Fundamentals-2 (Fundamentals-2.pdf)

Architecture Types

- **Shared memory** system
- **Distributed memory** system

Parallel computing vs. distributed computing

OpenMP (programmingmodel-1.pdf)

Parallel programming approach

- Write **serial** program and use **compiler** to **automatically** parallelize it
- **OpenMP** is used to **assist compilers** to understand the serial program

OpenMP Programming model

- An implementation of thread model (**Multiple threads**)
- Used for **shared memory** architecture
- **Fork-join** model

Process and Thread

How many threads are generated

Partition and allocation of loop iterations

schedule(static,4), schedule(dynamic [,chunk]), schedule(runtime)

Synchronisation in OpenMP

Critical, Barrier

Data Scope Clauses in OpenMP

Shared, Private, Reduction

Models of Parallel Programming (programmingmodel-2.pdf)

Different approaches for programming on a HPC system include:

- **Smart compilers**, which automatically ***parallelise sequential codes***
- **Data parallelism**: multiple processors run ***the same operation on different elements of a data structure***
- **Task parallelism**: multiple processors run ***different operations***
- **Dedicated languages** designed specifically for parallel computers

Compiler Approach

- Automatically **parallelize** the sequential codes
- Very **conservative**
- **Re-implementing** the code in a more efficient way
- Can **remove the dependency** if possible

Dependency and Parallelism

Two types of Dependency:

- **Control dependency**: waiting for the instruction which controls the execution flow to be completed

        - IF (X!=0) Then Y=1.0/X: Y=1.0/X has the control dependency on X!=0
    - **Data dependency**: dependency because of calculations or memory acces
        - **Flow dependency**: A=X+Y; B=A+C;
        - **Anti-dependency**: B=A+C; A=X+Y;
        - **Output dependency**: A=2; X=A+1; A=5;

Removing the dependency

Anti-dependency and output-dependency can be removed by **renaming** the variables(by creating another place)

Automatic parallelization

For a for loop, check whether there is **dependency** between different iterations.

Features of the Compiler approach

- Can work fairly well for some regular problem
- Fully automatic (efficient) parallelisation is difficult, and unlikely to be efficient in general

Assisting Compiler

Programmers can assist the compiler by **writing the code in a way that explicitly expresses** the parallelism in the program.

-- Using directives， OpenMP

Data Parallelism

Task parallelism vs. Data parallelism

Data Parallelism in OpenMP

- #pragma omp **parallel for**
- #pragma omp **parallel** { #pragma omp **for** ...}

Task Parallelism in OpenMP

- #pragma omp **parallel** { #pragma omp **sections** { #pragma omp **section** ...}}

Languages that supports Data parallelism

F90 and HPF allow scalar operations to be applied to arrays to support the data parallelism.

Data parallelism

- A data parallel program is **a sequence of explicitly and/or implicitly parallel statements.**
- The compiler can **partition** the data into disjoint sub- domains, one per thread (CPU core).
- **Data placement** is an essential part of data-parallelism (for NUMA machine), and if you get the data locality wrong, you will take a performance hit.
- Fortunately, compilers can achieve **data partition, data allocation/placement, thread communications.**

Something about Data parallelism

A data-parallel programming is higher-level than the message passing-type approach

- programmer does **not need** to specify **data partition** and **thread communication** structures
- this is done by the compiler (inferred from the program and underlying computer architecture)

However

- **not all algorithms** can be specified in data parallel terms
- if the program has **irregular communication patterns** then this will be compiled **less efficiently**

Specially Designed Language

Occam

Shared Memory Parallel Programming (programmingmodel-3.pdf)

Multiprocessing

Scheduling Processes

When: -- time slice runs out -- System call -- trap

Overhead of switching processes: is relatively **high**, have to save and load the following information

User Space Thread

Scheduling User Space Threads

When: -- no time slice for each thread, -- call the thread switching explicitly

Overhead: User space threads usually switch **fast**

Kernel Space(OS-managed) Thread

Scheduling Kernel space Threads

When: -- Time slicing -- System calls -- Traps

Overhead: The switching overhead stands **between** processes and user space threads

Dealing with Multithreading

Problems with concurrency

- Race conditions

- Deadlock
- Starvation

Synchronization

Two types: **mutual exclusion** and **cooperation**.

Techniques: 1. **Mutex** (**semaphore**, **lock**) to address mutual exclusion. 2. **wait** and **notify** to address cooperation

Synchronization of Java Threads

Message Passing Programming I (mpi-1.pdf)

Message Passing Programming

Message Passing Interface (MPI)

OpenMP vs MPI

- MPI ==> **distributed-memory** systems
- OpenMP ==> **shared-memory** systems
- Both are **explicit** parallelism
- OpenMP is **higher-level** control (Compiler can automatically parallelise the codes when instructed)
- MPI is **lower-level** control (Data partition, allocation and process communication are conducted by programmers)

MPI functions

Intuitive Interfaces for sending and receiving messages

Send(data, destination), Receive(data_location, source)

>>>NOT ENOUGH BECAUSE<<<

Express the data in the interface

Early stages:

(address, length) for the send interface, (address, max_length) for the receive interface

shortage:

- The data may **not** occupy **contiguous** memory locations
- Storing **format** for data may **not** be the **same** in a heterogeneous platform

Triple:

(address, count, datatype) for send, (address, max_count, datatype) for receive.

