Lecture Notes on Parallel Computation

Stefan Boeriu, Kai-Ping Wang and John C. Bruch Jr.
Office of Information Technology and
Department of Mechanical and Environmental Engineering
University of California
Santa Barbara, CA

1. Introduction

1.1 What is Parallel Computation?

Computations that use multi-processor computers and/or several independent computers interconnected in some way, working together on a common task.

- Examples: CRAY T3E, IBM-SP, SGI-3K, Cluster of Workstations.

1.2 Why use Parallel Computation?

- Computing power (speed, memory)
- Cost/Performance
- Scalability
- Tackle intractable problems

1.3 Performance limits of Parallel Programs

- Available Parallelism – Amdahl's Law
- Load Balance
  - some processors work while others wait
- Extra work
  - management of parallelism
  - redundant computation
- Communication

1.4 Top 500 Supercomputers – Worldwide

- Listing of the 500 most powerful computers in the World, available from www.top500.org.
- Rmax [Gflops/s for the largest problem] - from LINPACK MPP [Massively Parallel Processors]
- Updated twice a year.
- Top 13 presented in Table 1.4.

Table 1.4
TOP 10 - June 2003

Rank	Manufacturer	Computer	Rmax	Installation Site	Country	Year	# Proc
1	NEC	Earth-Simulator	**35860**	Earth Simulator Center Japan/2002	Japan	2002	5120
2	Hewlett-Packard	ASCI Q - AlphaServer SC ES45/1.25 GHz	**13880**	Los Alamos National Laboratory	USA	2002	8192
3	Linux Networx	MCR Linux Cluster Xeon 2.4 GHz - Quadrics	**7634**	Lawrence Livermore National Laboratory	USA	2002	2304
4	IBM	ASCI White, SP Power3 375 MHz	**7304**	Lawrence Livermore National Laboratory	USA	2000	8192
5	IBM	SP Power3 375 MHz 16 way	**7304**	NERSC/LBNL	USA	2002	6656
6	IBM	xSeries Cluster Xeon 2.4 GHz - Quadrics	**6586**	Lawrence Livermore National Laboratory	USA	2003	1920
7	Fujitsu	PRIMEPOWER HPC2500 (1.3 GHz)	**5406**	National Aerospace Laboratory of Japan	Japan	2002	2304
8	Hewlett-Packard	rx2600 Itanium2 1 GHz Cluster - Quadrics	**4881**	Pacific Northwest National Laboratory	USA	2003	1540
9	Hewlett-Packard	AlphaServer SC ES45/1 GHz	**4463**	Pittsburgh Supercomputing Center	USA	2001	3016
10	Hewlett-Packard	AlphaServer SC ES45/1 GHz	**3980**	Commissariat a l'Energie Atomique (CEA)	France	2001	2560

2. Parallel Systems

2.1 Memory Distribution

2.2.1 Distributed Memory

- Each processor in a parallel computer has its own memory (local memory); no other processor can access this memory.
- Data can only be shared by message passing
- Examples: Cray T3E, IBM SP2

2.2.2 Shared Memory

- Global memory which can be accessed by all processors of a parallel computer.
- Data in the global memory can be read/write by any of the processors.
- Examples: Sun HPC, Cray T90

2.1.3 Hybrid (SMP Cluster)

- A distributed memory parallel system but has a global memory address space management. Message passing and data sharing are taken care of by the system.
- Examples: SGI Power Challenge Array

2.1.4 Comparison

- ***Shared Memory***
    - Explicit global data structure
    - Decomposition of work is independent of data layout
    - Communication is implicit
    - Explicit synchronization
        - Need to avoid race condition and over writing
- ***Message Passing***
    - Implicit global data structure
    - Decomposition of data determines assignment of work
    - Communication is explicit
    - Synchronization is implicit

2.2. Instruction

Flynn's classification of computer architectures (1966):

2.2.1 MIMD (Multi-Instruction Multi-data)

- All processors in a parallel computer can execute different instructions and operate on different data at the same time.
- Parallelism achieved by connecting multiple processors together
- Shared or distributed memory
- Different programs can be run simultaneously
- Each processor can perform any operation regardless of what other processors are doing.
- Examples: Cray T90, Cray T3E, IBM-SP2

2.2.2. SIMD (Single-Instruction Multi-Data)

- All processors in a parallel computer execute the same instructions but operate on different data at the same time.
- Only one program can be run at a time.
- Processors run in synchronous, lockstep function
- Shared or distributed memory
- Less flexible in expressing parallel algorithms, usually exploiting parallelism on array operations, e.g. F90
- Examples: CM2, MsPar

2.2.3 MISD (Multiple-Instruction Single-Data)

- Special purpose computer

2.2.4 SISD (Single-Instruction Single-Data)

- Serial computer

2.3 Processes and Granularity

On a parallel computer, user applications are executed as processes, tasks or threads. The traditional definition of ***process*** is ***a program in execution***. To achieve an improvement in speed through the use of parallelism, it is necessary to divide the computation into tasks or processes that can be executed simultaneously. The size of a process can be described by its granularity.

2.3.1 Fine-grain

- In fine granularity, a process might consist of a few instructions, or perhaps even one instruction.

2.3.2. Medium-grain

- Medium granularity describes the middle ground between fine-grain and course grain.

2.3.3 Course-grain

- In course granularity, each process contains a large number of sequential instructions and takes a substantial time to execute.

Sometimes granularity is defined as the size of the computation between communication or synchronization points. Generally, we want to increase the granularity to reduce the cost of process creation and interprocess communication, but of course this will likely reduce the number of concurrent processes and the amount of parallelism. A suitable compromise has to be made.

In general, we would like to design a parallel program in which it is easy to vary granularity: i.e. a ***scalable program design.***

2.4 Connection Topology

The best choice would be a fully connected network in which each processor has a direct link to every other processor. Unfortunately, this type of network would be very expensive and difficult to scale. Instead, processors are arranged in some variation of a grid, torus, hypercube, etc. Key issues in network design are the network *bandwidth* and the network *latency*. The *bandwidth* is the number of bits that can be transmitted in unit time, given as bits/sec. The network *latency* is the time to make a message transfer through the network.

2.4.1 Static Interconnects

- Consist of point-to-point links between processors
- Can make parallel system expansion easy
- Some processors may be "closer" than others
- Examples: Line/Ring, Mesh/Torus, Tree, Hypercube

Line/Ring.

- a line consists of a row of processors with connections limited to the adjacent nodes.
- the line can be formed into a ring structure by connecting the free ends.

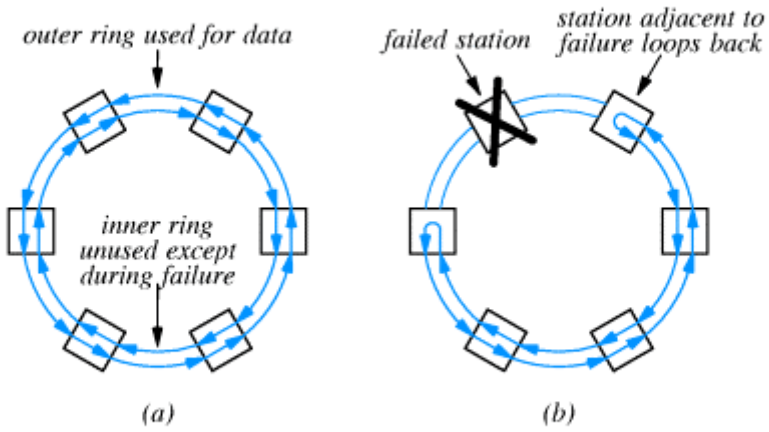


Fig. 2.4.1.a - Ring

Mesh

- processors are connected in rows and columns in a 2 dimensional mesh
- example: Intel Paragon

