be logically mixed, even when the messages are transferred between processes of the same group
  - A separate communication layer associated with a group of processes
    - There may be several communicators associated with the same group, providing non-intersecting communication layers
  - Communication operations explicitly specify the communicator
    - Messages transmitted over different communicators cannot be mixed (a message sent through a communicator is never received by a process not communicating over the communicator)

MPI (ctd)

- *Communicators* (ctd)
  - Technically, a communicator is implemented as follows
    - A unique tag is generated at runtime
      - shared by all processes of the group, with which the communicator is associated
      - attached to all messages sent through the communicator
      - used by the processes to filter incoming messages
  - Communicators make MPI suitable for writing parallel libraries

MPI (ctd)

- MPI vs PVM
  - PVM can't be used for implementation of parallel libraries
    - No means to have separate safe communication layers
    - All communication attributes, which could be used to separate messages, such as groups and tags, are user-defined
    - The attributes do not have to be unique at runtime
      - Especially if different modules of the program are written by different programmers

MPI (ctd)

- Example 1. The pseudo-code of a PVM application

```
extern Proc();
if(my process ID is A)
  Send message M with tag T to process B
Proc();
if(my process ID is B)
  Receive a message with tag T from process A
```

  - Does not guarantee that message M will not be intercepted inside the library procedure Proc
    - In this procedure, process A may send a message to process B
    - The programmer, who coded this procedure, could attach tag T to the message

MPI (ctd)

- Example 2. MPI solves the problem as follows

```
extern Proc();
  Create communicator C for a group including
       processes A and B
if(my process ID is A)
  Send message M to process B through
       communicator C
Proc();
if(my process ID is B)
  Receive a message from process A through
       communicator C
```

  - A unique tag attached to any message sent over the communicator C prevents the interception of the message inside the procedure Proc

Groups and Communicators

- A *group* is an ordered set of processes
  - Each process in a group is associated with an integer *rank*
  - Ranks are contiguous and start from zero
  - Groups are represented by opaque *group objects* of the type `MPI_Group`
    - hence cannot be directly transferred from one process to another
- A *context* is a unique, system-generated tag
  - That differentiates messages
  - The MPI system manages this differentiation process

Groups and Communicators (ctd)

- *Communicator*
  - Brings together the concepts of group and context
  - Used in communication operations to determine the scope and the "communication universe" in which an operation is to operate
  - Contains
    - an instance of a group
    - a context for point-to-point communication
    - a context for collective communication
  - Represented by opaque *communicator objects* of the type `MPI_Comm`

Groups and Communicators (ctd)

- We described *intra-communicators*
  - This type of communicators is used
    - for point-to-point communication between processes of the same group
    - for collective communication
- MPI also introduces *inter-communicators*
  - Used specifically for point-to-point communication between processes of different groups
  - We do not consider inter-communicators

Groups and Communicators (ctd)

- An initial pre-defined communicator `MPI_COMM_WORLD`
  - A communicator of all processes making up the MPI program
  - Has the same value in all processes
- The group associated with `MPI_COMM_WORLD`
  - The base group, upon which all other groups are defined
  - Does not appear as a pre-defined constant
    - Can be accessed using the function

```
int MPI_Comm_group(MPI_Comm comm, MPI_Group *group)
```

Groups and Communicators (ctd)

- Other group constructors
  - Explicitly list the processes of an existing group, which make up a new group, or
  - Do set-like binary operations on existing groups to construct a new group
    - Union
    - Intersection
    - Difference

Groups and Communicators (ctd)

- Function

```
int MPI_Group_incl(MPI_Group group, int n,
                   int *ranks, MPI_Group *newgroup)
```

creates `newgroup` that consists of the `n` processes in `group` with ranks `rank[0],..., rank[n-1]`. The process `k` in `newgroup` is the process `rank[k]` in `group`.

- Function

```
int MPI_Group_excl(MPI_Group group, int n,
                   int *ranks, MPI_Group *newgroup)
```

creates `newgroup` by deleting from `group` those processes with ranks `rank[0],..., rank[n-1]`. The ordering of processes in `newgroup` is identical to the ordering in `group`.