And now the interfaces are:

- send(address, count, datatype, destination, msg_tag)
- receive(address, max_count, datatype, source, msg_tag)

advantages:

- Such a format reflects the fact that a message contains the **structures**, not just a string of bits
- Programmers can construct their **own datatype**. Handle **non-contiguous** data

Communicator (distinguish messages)

A communicator consists of **a group of processes** and **a communication context**

Initial communicator: MPI_COMM_WORLD

Interface:

send(address, count, datatype, destination, tag, **comm**)

receive(address, maxcount, datatype, source, tag, **comm**, **status**), **Status** holds the information about source, tag and actual size of the received message

Express source and destination

They are processes in communicator, and identified by **ranks**

Some other issues

In receiver, **tag and source** can be a **wildcard**.
ie. MPI_ANY_TAG, MPI_ANY_SOURCE

First six functions

- MPI_Send (buf, count, datatype, dest, tag, comm)
- MPI_Recv (buf, count, datatype, source, tag, comm, status)
- MPI_Init (int *argc, char**argv)
- MPI_Comm_size (MPI_Comm comm, int* size)
- MPI_Comm_rank (MPI_Comm comm, int* pid)
- MPI_Finalize()

Other functions added in MPI

- Convenience (collective operations, topology)
- Modularity (communicators, groups)
- Efficiency (non-blocking send/receive)
- Robustness (ready-mode communication)
- Flexibility (datatype)

Convenience (collective operations,topology)

- **MPI_Reduce(sendbuf, recvbuf, count, type, op, root, comm)**, MPI performs op

over the data in the sendbuf and put the result in the recvbuf in root.

- **MPI_Allreduce(sendbuf, recvbuf, count, type, op, comm)**, MPI performs op over the data in sendbuf and distributes the result back to recvbuf in all processes.
- **MPI_Barrier(comm)**, Global synchronization. No processes return from function **until** all processes have called it.
- **MPI_Bcast (buf, count, type, root, comm)**, Broadcast data from root to all processes.
- **MPI_Gather (sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, root, comm)**, Gather all the data from other processes to root.
- **MPI_Scatter(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, root, comm)**, Scatter data from one process to all processes.

Modularity (communicators, groups)

MPI supports modular programming via **communicators**. Provides **local namespaces** for processes and message tag. All MPI communication operations specify a communicator (process group that is engaged in the communication)

Creating new communicators - Approach 1

- int MPI_Comm_group(MPI_Comm comm, MPI_Group *group)
- int MPI_Group_excl(MPI_Group group, int n, int *ranks, MPI_Group *newgroup)
- int MPI_Group_incl(MPI_Group group, int n, int *ranks, MPI_Group *newgroup)
- int MPI_Comm_create(MPI_Comm comm, MPI_Group group, MPI_Comm *newcomm)
- int MPI_Group_free(MPI_Group *group) int MPI_Comm_free(MPI_Comm *comm)

Creating new communicators - Approach 2

- MPI_Comm_split(comm, colour, myid, *newcomm);

Robustness (non-blocking send/receive)

Blocking send

The sender doesn't return until the application buffer can be re-used (which often means that the data have been copied from application buffer to system buffer) //note: it doesn't mean that the data will be received

Blocking receive

The receiver doesn't return until the data have been ready to use by the receiver (which often means that the data have been copied from system buffer to application buffer)

Non-blocking send/receive

- The calling process **returns immediately**
- Just request the MPI library to perform the communication: **no guarantee** when this will happen
- **Unsafe** to modify the application buffer until you can make sure the requested operation has been performed (MPI provides routines to test this)
- Can be used to **overlap** computation with communication and have possible **performance gains**

Testing non-blocking communications

Efficiency (communication mode)

The communication modes refers to the send routines:

- **Standard** send: MPI_Send (blocking), MPI_Isend (nonblocking)
- **Synchronous** send: MPI_Ssend (blocking), MPI_Issend (non- blocking)
- **Buffered** send: MPI_Bsend (blocking), MPI_Ibsend (nonblocking)
- **Ready** send: MPI_Rsend (blocking), MPI_Irsend (nonblocking)

Two Receive routines:

- Blocking receive routine: MPI_Recv()
- Non-blocking receive routine: MPI_Irecv()

Virtual topology --Cartesian Topology

MPI_Cart_create(MPI_Comm comm_old, int ndims, int *dims, int *periods, int reorder, MPI_Comm *comm_cart)

MPI_Cart_rank(comm, coords, rank)

MPI_Cart_coords(comm, rank, ndims, coords)

Flexibility (datatype)

Derived datatype

Derived datatype allows the users to construct (derive) their own data types.