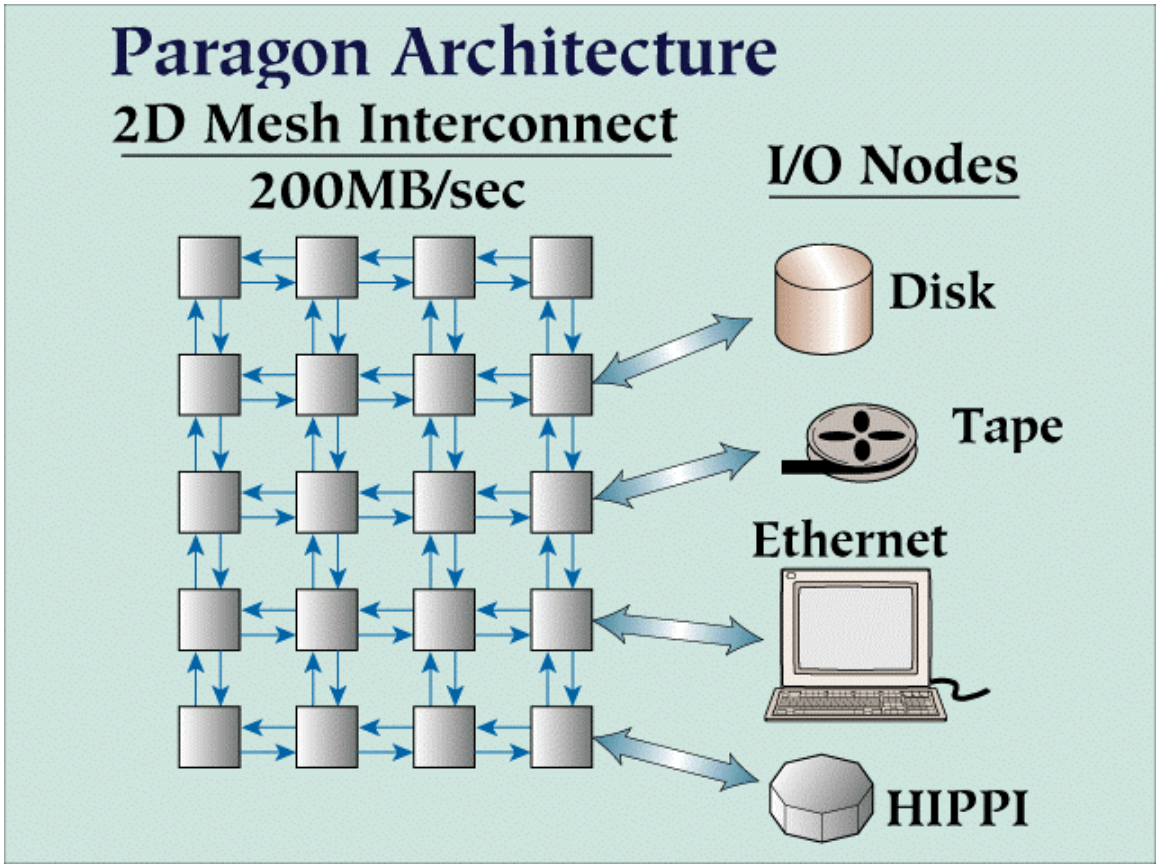


Fig. 2.4.1.b – 2D Mesh

In a mesh network of dimension D, each nonboundary processor is connected to 2D immediate neighbors. Connections typically consist of two wires, one in each direction.

Torus

This architecture extends from the mesh by having wraparound connections. The torus is a symmetric topology, whereas a mesh is not. All added wraparound connections help reduce the torus diameter and restore the symmetry.

- one-dimensional torus
- two-dimensional torus
- three-dimensional torus
- example: Cray T3E

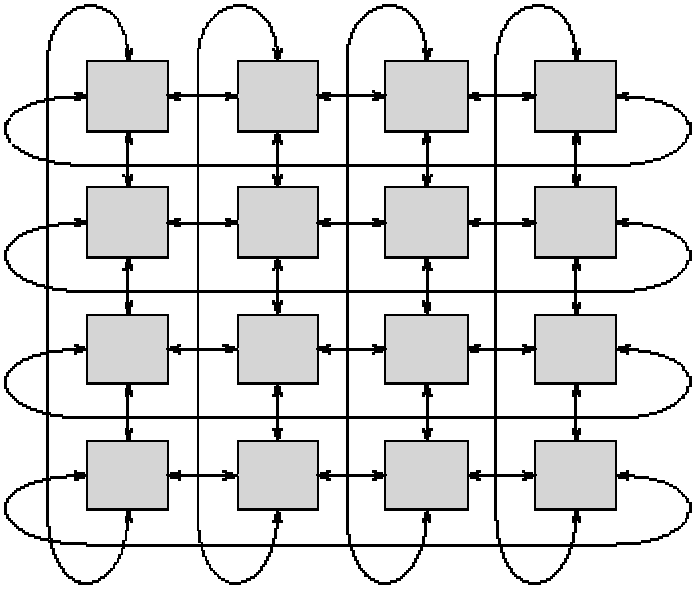


Fig. 2.4.1.c – 2D Torus

Tree

- ***binary tree***
  - first node is called root
  - each node has two links connecting to two nodes below it as the network fans out from the root node
  - At the first level below the root node, there are two nodes. At the next level, there are four nodes, and at the j-th level below the root node there are 2^j nodes.
- ***fat tree***
  - The number of links is progressively increased toward the root.

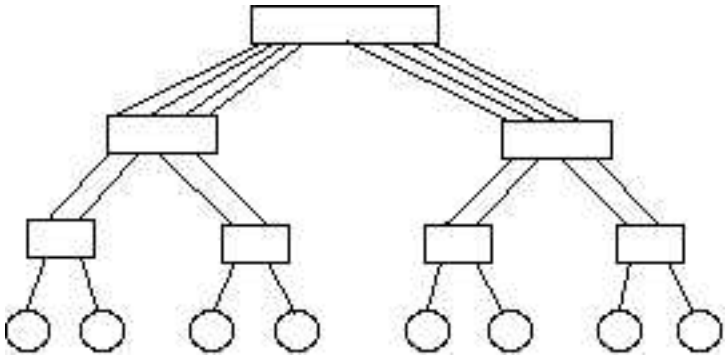


Fig. 2.4.1.d – Fat tree

- ***universal fat tree***
  - number of links between the nodes grows exponentially toward the root, thereby allowing increased traffic toward the root and reducing the communication bottleneck.
  - examples: the Thinking Machine's CM5, Meiko CS2

Hypercube

- each processor connects to 2^n neighbors in a n dimension Hypercube
- examples: iPSC, nCUBE, SGI O2K

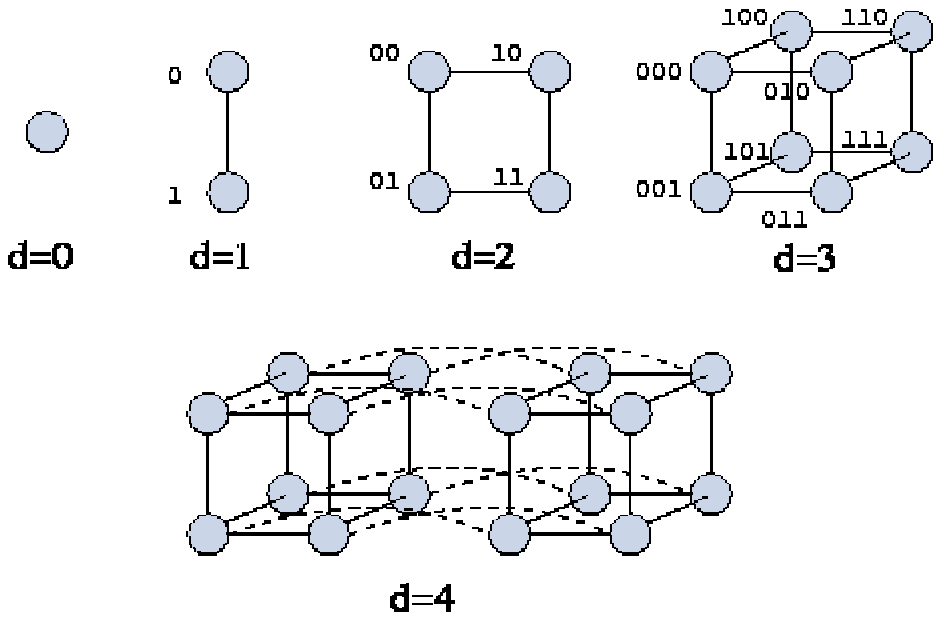


Fig. 2.4.1.e – Hypercubes

Hypercubes of dimension zero through four. The processors in the cubes are labeled with integers, here represented as binary numbers. Two processors are neighbors if and only if their binary labels differ only in one digit place.