Groups and Communicators (ctd)

- Function

```
int MPI_Group_range_incl(MPI_Group group, int n,
                         int ranges[][3],
                         MPI_Group *newgroup)
```

assumes that `ranges` consist of triplets

$$(f_0, l_0, s_0),\ldots, (f_{n-1}, l_{n-1}, s_{n-1})$$

and constructs a group `newgroup` consisting of processes in `group` with ranks

$$f_0, f_0+s_0,\ldots, f_0+\left\lfloor \frac{l_0-f_0}{s_0} \right\rfloor \times s_0, \ldots, f_{n-1}, f_{n-1}+s_{n-1},\ldots, f_{n-1}+\left\lfloor \frac{l_{n-1}-f_{n-1}}{s_{n-1}} \right\rfloor \times s_{n-1}$$

Groups and Communicators (ctd)

- Function

```
int MPI_Group_range_excl(MPI_Group group, int n,
                         int ranges[][3], MPI_Group *newgroup)
```

constructs `newgroup` by deleting from `group` those processes with ranks

$$f_0, f_0+s_0, \ldots, f_0+\left\lfloor \frac{l_0-f_0}{s_0} \right\rfloor \times s_0, \ldots, f_{n-1}, f_{n-1}+s_{n-1}, \ldots, f_{n-1}+\left\lfloor \frac{l_{n-1}-f_{n-1}}{s_{n-1}} \right\rfloor \times s_{n-1}$$

The ordering of processes in `newgroup` is identical to the ordering in `group`

Groups and Communicators (ctd)

- Function

```
int MPI_Group_union(MPI_Group group1,
                    MPI_Group group2,
                    MPI_Group *newgroup)
```

creates `newgroup` that consists of all processes of `group1`, followed by all processes of `group2` .

- Function

```
int MPI_Group_intersection(MPI_Group group1,
                           MPI_Group group2,
                           MPI_Group *newgroup)
```

creates `newgroup` that consists of all processes of `group1` that are also in `group2`, ordered as in `group1` .

Groups and Communicators (ctd)

- Function

```
int MPI_Group_difference(MPI_Group group1,
                         MPI_Group group2,
                            MPI_Group *newgroup)
```

  creates `newgroup` that consists of all processes of `group1` that are not in `group2`, ordered as in `group1`.

- The order in the output group of a set-like operation
  - Determined primarily by order in the first group
    - and only than, if necessary, by order in the second group
  - Therefore, the operations are not commutative
    - but are associative

Groups and Communicators (ctd)

- Group constructors are *local* operations
- Communicator constructors are *collective* operations
  - Must be performed by all processes in the group associated with the existing communicator, which is used for creation of a new communicator
- The function

```
int MPI_Comm_dup(MPI_Comm comm,
                 MPI_Comm *newcomm)
```

  creates a communicator `newcomm` with the same group, but a new context

Groups and Communicators (ctd)

- The function

```
int MPI_Comm_create(MPI_Comm comm,
                    MPI_Group group,
                    MPI_Comm *newcomm)
```

creates a communicator `newcomm` with associated group defined by `group` and a new context

- Returns `MPI_COMM_NULL` to processes that are not in `group`
- The call is to be executed by all processes in `comm`
- `group` must be a subset of the group associated with `comm`

Groups and Communicators (ctd)

- The function

```
int MPI_Comm_split(MPI_Comm comm, int color,
                   int key, MPI_Comm *newcomm)
```

partitions the group of `comm` into disjoint subgroups, one for each nonnegative value of `color`

- Each subgroup contains all processes of the same color
  - `MPI_UNDEFINED` for non-participating processes
- Within each subgroup, the processes are ordered `key`
  - processes with the same key are ordered according to their rank in the parent group
- A new communicator is created for each subgroup and returned in `newcomm`
  - `MPI_COMM_NULL` for non-participating processes

Groups and Communicators (ctd)