Derived datatypes can be used in:

- Grouping **non-contiguous** data.
- Grouping data of **different datatypes**.

Provides the opportunity to send this kind of data conveniently and potentially improving performance

Memory Layout of a Datatype

Format: {(type_0, offset_0), (type_1, offset_1), ..., (type_n, offset_n)}

Examples for primitive datatypes: **MPI_Char: {(MPI_Char, 0)}, MPI_Int: {(MPI_Int, 0)}**

Three Attributes to characterize non-contiguous data

	Same type in each block	Same number of elements in each block	Same distance between consecutive block
MPI_Type_vector	Yes	Yes	Yes
MPI_Type_create_indexed_block	Yes	Yes	No
MPI_Type_indexed	Yes	No	No
MPI_Type_struct	No	No	No

MPI_Type_Vector(count, blocklen, stride, oldtype, newtype)

MPI_Type_create_indexed_block(int count, int blocklength, int array_of_offsets[], MPI_Datatype oldtype, MPI_Datatype *newtype)

MPI_Type_Indexed(count, lengths[], offsets[], oldtype, newtype)

MPI_Type_struct(int count, int *array_of_blocklengths, MPI_int *array_of_displacements, MPI_Datatype *array_of_types, MPI_Datatype *newtype)

MPI_Type_contiguous(int count, MPI_Datatype old_type, MPI_Datatype *new_type_p)

How are the data with derived data type are sent?

- Pack the data elements specified by the memory layout into the contiguous space in system buffer
- MPI library sends the data in system buffer

Why using derived data type can improve performance? --Two possible solutions to non-contiguous data

- Call a MPI_Send for each data block that occupies contiguous space. Calling MPI_Send multiple times, which occurs higher overhead (e.g., handshaking between sender and receiver, metadata enclosed for each message).
- Copy the non-contiguous data to a contiguous buffer space and then call a MPI_Send to send the data in the buffer. Needs to copy the data twice: 1) copy the non-contiguous data to a contiguous buffer space; 2) copy the data in the application buffer to system buffer.

Performance (performance-1.pdf)

Metrics to measure the parallelization quality of parallel programs

- Degree of Parallelism, average parallelism
- Effective work

- Speedup
- Parallel efficiency

Degree of Parallelism (DOP)

The **number** of processors engaged in execution **at the same time.**

Two forms of functions

continuous form and discrete form

Factors that affect the DOP

- Application properties: data dependency, communication overhead
- Resource limitations: number of processors, memory, I/O
- Parallelization strategy: divide up work as evenly as possible

Effective Work

This is the **total amount of computation** executed **within a given time interval**. It relates to DOP

Discrete form: $W = \Delta \sum_{i=1}^{m} i.t_i$ where t_i is the total time that $\text{DOP} = i$ and $\sum_{i=1}^{m} t_i = t_2 - t_1$

Continuous form: $W = \Delta \int_{t_1}^{t_2} \text{DOP(t)} dt$

Average Parallelism

Continuous form: $A = \frac{1}{t_2 - t_1} \int_{t_1}^{t_2} \text{DOP(t)} dt$

Discrete form: $A = \frac{\sum_{t=1}^{m} i.t_i}{\sum_{t=1}^{m} t_i}$

Speedup

The **improvement** can be measured by **speedup**

$$S(n) = t_1/t_n$$

- t_1 is the worst case execution time of the optimal serial implementation.
- t_n is the worst case execution time of the parallel algorithm using n processors.

What is “good” speedup

How to obtain good S(n) for large n

Reducing the Impact of Communication

Parallel efficiency

E(n) = S(n) / n

Main issues that affect parallel efficiency:

- Same as the factors for affecting speedup
- The impact of n; typically, greater n, lower efficiency

Iso-efficiency

Constant Efficiency:

- **How the amount of computation performed (N) must scale with processor number P to keep parallel efficiency E constant !!!!!!!**
- The function of **N over P** is called an **algorithm’s iso-efficiency function**
- An algorithm with an iso-efficiency function of N=O(P) (**linear function**) is **highly** scalable
- An algorithm with a **quadratic or exponential** iso-efficiency function is **less** scalable

Four approaches to modelling the performance of a parallel application

- Speedup

- Amdahl's law
- Asymptotic analysis
- Modelling execution time

Speedup approach

Amdahl's Law

Amdahl's law shows us the limitation of parallelising codes.

Disadvantages:

- Can only tell the upper bound of the speedup for a particular algorithm
- Cannot tell if other algorithms with greater parallelism exist for the problem.