2.4.2 Dynamic Interconnects

- Paths are established as needed between processors
- System expansion is difficult
- Processors are usually equidistant
  Examples: Bus-based, Crossbar, Multistage Networks

Bus-based Networks

- In a bus-based network, processors share a single communication resource [the bus].
- A bus is a highly non-scalable architecture, because only one processor can communicate on the bus at a time.
- Used in shared-memory parallel computers to communicate read and write requests to a shared global memory

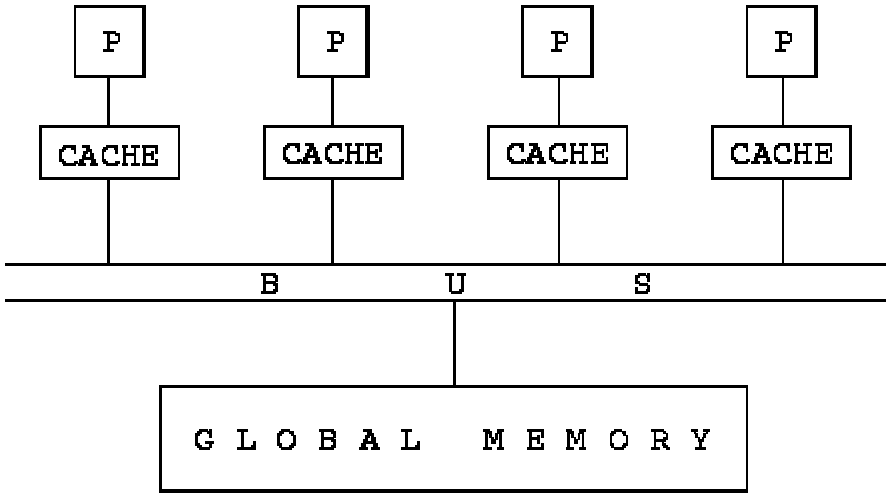


Fig. 2.4.2.a – Bus-based Networks
A bus-based interconnection network, used here to implement a shared-memory parallel computer. Each processor (P) is connected to the bus, which in turn is connected to the global memory. A cache associated with each processor stores recently accessed memory values in an effort to reduce the bus traffic.

Crossbar Switching Network

- A crossbar switch avoids competition for bandwidth by using $O(N^2)$ switches to connect N inputs to N outputs.
- Although highly non-scalable, crossbar switches are a popular mechanism for connecting a small number of workstations, typically 20 or fewer.

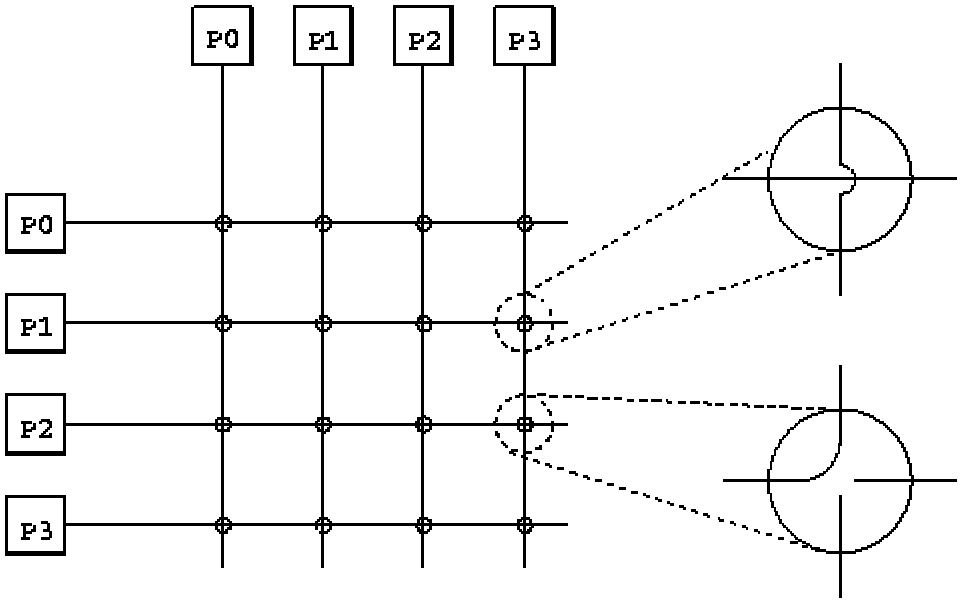


Fig. 2.4.2.b– Crossbar Network

A 4*4 nonblocking crossbar, used here to connect 4 processors. On the right, two switching elements are expanded: the top one is set to pass messages through and the lower one to switch messages. Each processor is depicted twice. Pairs of processors can communicate without preventing other processor pairs from communicating.

Multistage Interconnection Networks

- In a multistage interconnection network (MIN), switching elements are distinct from processors.
- Fewer than $O(p^2)$ switches are used to connect p processors.
- Messages pass through a series of switch stages.
- In a unidirectional MIN, all messages must traverse the same number of wires, and so the cost of sending a message is independent of processor location – in effect, all processors are equidistant.
- In a bi-directional MIN, the number of wires traversed depends to some extent on processor location, although to a lesser extent than in a mesh or hypercube.
- Example: IBM SP networks are bi-directional multistage inter-connection networks:
    - bi-directional, any-to-any inter-node connection: allows all processors to send messages simultaneously.
    - multistage interconnection: on larger systems (over 80 nodes), additional intermediate switches are added as the system is scaled upward

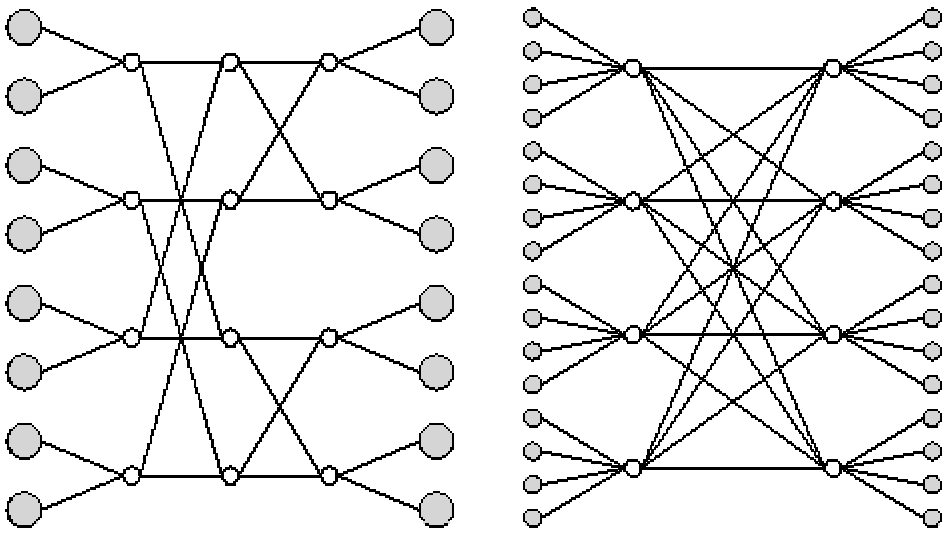
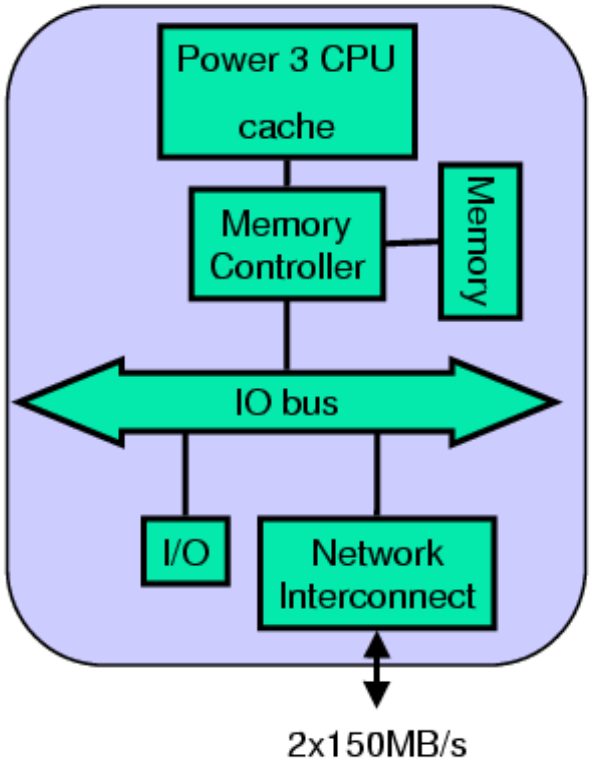


Fig. 2.4.2.c – Multistage interconnection network
Shaded circles represent processors and unshaded circles represent crossbar switches.