- Two local operations to determine the process's rank and the total number of processes in the group
  - The function

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

    returns in `size` the number of processes in the group of `comm`
  - The function

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

    returns in `rank` the rank of the calling processes in the group of `comm`

Groups and Communicators (ctd)

- Example. See handout for an MPI program:
  - Each process first determines the total number of processes executing the program and its rank in the global group associated with `MPI_COMM_WORLD`
  - Then two new communicators are created
    - Containing processes with even global ranks
    - Containing processes with odd global ranks
  - Then each process determines its local rank in the group associated with one of the newly created communicators

Groups and Communicators (ctd)

- The function

```
int MPI_Init(int *argc, char ***argv)
```

  initializes the MPI environment

  - Must be called by all processes of the program before any other MPI function is called
  - Must be called at most once

- The function

```
int MPI_Finalize(void)
```

  cleans up all MPI state

  - Once the function is called, no MPI function may be called
    - even `MPI_Init`

Groups and Communicators (ctd)

- Group and communicator destructors
  - Collective operations
  - Mark the group or communicator for deallocation
    - actually deallocated only if there are no other active references to it
  - Function `int MPI_Comm_free(MPI_Comm *comm)`

    marks the communication object for dealocation
    - The handle is set to `MPI_COMM_NULL`
  - Function

    ```
    int MPI_Group_free(MPI_Group *group)
    ```

    marks the group object for dealocation
    - The handle is set to `MPI_GROUP_NULL`

Point-to-Point Communication

- Point-to-point communication operations
  - The basic MPI communication mechanism
  - A wide range of send and receive operations for different modes of point-to-point communication
    - Blocking and nonblocking
    - Synchronous and asynchronous, etc
- Two basic operations
  - A *blocking send* and a *blocking receive*
    - Predominantly used in MPI applications
    - Implement a clear and reliable model of point-to-point communication
    - Allow the programmers to write portable MPI code

Point-to-Point Communication (ctd)

- The function

```
int MPI_Send(void *buf, int n, MPI_Datatype datatype,
             int dest, int tag, MPI_Comm comm)
```

  implements a standard blocking send operation
  - Forms a message
  - Sends it to the addressee
- The *message* consists of
  - A data to be transferred
    - The data part may be empty (`n=0`)
  - An envelope
    - A fixed part of message
    - Used to distinguish messages and selectively receive them

Point-to-Point Communication (ctd)

- The *envelope* carries the following information
  - The *communicator*
    - specified by `comm`
  - The message *source*
    - implicitly determined by the identity of the message sender
  - The message *destination*
    - specified by `dest`
      - a rank within the group of `comm`
  - The message *tag*
    - specified by `tag`
      - Non-negative integer value

Point-to-Point Communication (ctd)

- The *data part* of the message
  - A sequence of `n` values of the type specified by `datatype`
  - The values are taken from a *send buffer*
    - The buffer consists of `n` entries of the type specified by `datatype`, starting at address `buf`
  - `datatype` can specify
    - A *basic datatype*
      - corresponds to one of the basic datatypes of the C language
    - A *derived datatype*
      - constructed from basic ones using datatype constructors provided by MPI

Point-to-Point Communication (ctd)

- `datatype`
  - An opaque object of the type `MPI_Datatype`
  - Pre-defined constants of that type for the basic datatypes
    - `MPI_CHAR` (corresponds to **`signed char`**)
    - `MPI_SHORT` (**`signed short int`**)
    - `MPI_INT` (**`signed int`**)
    - `MPI_FLOAT` (**`float`**)
    - etc
  - Basic datatypes
    - Contiguous buffers of elements of the same basic type
    - More general buffers are specified by using derived datatypes

Point-to-Point Communication (ctd)

- The function

```
int MPI_Recv(void *buf, int n, MPI_Datatype datatype,
             int source, int tag, MPI_Comm comm,
             MPI_Status *status)
```

implements a standard blocking receive operation

- The receive buffer
  - the storage containing n consecutive elements of type `datatype`, starting at address `buf`
  - the data part of the received message $\leq$ the receive buffer