Asymptotic analysis

In this modelling approach, we can say something like "the algorithm takes the time of O(nlog2n) on n processors"

Disadvantages:

- ignore the lower-order term
- Only tell the order of the execution time of a program, not its actual execution time

Modelling execution time

Example (1-D):

Modelling computation time:

If we assume a grid of size N*N*Z points, and using 1-D decomposition (along y-axis) to partition the grid among P processors, then

- each task is responsible for a subgrid of size N*(N/P)*Z
- then, T_{comp} for each subgrid can be calculated as follows, where t_c is the average time of calculating a single grid point

$$T_{comp}=t_c*N*(N/P)*z$$

Modelling the time of sending one message:

- T_{msg} (the time spent in sending one message) can be calculated as follows,

$$T_{msg}=t_s+t_wL$$

where t_s is the message startup time, t_w is the transfer time per byte, L is the size of the message

T_{msg} is a function of L; t_s and t_w are constants given a computing platform

Modelling communication time:

- Communication time for calculating a subgrid can be computed as

$$T_{comm}=2(t_s+t_w2NZ)$$

- Hence, the performance model for the execution time of calculating the velocity of the grid of points is

$$T_p=T_{comp}+T_{comm}=t_c*N*(N/p)*Z+2(t_s+t_w2NZ)$$

Speedup and parallel efficiency:

- The execution time on one processor is

$$T_1 = t_c N^2 Z$$

- Speedup is

$$S(P) = \frac{t_c N^2 ZP}{t_c N^2 Z + 2t_s P + t_w 4NZP}$$

- Parallel efficiency can be calculated as

$$E = \frac{t_c N^2 Z}{t_c N^2 Z + 2t_s P + t_w 4NZP}$$

Iso-efficiency

Question: What is the iso-efficiency function?

E can be reduced to

$$E = \frac{t_c}{t_c + \frac{t_s 2P}{ZN^2} + \frac{t_w 4P}{N}}$$

When N=P, E remains approximately constant as P changes(except when P is small)

Example (2-D):

- If applying 2-D decomposition, then
- Execution time can be modelled as

$$T_{comp} = t_c N^2 Z/P$$

$$T_{comm} = 4(t_s + t_w 2\frac{N}{\sqrt{P}}Z)$$

$$T_P = T_{comp} + T_{comm} = \frac{t_c N^2 Z + t_s 4P + t_w 8NZ\sqrt{P}}{P}$$

1D decomposition

$$T_{comp} = t_c * N * (N/P) * z$$

$$T_{comm} = 2(t_s + t_w 2NZ)$$

Iso-efficiency

Parallel efficiency can be modelled as

$$\frac{t_c N^2 Z}{t_c N^2 Z + t_s 4P + t_w 8NZ\sqrt{P}}$$

- When N=$\sqrt{P}$, E remains constant as P increases
- Iso-efficiency in 1D is N=P
- Therefore, for this particular communication pattern, applying 2D decomposition will achieve better scalability than 1D decomposition

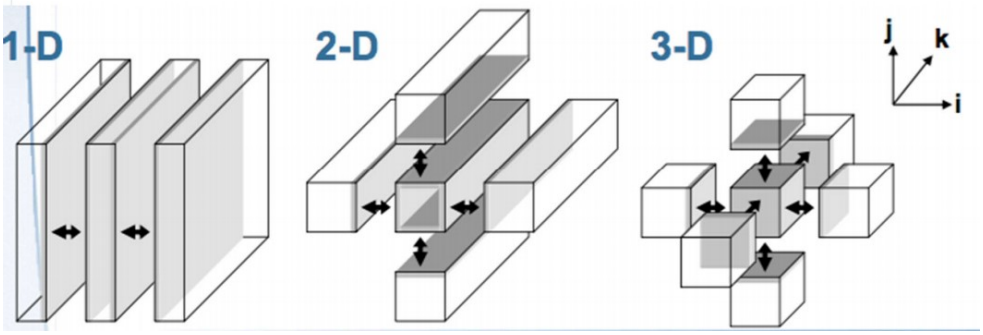
！！！注意 2D中如果除以 P的时候需要除的是 **根号P**，所以Tcomp虽然和1D一样，但是含义不一样！！！

Decomposition analysis

The lower surface-to-volume ratio, the better:

- Surface = communication
- Volume = computation
- means lower proportion of communication time in the whole execution time

		Sub-grid Length				
	Sub-grid Volume	I	J	K	Sub-grid Surfaces	Surface to Volume
1-D	c	$V^{1/3} / n$	$V^{1/3}$	$V^{1/3}$	$2V^{2/3}$	$2n / V^{1/3}$
2-D	c	$V^{1/3} / n^{1/2}$	$V^{1/3} / n^{1/2}$	$V^{1/3}$	$4V^{2/3} .n^{1/2}$	$4n^{1/2} / V^{1/3}$
3-D	c	$V^{1/3} / n^{1/3}$	$V^{1/3} / n^{1/3}$	$V^{1/3} / n^{1/3}$	$6c^{2/3}$	$6n^{1/3} / V^{1/3}$