2.5 Hardware Specifics – Examples

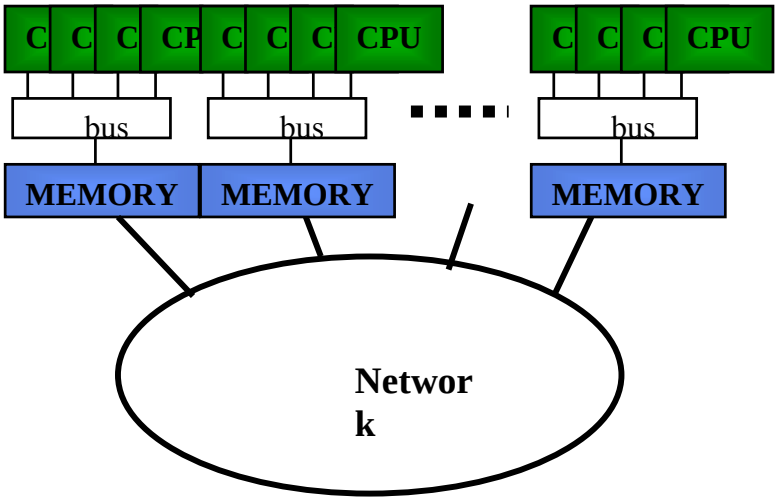
2.5.1 IBM SP2

- Message passing system
- Cluster of workstations
- 200 MHz power 3 CPU
  - Peak 800 MFLOPS
  - 4-16 MB 2nd-level cache
  - sustained memory bandwidth 1.6 GB/s
- Multistage crossbar switch
- MPI
  - Latency 21.7 usec
  - Bandwidth 139 MB/sec
- I/O hardware



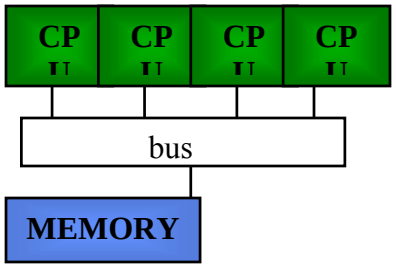
2.5.2 IBM PWR3 – SDSC Blue Horizon

- 222 MHz …888MFLOPS (1152 CPUs, 144 nodes with 8 CPUs (SMP))
- 2 Pipes, 1FMA per pipe per clock tick
- MPI & OpenMP programming
- 32 KB L1 Cache, 2MB L2 Cache



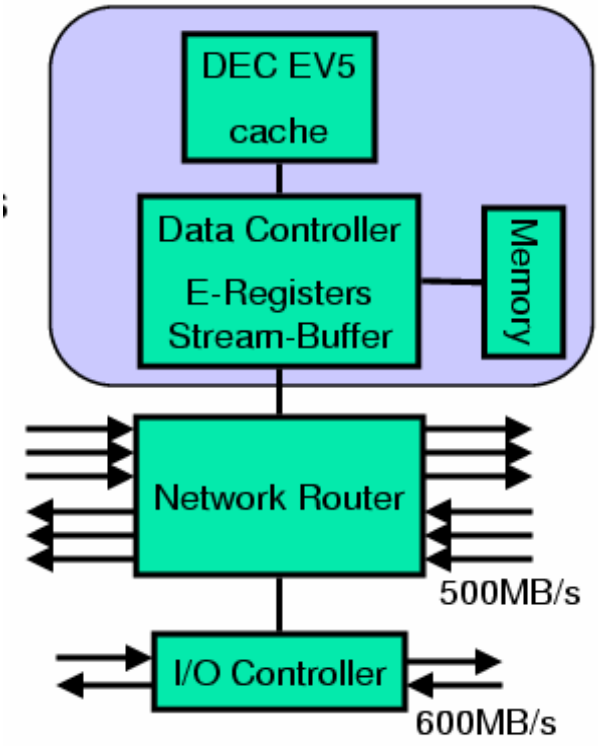
2.5.3 Sun HPC

- 400 MHz …..800 MFLOPS (64 CPUs)
- MPI or OpenMP Programming
- 16 KB L1 Cache, 4MB L2 Cache, 64GB total Main memory
- 2 Pipes, 1 FLOP per pipe per cycle



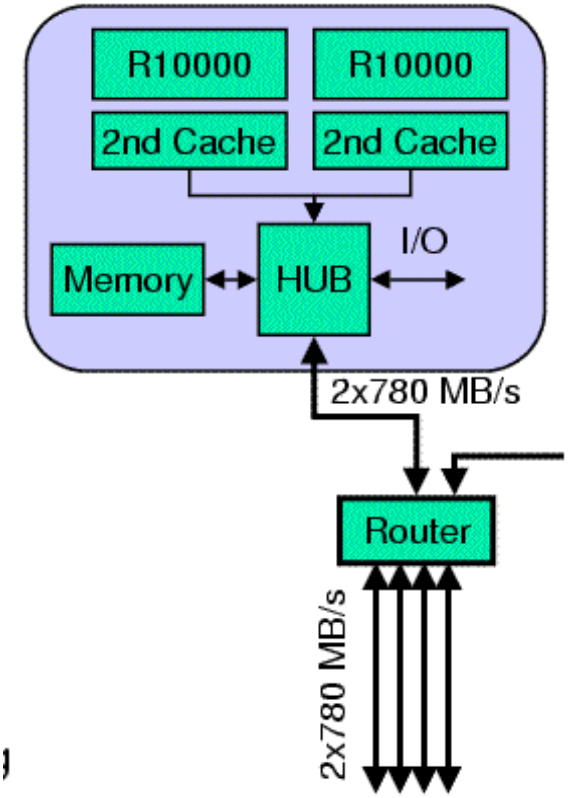
2.5.4 Cray T3E

- Remote memory access system
- Single system image
- 600 MHz DEC Alpha CPU
  - Peak 1200 MFLOPS
  - 96 KB 2nd-level cache
  - Sustained memory bandwidth 600 MB/s
- 3D torus network
- MPI
  - Latency 17 usec
  - Bandwidth 300 MB/s
- Shmem
  - Latency 4 usec
  - Bandwidth 400 MB/s
- SCI-based I/O network



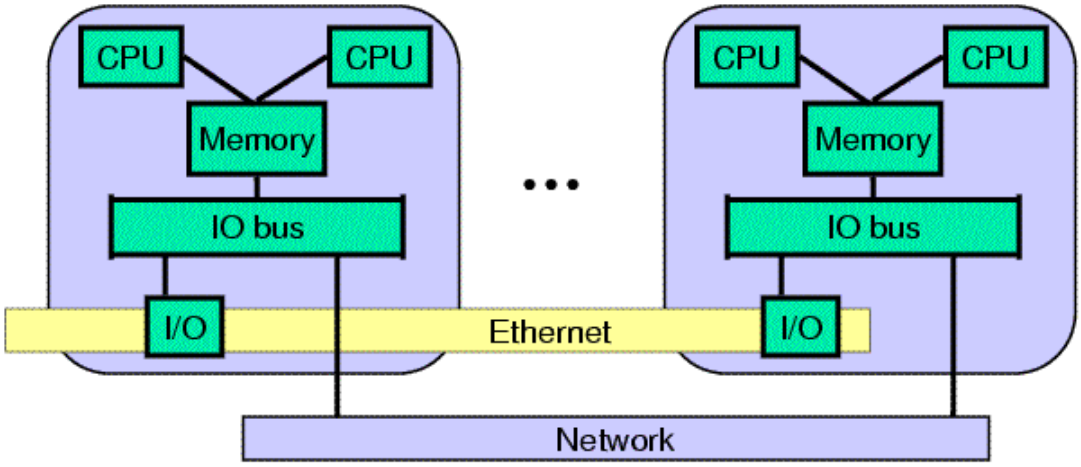
2.5.5 SGI O2K

- Cc-NUMA system
- Single system image
- 600250 MHz MIPS R10000 CPU
  - Peak 500 MFLOPS
  - 2nd-level data cache 4-8 MB
  - Sustained memory bandwidth 670 MB/s
- 4D hypercube
- MPI
  - Latency 16 usec
  - Bandwidth 100 MB/s
- Remote memory access
  - Latency 497 usec
  - Bandwidth 600 MB/s



2.5.6 Cluster of workstations

- Hierarchical architecture: shared memory in a node, message passing across nodes.
- PC-based nodes or workstation-based nodes
- Networks: Myrianet, Scalable Coherent Interface, Gigabit Ethernet



3. PARALLEL PROGRAMMING MODELS

- A parallel computer system should be flexible and easy to use and should exhibit good programmability in supporting various parallel algorithms.
- ***Explicit parallelism*** means that parallelism is explicitly specified in the source code by the programmer using special language constructs, compiler directives or library function calls.
- If the programmer does not explicitly specify parallelism, but lets the compiler and the run-time support system automatically exploit it, we have the ***implicit parallelism.***

3.1 Implicit Parallelism

3.1.1 Parallelizing Compilers

- Automatic parallelization of sequential programs
- Do not exploit functional parallelism
- Compiler performs dependence analysis on a sequential program's source data and then – using a suite of program transformation techniques – converts the sequential code into a native parallel code.
- Some performance studies indicate, however, that the parallelizing compilers are not very effective.