Point-to-Point Communication (ctd)

- The selection of a message is governed by the message envelope
  - A message can be received only if its envelope matches the `source`, `tag` and `comm` specified by the receive operation
    - `MPI_ANY_SOURCE` for `source`
      - any source is acceptable
    - `MPI_ANY_TAG` for `tag`
      - any tag is acceptable
    - No wildcard for `comm`

Point-to-Point Communication (ctd)

- The asymmetry between send and receive operations
  - A sender always directs a message to a unique receiver
  - A receiver may accept messages from an arbitrary sender
  - A *push* communication
    - Driven by the sender
    - No *pull* communication driven by the receiver
- The `status` argument
  - Points to an object of the type `MPI_Status`
  - A structure containing at least three fields
    - `MPI_SOURCE`
    - `MPI_TAG`
    - `MPI_ERROR`
  - Used to return the *source*, *tag*, and *error code* of the received message

Point-to-Point Communication (ctd)

- `MPI_Recv` and `MPI_Send` are *blocking* operations
- Return from a blocking operation means that resources used by the operation is allowed to be re-used
  - `MPI_Recv` returns only after the data part of the incoming message has been stored in the receive buffer
  - `MPI_Send` does not return until the message data and envelope have been safely stored away
    - The sender is free to access and overwrite the send buffer
    - No matching receive may have been executed by the receiver
    - Message buffering decouples the send and receive operations

Point-to-Point Communication (ctd)

- `MPI_Send` uses the *standard* communication mode
  - MPI decides whether outgoing messages will be buffered
    - If buffered, the send may complete before a matching receive is invoked
    - If not, the send call will not complete until a matching receive has been posted, and the data has been moved to the receiver
      - buffer space may be unavailable
      - for performance reasons
- The standard mode send is *non-local*
  - An operation is non-local if its completion may require the execution of some MPI procedure on another process
    - Completion of `MPI_Send` may depend on the occurrence of a matching receive

Point-to-Point Communication (ctd)

- Three other (non-standard) send modes
  - *Buffered* mode
    - MPI must buffer the outgoing message
    - local
    - not safe (an error occurs if there is insufficient buffer space)
  - *Synchronous* mode
    - complete only if the matching receive operation has started to receive the message sent by the synchronous send
    - non-local
  - *Ready* mode
    - started only if the matching receive is already posted

Point-to-Point Communication (ctd)

- Properties of MPI point-to-point communication
  - A certain *order* in receiving messages sent from the same source is guaranteed
    - messages do not overtake each other
    - LogP is more liberal
  - A certain *progress* in the execution of point-to-point communication is guaranteed
    - If a pair of matching send and receive have been initiated on two processes, then at least one of these two operations will complete
  - No guarantee of *fairness*
    - A message may be never received, because it is each time overtaken by another message, sent from another source

Point-to-Point Communication (ctd)

- *Nonblocking* communication
  - Allows communication to overlap computation
    - MPI's alternative to multithreading
  - Split a one-piece operation into 2 sub-operations
    - The first just initiates the operation but does not complete it
      - Nonblocking *send start* and *receive start*
    - The second sub-operation completes this communication operation
      - *Send complete* and *receive complete*
  - Non-blocking sends can be matched with blocking receives, and vice-versa

Point-to-Point Communication (ctd)

- Nonblocking send start calls
  - Can use the same communication modes as blocking sends
    - standard, buffered, synchronous, and ready
  - A nonblocking ready send can be started only if a matching receive is posted
  - In all cases, the send start call is local
  - The send complete call acts according to the send communication mode set by the send start call

Point-to-Point Communication (ctd)

- A start call creates an opaque *request* object of the type `MPI_Request`
  - Identifies various properties of a communication operation
    - (send) mode, buffer, context, tag, destination or source, status of the pending communication operation

```
int MPI_Ixsend(void *buf, int n, MPI_Datatype datatype,
               int dest, int tag, MPI_Comm comm,
              MPI_Request *request)

int MPI_Irecv(void *buf, int n, MPI_Datatype datatype,
              int source,int tag, MPI_Comm comm,
             MPI_Request *request)
```