Time for sending a message

GPU and CUDA - I (gpu-1.pdf)

GPU vs CPU

CUDA

CUDA keywords and kernel

Keywords:

- __global__: a kernel function to be executed in **GPU**
- __host__ : run on **host**, (default if do not have any CUDA keywords)

Outline of a vecAdd() function for GPU

a. **Part 1: Allocate device memory** for A, B, and C and **Copy** A and B from host memory to device memory

b. **Part 2: Launch Kernel code** to perform the actual operation on GPU

c. **Part 3: Copy** the result C from device memory to host memory; **Free** device vectors

Memory Management in GPU

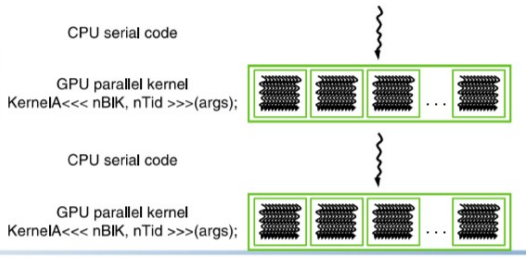
- Allocate the device memory in GPU: **cudaMalloc(void ** devPtr, size_t size)**
  - devPtr: a pointer to the address of the allocated memory
  - size: Size of allocated memory
- Memory data transfer: **cudaMemcpy(dst, src, count, kind)**
  - destination location of the data to be copied
  - source location of the data
  - size of the data
  - The types of memory copying: host to host, host to device, device to device, device to host

Various related issues in Part 2

1. Execution model of the kernel function
2. Thread structure
3. Execution configuration
4. Workload distribution

Execution model of the kernel function

- The execution starts with host (CPU) execution
- When a kernel function is called, it is executed by a large number of threads on the GPU
  - All the threads are collectively called a *grid*
- When all threads of a kernel complete their execution, the corresponding grid terminates
- The execution continues on the host until another kernel is called



Thread Structure for Running a Kernel

- When a host code launches a kernel, CUDA generates a grid of thread blocks
- Each block contains the same number of threads (up to 1024)
- Each thread runs the same kernel function

Thread Organization

- gridDim(x, y, z): the dimensions of the grid
- blockDim(x, y, z): the dimensions of the block
- blockIdx(x, y, z): - the coordinate (ID) of the block in the grid, - it can be accessed by the calling thread to obtain which block it is in
- threadIdx(x, y, z): - the local coordinate (ID) of a thread in a block, - It can be accessed by the calling thread to obtain its local position in the block

Execution configuration of kernel launch

Execution configuration is specified when invoking a kernel function, the format is :**kernelFunction** <<<**grid_dimension**, **block_dimension**>>>(...);

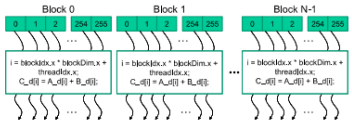
Example: How many threads?

dim3 a(3, 2, 4); dim3 b(128, 1, 1); vecAdd <<>> (…); Answer: 3*2*4*128

Workload distribution

- Use different threads process different parts of data in the kernel
- Need to match different threads to different parts of the data
  - **All threads in a grid execute the same kernel function**
  - **The threads use their coordinates (i.e., blockidx and threadidx) to**
    - **distinguish themselves from each other**
    - **identify the appropriate part of the data to process**

```
__global__
void vecAddKernel(float* A, float* B, float* C, int n)
{
     int i = threadIdx.x + blockDim.x * blockIdx.x;
    if(i<n) C[i] = A[i] + B[i];
}
```



Parallel I/O -I (io-1.pdf)

Version 1.0

version 2.0

version 3.0

What is Parallel I/O

Multiple processes of a parallel program **accessing** (reading or writing) different parts of a common file **at the same time**

I/O optimization

Data Sieving reading

Data sieving is used to combine lots of small accesses into a single larger one. (**Reducing the number of I/O operations**)

Data Sieving Writes

Using data sieving for writes is more complicated. (**-read the entire region first -Then make the changes -Then write the block back**)

Requires locking in the file system (Can result in **false sharing**)

Collective I/O

Collective I/O functions must be called by all processes participating in I/O at the same time.

Allows I/O layers to know more as a whole about the data to be accessed.