3.2 Explicit Parallelism

Although many explicit programming models have been proposed, three models have become dominant ones: data parallel, message passing and shared variable.

3.2.1 Data parallel

- Execute the same instruction or program segment over different data sets simultaneously on multiple computing nodes.
- Has a single thread of control
- Parallelism is exploited at data set level
- No functional parallelism available

3.2.1.1 Fortran 90

- Uses array syntax to express parallelism
- Implementation on SIMD and MIMD machines
- Single processor versions are available
- Communication is transparent

3.2.1.2 High Performance Fortran (HPF)

- Evolves from Fortran 90, allows for far more detail in expressing parallelism
- Attempt to standardize data parallel programming
- Data distribution and alignment can be defined
- Allows explicit definition of parallelism

3.2.2 Message-passing model

- Multithreading – a message-passing program consists of multiple processes, each of which has its own thread of control and may execute different code. Both control parallelism (MPMD – Multiple-Program-Multiple-Data) and data parallelism (SPMD – Single-Program-Multiple-Data) are supported.
- Asynchronous – the processes of a message-passing program execute asynchronously.
- Separate address space - the processes of a parallel program reside in different address spaces.
- Explicit interactions – the programmer must solve all the interaction issues, including data mapping, communication and synchronization.
- Scales well, especially if data is well distributed.

3.2.2.1 PVM

The PVM (Parallel Virtual Machine) is a software package that permits a heterogeneous collection of Unix and/or NT computers hooked together by a network to be used as a single large parallel computer. Thus large computational problems can be solved most cost effectively by using the aggregate power and memory of many computers. The software is very portable. The source, which is available free thru Netlib [www.netlib.org], has been compiled on everything from laptops to CRAYs.

PVM enables users to exploit their existing computer hardware to solve much larger problems at minimal additional cost. Hundreds of sites around the world are using PVM to solve important scientific, industrial, and medical problems in addition to PVM's use as an educational tool to teach parallel programming.

3.2.2.2 MPI

- MPI (Message Passing Interface) is the standard programming interface
  - MPI 1.0 in 1994
  - MPI 2.0 in 1997
- Library interface (Fortran, C, C++)
- It includes
  - point-to-point communication
  - collective communication
  - barrier synchronization
  - one-sided communication (MPI 2.0)
  - parallel I/O (MPI 2.0)
  - process creation (MPI 2.0)

3.2.3 Shared variable

- Similar to data-parallel model, in that it has single address space
- Similar to message-passing model, in that it is multithreading and asynchronous
- Data reside in a single, shared address space and does not have to be explicitly allocated
- Communication is done implicitly through shared reads and writes of variables
- Synchronization is explicit

3.2.3.1 SGI Power C Model

- extension to the sequential C language with compiler directives (pragmas) and library functions
- supports shared-variable parallel programming
- similar extended constructs are also provided for Fortran
- it is structured and relatively simple

3.2.3.2 OpenMP: Directive-based SM parallelization

- OpenMP is a standard shared memory programming interface(1997)
- directives for Fortran77 and C/C++
- fork-join model resulting in global program
- it includes:
  - parallel loops
  - parallel sections
  - parallel regions
  - shared and private data
  - synchronization primitives
    - barrier
    - critical region

4. Topics in Parallel Computation

4.1 Types of parallelism: two extremes

4.1.1 Data parallel

- Each processor performs the same task on different data
- Data mapping is critical
- Programmed with HPF or message passing
- Example – grid problems

4.1.2 Task parallel

- Each processor performs a different task
- More difficult to balance load
- Commonly programmed with message passing
- Example – signal processing

Most applications fall somewhere on the continuum between these two extremes

4.2 Programming Methodologies

- **Bulk of program in Fortran, C, or C++**
- **Data and/or tasks are split up onto different processors by:**
    - Distributing the data onto local memory of CPU thus causing CPU to work on its local memory (MPPs, MPI).
    - Distribute work of each loop to different CPU's (SMP, OpenMP).
    - Hybrid distribute data onto SMP box and then within the SMP distribute work of each loop to different CPUs within the box (SMP-Cluster, MPI&OpenMP).

4.3 Computation Domain decomposition and Load Balancing

4.3.1 Domain decomposition

- The computation domain is partitioned into several subdomains and then mapped onto processors of a parallel system.
- In general, the number of subdomains equals to the number of processors in a parallel system.

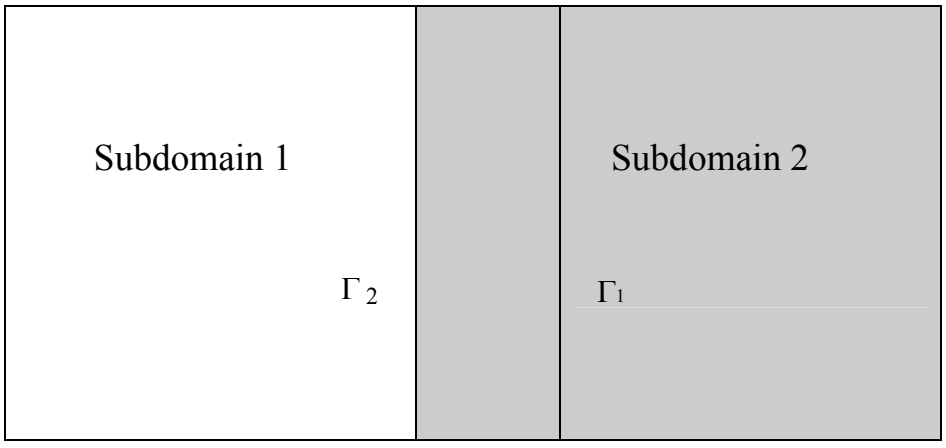
4.3.2 Load Balancing

- The goal of partitioning is to distribute the computation load such that all processors can finish their computation at about the same time.
- For homogeneous parallel systems, the computation load is distributed as evenly as possible in a parallel computer.
- For heterogeneous parallel system, the computation load is distributed according to the computing power of each processor.

4.3.3 Overlapping Subdomains and Non-Overlapping Subdomains:

4.3.3.1 Overlapping Subdomains

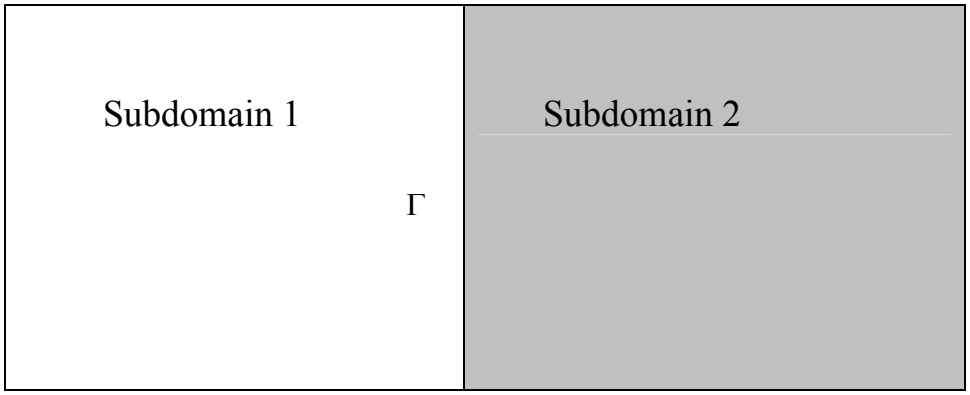
- There is a common computation domain between two adjacent subdmains.