Point-to-Point Communication (ctd)

- The request is used later
  - To wait for its completion

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

  - To query the status of the communication

```
int MPI_Test(MPI_Request *request, int *flag,
             MPI_Status *status)
```

  - Additional complete operations
    - can be used to wait for the completion of any, some, or all the operations in a list, rather than having to wait for a specific message

Point-to-Point Communication (ctd)

- The remaining point-to-point communication operations
  - Aimed at optimisation of memory and processor cycles
  - `MPI_Probe` and `MPI_Iprobe` allow incoming messages to be checked for, without actually receiving them
    - the user may allocate memory for the receive buffer, according to the length of the probed message
  - If a communication with the same argument list is repeatedly executed within the loop
    - the list of arguments can be bound to a *persistent communication request* once, out of the loop, and, then, repeatedly used to initiate and complete messages in the loop

Collective Communication

- Collective communication operation
  - Involves a group of processes
    - All processes in the group must call the corresponding MPI communication function, with the matching arguments
  - The basic collective communication operations are:
    - Barrier synchronization across all group members
    - Broadcast from one member to all member of a group
    - Gather data from all group members to one member
    - Scatter data from one member to all members of a group
    - Global reduction operations
      - such as sum, max, min, or user-defined functions

Collective Communication (ctd)

- Some collective operations have a single originating or receiving process called the *root*
- A *barrier* call returns only after all group members have entered the call
- Other collective communication calls
  - Can return as soon as their participation in the collective communication is complete
  - Should not be used for synchronization of calling processes
- The same communicators can be used for collective and point-to-point communications

Collective Communication (ctd)

- The function

```
int MPI_Barrier(MPI_Comm comm)
```

  - Blocks the caller until all members of the group of `comm` have called it
  - The call returns at any process only after all group members have entered the call
- The function

```
int MPI_Bcast(void *buf, int count,
              MPI_Datatype datatype,
              int root, MPI_Comm comm)
```

  - Broadcasts a message from `root` to all processes of the group
  - The type signature obtained by `count`-fold replication of the type signature of `datatype` on any process must be equal to that at `root`

Collective Communication (ctd)

- ```
  int MPI_Gather(void *sendbuf, int sendcount,
                 MPI_Datatype sendtype,
                  void *recvbuf, int recvcount,
                  MPI_Datatype recvtype,
                  int root, MPI_Comm comm)
  ```
  - Each process (`root` included) sends the contents of its send buffer to `root`
  - `root` receives the messages and stores them in rank order
  - The specification of counts and types should not cause any location on `root` to be written more than once
  - `recvcount` is the number of items `root` receives from each process, not the total number of items it receives
  ```

Collective Communication (ctd)

- The function

```
int MPI_Scatter(void *sendbuf, int sendcount,
                MPI_Datatype sendtype,
                void *recvbuf, int recvcount,
                MPI_Datatype recvtype,
                int root, MPI_Comm comm)
```

  – Is the inverse operation to `MPI_Gather`

Collective Communication (ctd)

- The function

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

  - Combines the elements in the input buffer (`inbuf`, `count`, `datatype`) using the operation `op`
  - Returns the combined value in the output buffer (`outbuf`, `count`, `datatype`) of `root`
  - Each process can provide a sequence of elements
    - the combine operation is executed element-wise on each entry of the sequence

Collective Communication (ctd)

- Global reduction operations can combine
  - Pre-defined operations
    - Specified by pre-defined constants of type `MPI_Op`
  - A user-defined operation
- Pre-defined constants of type `MPI_Op`
  - `MPI_MAX` (maximum), `MPI_MIN` (minimum), `MPI_SUM` (sum), `MPI_PROD` (product)
  - `MPI_LAND` (logical and), `MPI_LOR` (logical or), `MPI_LXOR` (logical exclusive or)
  - `MPI_BAND` (bit-wise and), `MPI_BOR` (bit-wise or), `MPI_BXOR` (bit-wise exclusive or)
  - `MPI_MAXLOC` (maximum value and its location), `MPI_MINLOC` (minimum value and its location)

Collective Communication (ctd)

- A user-defined operation
  - Associative
  - Bound to an `op` handle with the function