Access pattern 1

int MPI_File_seek(MPI_File mpi_fh, MPI_Offset offset, int whence);

int MPI_File_read(MPI_File mpi_fh, void *buf, int count, MPI_Datatype datatype, MPI_Status *status);

- Individual file pointers per process per file handle
- Each process sets the file pointer with some suitable offset
- The data is then read into the local array
- This is **not a collective operation**

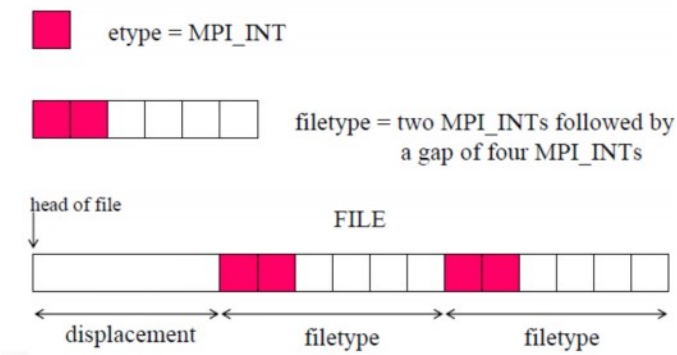
Access pattern 2

int MPI_File_read_all(MPI_File mpi_fh, void *buf, int count, MPI_Datatype datatype, MPI_Status *status);

- read_all is a **collective** version of the read operation
- This is **blocking read**
- Each process accesses the file **at the same time**
- This may be useful as independent I/O operations **do not convey** what other procs are doing at the same time

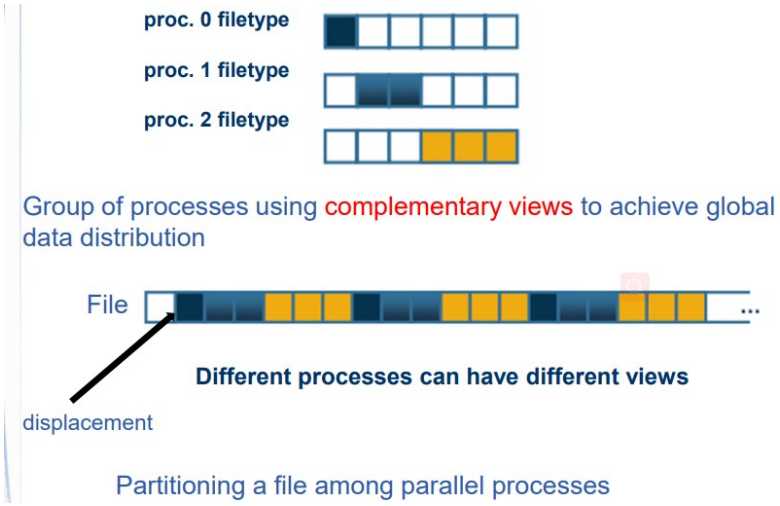
Access pattern 3

Definitions



- File view
  - View is the set of data visible to a process in a file, defined by displacement, etype and filetype
- Displacement
  - Position relative to the beginning of a file
  - Defines the location where a view begins
- etype (elementary datatype)
  - Can be a predefined or derived datatype
  - Unit of data access and positioning
  - Displacement is expressed as multiples of etypes
- Filetype
  - Defines a template/pattern in a file accessible by a process
- View is a repeated pattern defined by filetype (in units of etypes) beginning at the displacement

Complementary views of multiple processes



- Creates a datatype describing a subarray of a multi-dimentional array
- Commits the datatype
- Open the file as before
- Now changes the process's view of the data in the file using set_view
- set_view is **collective**
- Although the **reads** are still **independent**

Access pattern 4

- Creates and commits datatype as before
- Now changes the processe's view of the data in the file using set_view
- **set_view** is **collective**
- **Reads** are now **collective**

Pattern 4 offers (potentially) the best performance

Parallel I/O -II (io-2.pdf)

Data Access Routines

- According to positioning methods, there are
  - explicit offset
  - implicit file pointer

- According to synchronism, there are
  - Blocking
  - Non-blocking
- According to coordination, there are
  - collective
  - Non-collective

Positioning

Three types:

- **explicit offset**: The data access routines that accept explicit offsets contain **_AT in their name**
- **individual file pointer:** The names of the individual file pointer routines contain **no positional qualifier**
- **shared file pointer:** The data access routines that use shared file pointers contain **_SHARED**

Three positioning methods do not affect each other

Open a file collectively

Synchronism

Coordination

Interoperability

I/O Consistency

Reason: The routine is considered completed when the application buffer is copied to the system buff, but the system will only write to disks when necessary

- MPI can **automatically** guarantee the consistency **in some cases**

- When MPI does not automatically do so, the **user should take steps** to ensure consistency
    - **set atomicity to true**: the step of writing the data to system buffer and the step of transferring data to storage devices become an atomic step
    - **close the file and reopen it**: closing a file will force all previous writes to be transfered to storage devices
    - **Use MPI_File_sync**: cause all previous writes to be transfered to storage devices

Cluster Technologies -I (cluster-1.pdf)

Why Clusters?