```
int MPI_Op_create(MPI_User_function *fun,
                  int commute, MPI_Op *op)
```

  - Its type can be specified as follows:

```
typedef void MPI_User_function(
            void *invec, void *inoutvec,
            int len, MPI_Datatype datatype)
```

- Examples. See handouts.

Environment Management

- A few functions for various parameters of the MPI implementation and the execution environment
  - Function

```
int MPI_Get_processor_name(char *name, int *resultlen)
```

    returns the name of the processor, on which it was called
  - Function `double MPI_Wtime(void)` returns a floating-point number of seconds, representing elapsed wall-clock time since some time in the past
  - Function `double MPI_Wtick(void)` returns the resolution of `MPI_Wtime` in seconds
    - the number of second between sucessive clock ticks

Example of MPI Application

- Example. The application implements the simplest parallel algorithm of matrix-matrix multiplication.
  - See handouts for the source code and comments

Parallel Languages

Parallel Languages

- Many scientific programmers find the explicit message passing tedious and error-prone
  - Used to write their applications in Fortran
  - Consider MPI's parallel primitives too low level
    - unnecessary detailed description of parallel algorithms
  - Their algorithms are often straightforward and based on the *data parallel* paradigm

Parallel Languages (ctd)

- *Data parallel* programming model
  - Processors perform the same work on different parts of data
  - The distribution of the data across the processors determines distribution of work and interprocessor communication
- Main features of the data parallel programming style
  - Single threaded control
  - Global name space
  - Loosely synchronous processes
  - Parallelism implied by operations on data

Parallel Languages (ctd)

- Data parallel programming
  - Mainly supported by high-level parallel languages
  - A compiler generates the explicit message passing code, which will be executed in parallel by all participating processors (the SPMD model)
- Main advantage
  - Easy to use
    - data parallel applications are simple to write
    - easy to debug (due to the single thread of control)
    - easy to port legacy serial code to MPPs

Parallel Languages (ctd)

- HPF (High Performance Fortran)
  - The most popular data parallel language
  - A set of extensions to Fortran
    - aimed at writing data parallel programs for MPPs
  - Defined by the High Performance Fortran Forum (HPFF)
    - over 40 organizations
  - Two main versions of HPF
    - HPF 1.1 (November 10, 1994)
    - HPF 2.0 (January 31, 1997)

High Performance Fortran

- HPF 1.1
  - Based on Fortran 90
  - Specifies language constructs already included into Fortran 95
    - the FORALL construct and statement, PURE procedures
- HPF 2.0
  - An extension of Fortran 95
    - hence, simply inherits all these data parallel constructs

High Performance Fortran (ctd)

- Data parallel features provided by HPF
  - Inherited from Fortran 95
    - Whole-array operations, assignments, and functions
    - The `FORALL` construct and statement
    - `PURE` and `ELEMENTAL` procedures
  - HPF specific
    - The `INDEPENDENT` directive
      - can precede an indexed `DO` loop or `FORALL` statement
      - asserts to the compiler that the iterations in the following statement may be executed independently
      - the compiler relies on the assertion in its translation process

High Performance Fortran (ctd)

- HPF introduces a number of directives
  - To suggest the distribution of data among available processors to the compiler
  - The *data distribution directives*
    - structured comments of the form
      - `!HPF$ directive-body`
    - do not change the value computed by the program
      - an HPF program may be compiled by Fortran compilers and executed serially

High Performance Fortran (ctd)

- Two basic data distribution directives are
  - The `PROCESSORS` directive
  - The `DISTRIBUTE` directive
- HPF's view of the parallel machine
  - A rectilinear arrangement of abstract processors in one or more dimensions
  - Declared with the `PROCESSORS` directive specifying
    - its name
    - its rank (number of dimensions), and
    - the extent in each dimension
  - Example. `!HPF$ PROCESSORS p(4,8)`

High Performance Fortran (ctd)

- Two important intrinsic functions
  - `NUMBER_OF_PROCESSORS`, `PROCESSORS_SHAPE`
  - Example.
    - `!HPF$ PROCESSORS q(4, NUMBER_OF_PROCESSORS()/4)`
- Several processor arrangements may be declared in the same program
  - If they are of the same shape, then corresponding elements of the arrangements refer the same abstract processor
  - Example. If function `NUMBER_OF_PROCESSORS` returns `32`, then `p(2,3)` and `q(2,3)` refer the same processor

High Performance Fortran (ctd)

- The `DISTRIBUTE` directive
  - Specifies a mapping of data objects (mainly, arrays) to abstract processors in a processor arrangement
- Two basic types of distribution
  - Block
  - Cyclic
- Example 1. Block distribution