- 1. Very high performance Microprocessors
- 2. High speed networking
- 3. Standard tools for parallel/ distributed programming

Cluster Components

1.Nodes, 2.OS, 3.High performance Networks, 4.Network Interfaces, 5.Communication Software, 6.Cluster Middleware, 7.Development Tools, 8.Applications

Vampir Visualizes Communication Pattern

Cluster Management Software

- Help the **allocation of resources to jobs**, **subject to jobs' resource requirements and policy restrictions** in cluster
- Three parties in a cluster: **Users, Administrators, Cluster management software**.
- Five activities are performed
    - **Queuing**
        - **Job submission** usually consists of two primary parts

        - Job description
        - Resource requirements
    - Once submitted, **the jobs are held in the queue** until i) the job is at the head of the queue and ii) the matching resources are available
- **Scheduling**
    - Determining at what time a job should be put into execution on which resources
    - There are a variety of metrics to measure scheduling performance
        - System-oriented metrics
        - user-oriented metrics
        - They can contradicts each other and balance needs to be made
- **Monitoring**
    - Providing information to administrators, users and the Cluster manager on the status of jobs and resources
    - The method of collecting the info may differ between different cluster manager, but the general purpose is the same
- **Resource management**
    - Handling the details (Starting the job execution on the resources Stopping a job Cleaning up the temporary files generated by the jobs after the jobs are completed or aborted Removing or adding resources)
    - For the batch system, the jobs are put into execution in such a way that the users don’t have to be present during execution
    - For interactive systems, the users have to be present to supply information during the execution.
- **Accounting**
    - Accounting for which users are using what resources for how long
    - Collecting resource usage data (e.g. job owner, resources requested by the job, resource consumption by the job)
    - Accounting data can be used for

Schedule Polices

The simplest policy:

- **First-Come First-Served (FCFS)**
- Jobs are run in the same order as they are submitted.
- Does not require prior knowledge about jobs (e.g. runtime).
- Problems: jobs can block other jobs from starting, despite there being no performance benefit to either job.

Backfilling

- The problem with FCFS is that **idle time (sum of unused processing intervals) can be significant**.
- One improvement is to “backfill”.
- Allows a job to start if it does not delay the execution of the first job in the queue.

Advantages:

- Utilisation is improved.

Disadvantages:

- **Information** about the job execution time is **required**.
- User estimation are usually **inaccurate**.
- It is a policy decision to decide what to do if a job **overruns**;
- The default decsion is to **terminate** the job if it overruns
- Otherwise some users may deliberately **underestimate** the job length to get an earlier job start time.

Schedule Polices

Reservation

- **Quality of service** (QoS) is an important scheduling metric.
- In addition to normal scheduling, reservation services can be used to plan resource allocation.
- Users are able to reserve the resources in the cluster that the users use at some point in the future.

Single System Image (SSI) in Cluster

Cluster Networking (cluster-2.pdf)

Cluster Networks

Communication performance

communication is the overhead in parallel computing.

An increase in compute power typically demands **proportional increases** in communication performance.

Ways to improve communication performance

- **Overlap** communication with computation
- **Combine** several small **messages** into a big message
- Have a **network** with better performance

The metrics to measure performance: bandwidth and latency

Communication time = **time spent at the sender** (latency at the sender) + **datasize/bandwidth** + **switching time** (latency at the switch) + **time spent at the receiver** (latency at the receiver)

Interconnection Topologies

Two general structures for standard LANs

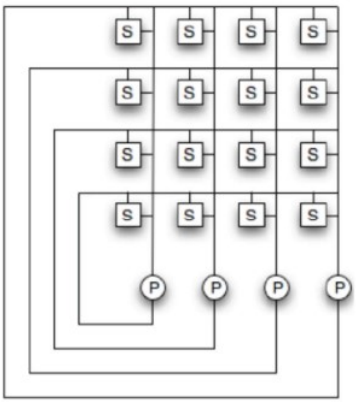
- Shared network(bus)
- Switched network

Metrics to evaluate network topology

- **Scalability** : the function of switch over nodes.
- **Degree**: number of links to / from a node
  - measure **aggregate bandwidth**
- **Diameter**: the shortest path between the furthest nodes.
  - measure latency
- **Bisection width**: **the minimum number of links** that must be cut in order to divide the topology into two independent networks of the same size (+/- one node).
  - measure of **bottleneck bandwidth** - if higher, the network will perform better under heavy load.

Crossbar switch

- **Low resource contention**
- **Switch scalability** is poor - **O(N^2)**
- Lots of **wiring**, cost is **high**



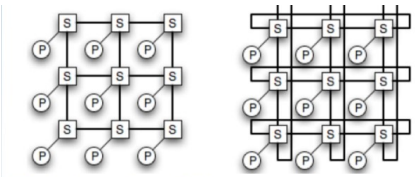
Linear Arrays and Rings

- We need networks with scaling better than O(N^2).
- In one dimension, we can construct linear arrays.

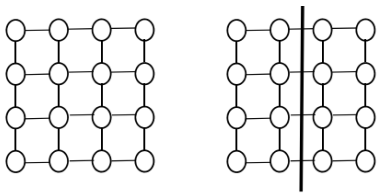
- Switch scalability: O(N)
- wrap around to make a ring or 1D torus.
- Diameter for linear array: O(N): latency is high.
- 2D/3D Cartesian applications will perform poorly with this network

☐Diameter for 1D array: O(N);
☐Bisection width for 1D linear array: O(1);
☐Diameter for 1D torus: O(N/2); Bisection width for 1D torus: O(2)

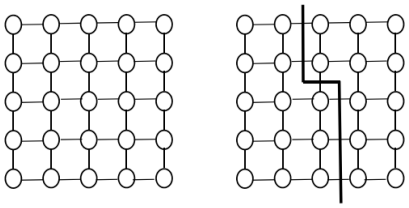
2D Mesh and Torus



☐Switch scaling: O(N)
☐Average degree: 4
☐Diameter for 2D mesh: $O(2n^{1/2})$
☐Bisection width for 2D mesh: $O(n^{1/2})$
☐Diameter for 2D torus: $O(n^{1/2})$
☐Bisection width for 2D torus: $O(2n^{1/2})$



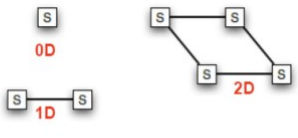
Bisection width for 2D mesh: $O(n^{1/2})$

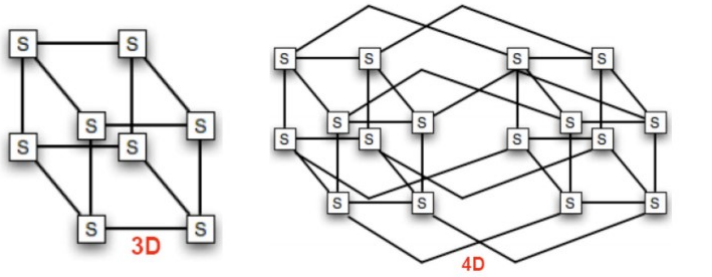


Bisection width for 2D mesh: $O(n^{1/2}+1)=O(n^{1/2})$

Hypercube

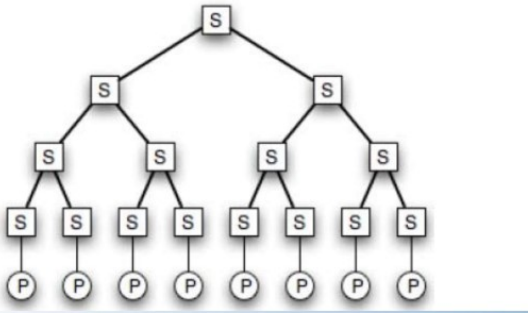
☐The methods to construct hypercube
☐K dimension, Switches N= 2^K.
☐Diameter: O(K).
☐Good bisectional width $O(2^{K-1})$ or O(N/2).





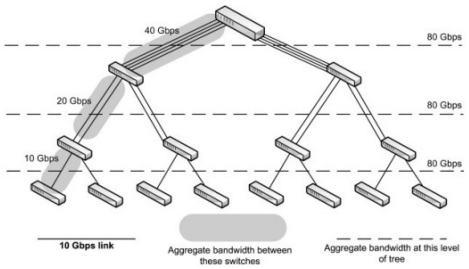
Binary Tree

- Scaling:
  - $n = 2^d$ processor nodes (where d = depth)
  - 2^{d+1}-1 switches
- Degree: 3
- Diameter: O(2d)
- Bisection width: O(1)



Fat tree

- Branches nearer the top of the tree are "fatter" than branches further down the the tree
- Add the links such that the number of down links of a switch is the same as the number of uplinks of the switch
- Similar in diameter to a binary tree.
- Bisection width is 2^{d-1}.



Summary of topologies

Topology	Degree	Diameter	Bisection
1D Array	2	N-1	1
1D Ring	2	N/2	2
2D Mesh	4	$2N^{1/2}$	$N^{1/2}$
2D Torus	4	$N^{1/2}$	$2N^{1/2}$
Hypercube	$n=\log_2(N)$	n	N/2

Switching Techniques

Store-and-forward

Worm hole routing (or cut-through switching)

Performance Model for Store-and-Forward

Performance model for Cutting Through

Cutting Through vs. Store-and-Forward

- **Store and Forward:**

  $T^s_{comm} = T_s + (T_w L + T_h) * D$

- **Cutting Through:**

  $T^c_{comm} = T_s + D * (T_h + T_w * f) + T_w * (L-f)$

Which is bigger?

$$T^s_{comm} > T^c_{comm} \quad \text{since } D*(L-f) > (L-f)$$

The underlying reason is because the rest of flits are sent in parallel through links under cut-through switching