```
                REAL A(10000)
!HPF$           DISTRIBUTE A(BLOCK)
```

  - Array A should be distributed across some set of abstract processors by partitioning it uniformly into blocks of contiguous elements

High Performance Fortran (ctd)

- Example 2. The block size may be specified explicitly

```
      REAL A(10000)
!HPF$ DISTRIBUTE A(BLOCK(256))
```

  - Groups of exactly 256 elements should be mapped to successive abstract processors
  - There must be at least 40 abstract processors if the directive is to be satisfied
  - The 40th processor will contain a partial block of only 16 elements

High Performance Fortran (ctd)

- Example 3. Cyclic distribution

```
      INTEGER D(52)
!HPF$ DISTRIBUTE D(CYCLIC(2))
```

  – Successive 2-element blocks of D are mapped to successive abstract processors in a round-robin fashion

- Example 4. CYCLIC ⇔ CYCLIC(1)

```
      INTEGER DECK_OF_CARD(52)
!HPF$ PROCESSORS PLAYERS(4)
!HPF$ DISTRIBUTE DECK_OF_CARDS(CYCLIC) ONTO PLAYERS
```

High Performance Fortran (ctd)

- Example 5. Distributions are specified independently for each dimension of a multidimensional array

```
      INTEGER CHESS_BOARD(8,8), GO_BOARD(19,19)
!HPF$ DISTRIBUTE CHESS_BOARD(BLOCK, BLOCK)
!HPF$ DISTRIBUTE GO_BOARD(CYCLIC,*)
```

  - CHESS_BOARD
    - Partitioned into contiguous rectangular patches
    - The patches will be distributed onto a 2D processors arrangement
  - GO_BOARD
    - Rows distributed cyclically over a 1D arrangement of abstract processors
    - '*' specifies that it is not to be distributed along the second axis

High Performance Fortran (ctd)

- Example 6. The HPF program implementing matrix operation $C=AxB$ on a 16-processor MPP, where A, B are dense square 1000x1000 matrices.
  - See handouts for its source code
  - The PROCESSORS directive specifies a logical 4x4 grid of abstract processors, p

High Performance Fortran (ctd)

```
          PROGRAM SIMPLE
          REAL, DIMENSION(1000,1000):: A, B, C
!HPF$     PROCESSORS p(4,4)
!HPF$     DISTRIBUTE (BLOCK,BLOCK) ONTO p:: A, B, C
!HPF$     INDEPENDENT
          DO J=1,1000
!HPF$                INDEPENDENT
                     DO I=1,1000
                                A(I,J)=1.0
                                B(I,J)=2.0
                     END DO
          END DO
!HPF$     INDEPENDENT
          DO J=1,1000
!HPF$                INDEPENDENT
                     DO I=1,1000
                                C(I,J)=0.0
                                DO K=1,1000
                                           C(I,J)=C(I,J)+A(I,K)*B(K,J)
                                END DO
                     END DO
          END DO
          END
```

High Performance Fortran (ctd)

- Example 6 (ctd).
  - The `DISTRIBUTE` directive recommends the compiler to partition each of the arrays A, B, and C into equal-sized blocks along each of its dimension
    - A 4x4 configuration of blocks each containing 250x250 elements, one block per processor
    - The corresponding blocks of arrays A, B, and C will be mapped to the same abstract and hence physical processor
  - Each `INDEPENDENT` directive is applied to a `DO` loop
    - Advises the compiler that the loop does not carry any dependences and therefore its different iterations may be executed in parallel

High Performance Fortran (ctd)

- Example 6 (ctd).
  - Altogether the directives give enough information to generate a target message-passing program
  - Additional information is given by the general HPF rule
    - Evaluation of an expression should be performed on the processor, in the memory of which its result will be stored

High Performance Fortran (ctd)

- A clever HPF compiler will be able to generate for the program in Example 6 the following SPMD message-passing code
  - See handouts
- The minimization of inter-processor communication
  - The main optimisation performed by an HPF compiler
  - Not a trivial problem
  - No HPF constructs/directives helping the compiler to solve the problem
    - Therefore, HPF is considered a difficult language to compile

High Performance Fortran (ctd)

- Many real HPF compilers
  - Will generate a message-passing program, where each process sends its blocks of *A* and *B* to *all* other processes
    - This guarantees that each process receives all the elements of A and B, it needs to compute its elements of C
    - This universal scheme involves a good deal of redundant communications
      - sending and receiving data that are never used in computation

High Performance Fortran (ctd)

- Two more HPF directives
  - The `TEMPLATE` and `ALIGN` directives
    - Facilitate coordinated distribution of a group of interrelated arrays and other data objects
    - Provide a 2-level mapping of data objects to abstract processors
      - Data objects are first *aligned* relative to some template
      - The template is then distributed with `DISTRIBUTE`
      - Template is an array of nothings

High Performance Fortran (ctd)

- Example.

```
          REAL, DIMENSION(10000,10000) :: NW,NE,SW,SE
!HPF$     TEMPLATE EARTH(10001,10001)
!HPF$     ALIGN NW(I,J) WITH EARTH(I,J)
!HPF$     ALIGN NE(I,J) WITH EARTH(I,J+1)
!HPF$     ALIGN SW(I,J) WITH EARTH(I+1,J)
!HPF$     ALIGN SE(I,J) WITH EARTH(I+1,J+1)
!HPF$     DISTRIBUTE EARTH(BLOCK, BLOCK)
```

High Performance Fortran (ctd)

- HPF 2.0 extends the presented HPF 1.1 model in 3 directions
  - Greater control over the mapping of the data
    - `DYNAMIC`, `REDISTRIBUTE`, and `REALIGN`
  - More information for generating efficient code
    - `RANGE`, `SHADOW`
  - Basic support for *task parallelism*
    - The `ON` directive, the `RESIDENT` directive, and the `TASK_REGION` construct

MPP Architecture: Summary

- MPPs provide much more parallelism than SMPs
  - The MPP architecture is *scalable*
    - No bottlenecks to limit the number of efficiently interacting processors
- Message passing is the dominant programming model
- No industrial optimising C or Fortran 77 compiler for the MPP architecture
- Basic programming tools for MPPs
  - Message-passing libraries
  - High-level parallel languages

MPP Architecture: Summary (ctd)

- Message-passing libraries directly implement the message passing paradigm
  - Explicit message-passing programming
  - MPI is a standard message-passing interface
    - Supports efficiently portable parallel programming MPPs
    - Unlike PVM, MPI supports modular parallel programming
      - can be used for development of parallel libraries
- Scientific programmers find the explicit message passing provided by MPI tedious and error-prone
  - They use data parallel languages, mainly, HPF

MPP Architecture: Summary (ctd)

- When programming in HPF
  - The programmer specifies the strategy for parallelization and data partitioning at a higher level of abstraction
  - The tedious low-level details are left to the compiler
- HPF programs
  - Easy to write and debug
    - HPF 2.0 is more complicated and not so easy
  - Can express only a quite limited class of parallel algorithms
  - Difficult to compile