- Mathematical formulations are applied on Γ_1 and Γ_2
- Difficult to deal with irregular overlapping areas.

4.3.3.2 Non-overlapping Subdomains

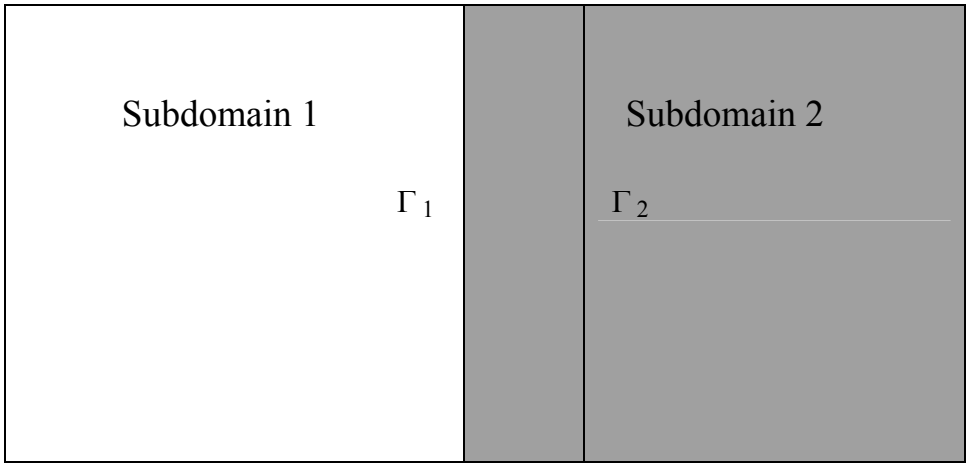
- There is only an interface between two adjacent subdomains



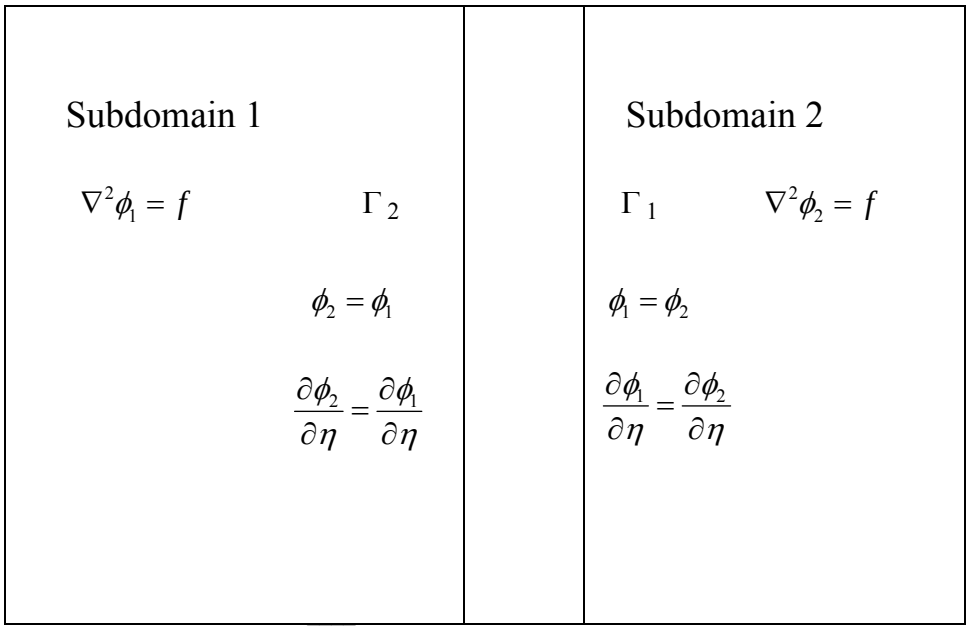
- Mathematical formulations are applied on Γ.
- Can handle irregular interfaces easily.

4.3.4 Domain Decomposition for Numerical Analysis

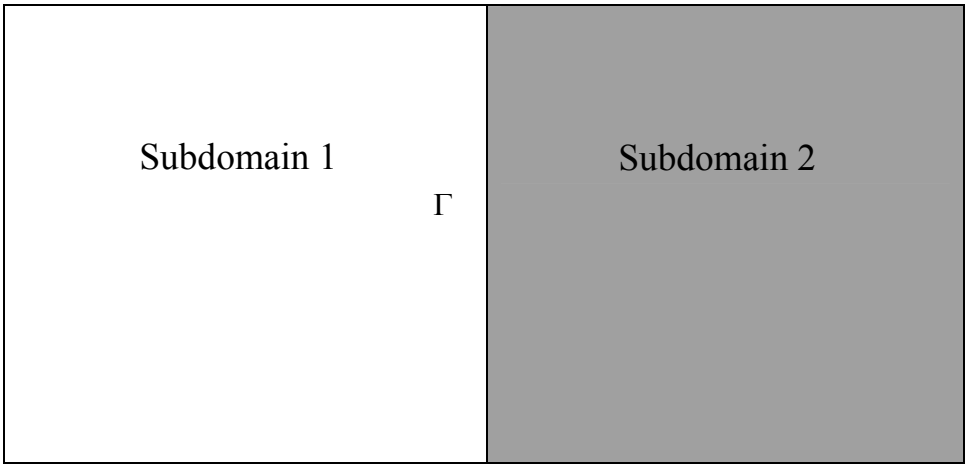
Overlapping Subdomains



Domain Decomposition



Non-overlapping Subdomains



Domain Splitting

Subdomain 1	Subdomain 2
Γ	
$\nabla^2 \phi_1^{(n)} = f$	$\nabla^2 \phi_2^{(n)} = f$
$\phi_1^{(n)} = g^{(n)}$	$\frac{\partial \phi_2^{(n)}}{\partial \eta} = \frac{\partial \phi_1^{(n)}}{\partial \eta}$
$g^{(n+1)} = \theta \phi_2^{(n)} + (1-\theta) g^{(n)}$	
(D)	(N)

Interface Relaxation Process

Iterative Scheme 1:

1. Solve interior completely.
2. Update the interface data.
3. Repeat 1. and 2. until convergence on the interface.

Iterative Scheme 2:

1. One iteration for the interior mesh points of both subdomains.
2. Update the interface mesh points.
3. Continue 1. and 2. until convergence of all mesh points.

4.4 Numerical Solution Methods

4.4.1 Iterative Solution Methods

4.4.1.1 Parallel SOR (successive over-relaxation)

4.4.1.1.1 Parallel SOR Iterative Algorithms for the Finite Difference Method.

One dimensional example:

$$\frac{d^2\phi}{dx^2} = 1$$

Difference equation:

$$\phi_{j+1} - 2\ \phi_j + \phi_{j-1} = \Delta x^2 \qquad j=2,\ldots,N-1$$

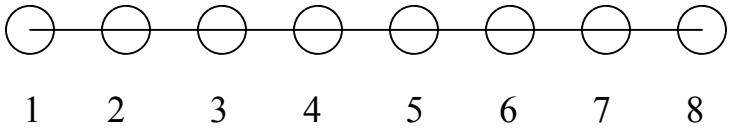
SOR Iterative Scheme:

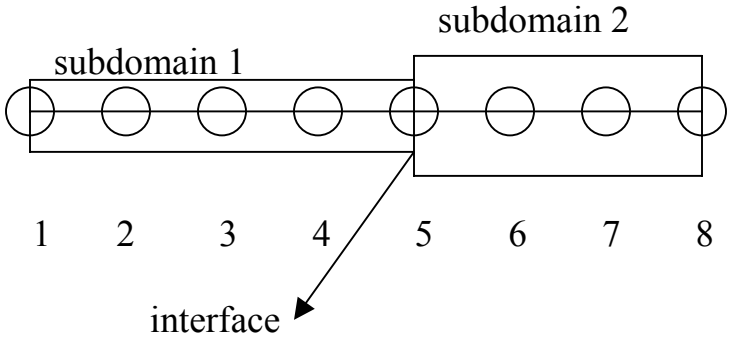
$$\phi_j^{(n+1/2)} = (\ \phi_{j+1}^{(n)} + \phi_{j-1}^{(n+1)} - \Delta x^2)/2$$

$$\phi_j^{(n+1)} = \alpha\ \phi_j^{(n+1/2)} + (1 - \alpha)\ \phi_j^{(n)}$$

Expand to Matrix Form:

$$\begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ -1 & 2 & -1 & 0 & 0 & 0 & 0 & 0 \\ 0 & -1 & 2 & -1 & 0 & 0 & 0 & 0 \\ 0 & 0 & -1 & 2 & -1 & 0 & 0 & 0 \\ 0 & 0 & 0 & -1 & 2 & -1 & 0 & 0 \\ 0 & 0 & 0 & 0 & -1 & 2 & -1 & 0 \\ 0 & 0 & 0 & 0 & 0 & -1 & 2 & -1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} \varphi_1 \\ \varphi_2 \\ \varphi_3 \\ \varphi_4 \\ \varphi_5 \\ \varphi_6 \\ \varphi_7 \\ \varphi_8 \end{pmatrix} = \Delta x^2 \begin{pmatrix} \varphi_1/\Delta x^2 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ \varphi_8/\Delta x^2 \end{pmatrix}$$





Reorder Equations:

$$\begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ -1 & 2 & -1 & 0 & 0 & 0 & 0 & 0 \\ 0 & -1 & 2 & -1 & 0 & 0 & 0 & 0 \\ 0 & 0 & -1 & 2 & 0 & 0 & 0 & -1 \\ 0 & 0 & 0 & 0 & 2 & -1 & 0 & -1 \\ 0 & 0 & 0 & 0 & -1 & 2 & -1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & -1 & -1 & 0 & 0 & 2 \end{pmatrix} \begin{pmatrix} \varphi_1 \\ \varphi_2 \\ \varphi_3 \\ \varphi_4 \\ \varphi_6 \\ \varphi_7 \\ \varphi_8 \\ \varphi_5 \end{pmatrix} = \Delta x^2 \begin{pmatrix} \varphi_1/\Delta x^2 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ \varphi_8/\Delta x^2 \\ 1 \end{pmatrix}$$

Subdomain 1: φ_2, φ_3, φ_4

Interface: φ_5

Subdomain 2: φ_6, φ_7

Two Dimensional Example:

$$\frac{\partial^2 \phi}{\partial x^2} + \frac{\partial^2 \phi}{\partial y^2} = 1$$

Difference Equation:

$$c_1(\varphi_{i+1,j} - 2\varphi_{i,j} + \varphi_{i-1,j}) + c_2(\varphi_{i,j+1} - 2\varphi_{i,j} - \varphi_{i,j-1}) = 1$$

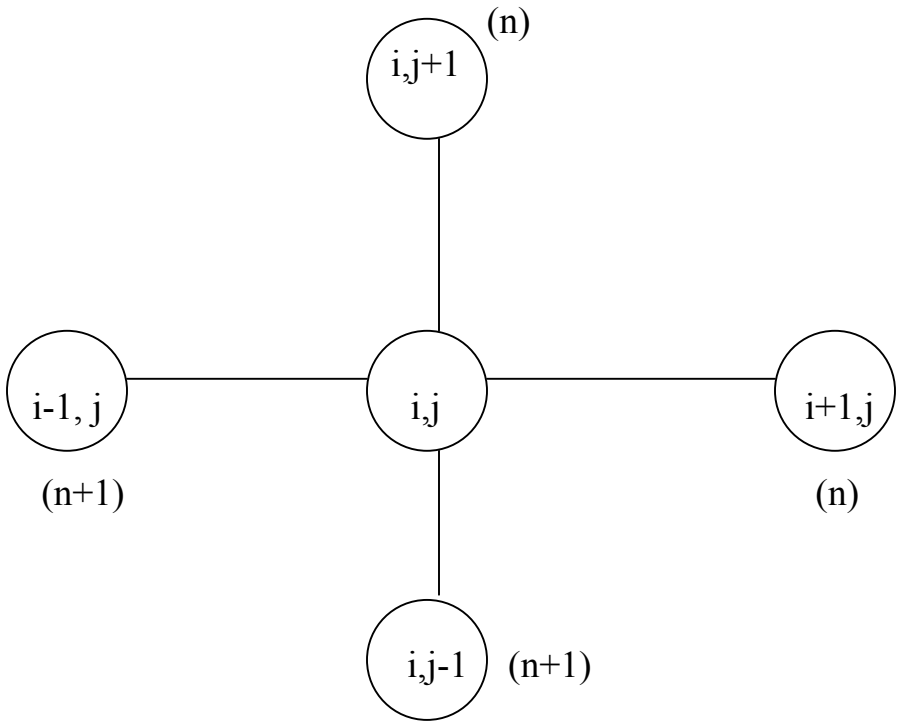
SOR Iterative Scheme:

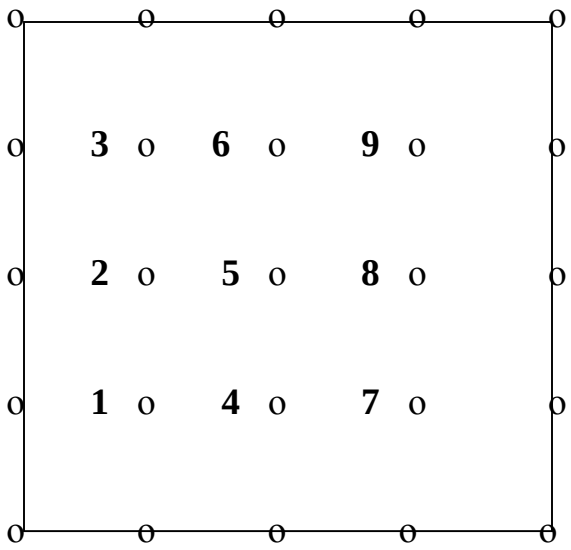
$$\varphi_{i,j}^{(n+1/2)} = (c_1/c_3)\,(\varphi_{i+1,j}^{(n)} + \varphi_{i-1,j}^{(n+1)}) + (c_2/c_3)\,(\varphi_{i,j+1}^{(n)} + \varphi_{i,j-1}^{(n+1)}) - 1/c_3$$

$$\varphi_{i,j}^{(n+1)} = \alpha\, \varphi_{i,j}^{(n+1/2)} + (1-\alpha)\varphi_{i,j}^{(n)}$$

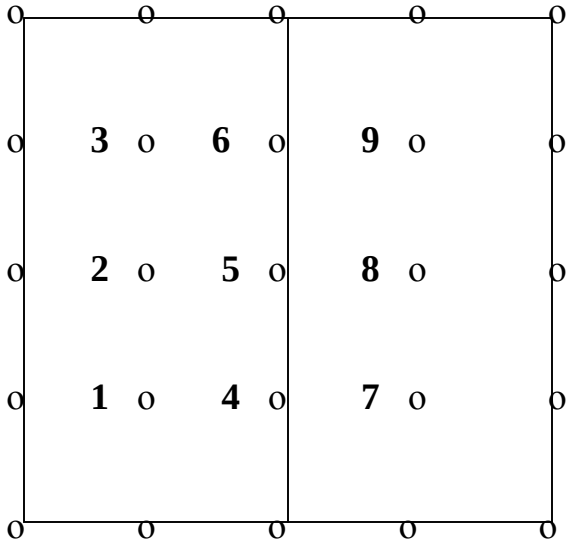
where:

$c_1 = 1/\Delta x^2$, $c_2 = 1/\Delta y^2$ and $c_3 = 2/\Delta x^2 + 2/\Delta y^2$





Reorder Equations:

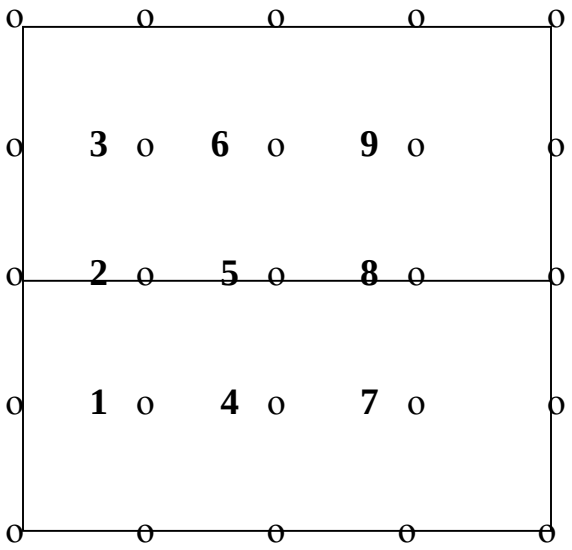


Column type subdomains:

Subdomain 1: 1, 2, 3

Subdomain 2: 7, 8, 9

Interface: 4, 5, 6

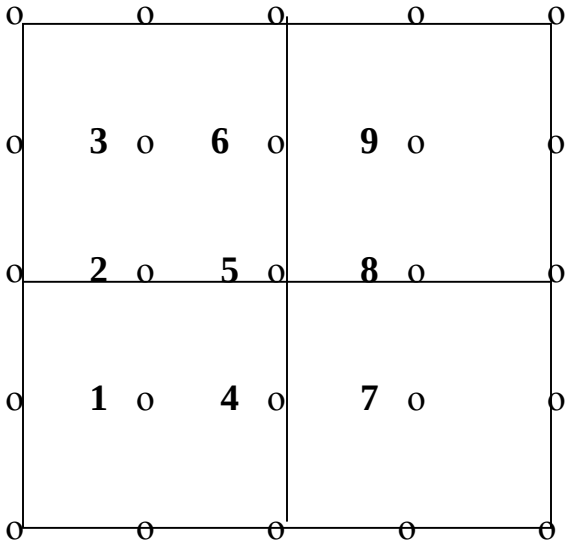


Row type subdomains:

Subdomain 1: 1, 4, 7

Subdomain 2: 3, 6, 9

Interface: 2, 5, 8



Block type subdomains:

Subdomain 1: 1

Subdomain 2: 7

Subdomain 3: 3

Subdomain 4: 9

Interface: 2, 8, 4, 5, 6

4.4.1.1.2 Parallel SOR Iterative Algorithms for the Finite Element Method.

The General Form of a Finite Element System:

$$\begin{pmatrix} k_{11} & k_{1i} & 0 \\ k_{i1} & k_{ii} & k_{i2} \\ 0 & k_{2i} & k_{22} \end{pmatrix} \begin{pmatrix} u_1 \\ u_i \\ u_2 \end{pmatrix} = \begin{pmatrix} f_1 \\ f_i \\ f_2 \end{pmatrix}$$

SOR Iterative Scheme:

$$k_{11}u_1^{(n+1/2)} = f_1 - k_{1i}\, u_i^{(n)}$$

$$u_1^{(n+1)} = \alpha u_1^{(n+1/2)} + (1-\alpha)u_1^{(n)} \qquad (1)$$

$$k_{ii}u_i^{(n+1/2)} = f_i - k_{i1}u_1^{(n+1)} - k_{i2}u_2^{(n)}$$

$$u_i^{(n+1)} = \alpha u_i^{(n+1/2)} + (1-\alpha)u_i^{(n)} \qquad (2)$$

$$k_{22}u_2^{(n+1/2)} = f_2 - k_{2i}u_i^{(n+1)}$$

$$u_2^{(n+1)} = \alpha u_2^{(n+1/2)} + (1-\alpha)u_2^{(n)} \qquad (3)$$

Reorder Equations:

$$\begin{pmatrix} k_{11} & 0 & k_{1i} \\ 0 & k_{22} & k_{2i} \\ k_{i1} & k_{i2} & k_{ii} \end{pmatrix} \begin{pmatrix} u_1 \\ u_2 \\ u_i \end{pmatrix} = \begin{pmatrix} f_1 \\ f_2 \\ f_i \end{pmatrix}$$

Parallel SOR Iterative Scheme:

$$k_{11}u_1^{(n+1/2)} = f_1 - k_{1i}\, u_i^{(n)}$$

$$u_1^{(n+1)} = \alpha u_1^{(n+1/2)} + (1-\alpha)u_1^{(n)} \qquad (4)$$

$$k_{22}u_2^{(n+1/2)} = f_2 - k_{2i}u_i^{(n)}$$

$$u_2^{(n+1)} = \alpha u_2^{(n+1/2)} + (1-\alpha)u_2^{(n)} \qquad (5)$$

$$k_{ii}u_i^{(n+1/2)} = f_i - k_{i1}u_1^{(n+1)} - k_{i2}u_2^{(n+1)}$$

$$u_i^{(n+1)} = \alpha u_i^{(n+1/2)} + (1-\alpha)u_i^{(n)} \qquad (6)$$

4.4.1.2 Conjugate Gradient Method

Conjugate Gradient (CG) Method is a popular iterative method for solving large systems of linear equations. CG is effective for systems of the form:

$$A\,x = b$$

where x is an unknown vector, b is a known vector, and A is a known, square, symmetric, positive-definite (or positive-indefinite) matrix. This system arises in many important settings, such as using finite difference and finite element methods for solving partial differential equations, structural analysis and circuit analysis.

4.4.1.2.1 Conjugate Iterative Procedure

$$d_{(0)} = r_{(0)} = b - Ax_{(0)}$$

$$\alpha_{(i)} = \frac{r^T{}_{(i)} r_{(i)}}{d^T{}_{(i)} A d_{(i)}}$$

$$x_{(i+1)} = x_{(i)} + \alpha_{(i)} d_{(i)}$$

$$r_{(i+1)} = r_{(i)} - \alpha_{(i)} A d_{(i)}$$

$$\beta_{(i+1)} = \frac{r^T{}_{(i+1)} r_{(i+1)}}{r^T{}_{(i)} r_{(i)}}$$

$$d_{(i+1)} = r_{(i+1)} + \beta_{(i+1)} d_{(i)}$$

4.4.1.3. Multigrid Method

Many standard iterative methods (i.e. Jacobi, SOR, Gauss-Seidel) possess the smoothing property. This property makes these methods very effective at eliminating the high-frequency or oscillatory components of the error, while leaving the low-frequency or smooth components relatively unchanged.

One way to improve a relaxation scheme, at least in its early stages, is to use a good initial guess. A known technique for obtaining an improved initial guess is to perform some preliminary iterations on a coarse grid and then use the resulting approximation as an initial guess on the original fine grid.

Relaxation on a coarser grid is less expensive since there are fewer unknowns to be updated. Also, since the convergence factor behaves like 1-$O(h^2)$, the coarser grid will have a marginally improved convergence rate.

The linear system of equations considered is:

$$Ax = b$$

4.4.1.3.1 First Strategy

1. Relax on Ax=b on a very coarse grid.
2. …
3. …
4. …
5. Relax on Ax=b on Ω^{4h} to obtain an initial guess for Ω^{2h}.
6. Relax on Ax=b on Ω^{2h} to obtain an initial guess for Ω^{h}.
7. Relax on Ax=b on Ω^{h} to obtain a final approximation to the solution.

4.4.1.3.2 Second Strategy (Coarse Grid Correction)

1. Relax on Ax=b on Ω^h to obtain an approximation v^h.
2. Compute the residual $r = b - Av^h$.
3. Relax on the residual equation Ae=r to obtain an approximation to the error e^{2h}.
4. Correct the approximation obtained on Ω^h with the error estimate obtained on
$$\Omega^{2h} : v^h \leftarrow v^h + e^{2h}.$$

Transformation between grids.
Interpolation (prolongation)

1. Operator: I_{2nh}^{nh}.
2. Transferring the data from a coarse grid Ω^{2nh} to a finer grid Ω^{nh}.
3. Linear interpolation can be used.

Injection (restriction)

1. Operator: I_{2nh}^{nh}
2. Moving data from a finer grid Ω^{nh} to a coarser grid Ω^{2nh}.
3. Data on the same grid can be used directly.
4. Full weighting can also be used.

Coarse Grid Correction Scheme: $v^h \leftarrow CG(v^h, b^h)$.
Relax v_1 times on $A^h x^h = b^h$ with initial guess v^h.
Compute $r^{2h} = I_h^{2h}(b^h - A^h v^h)$.
Solve $A^{2h} e^{2h} = r^{2h}$ on Ω^{2h}.
Correct fine grid approximation: $v^h \leftarrow v^h + I_{2h}^h e^{2h}$.
Relax v_2 times on $A^h x^h = b^h$ on Ω^h with initial guess v^h.

4.4.2 Direct Solution Method

4.4.2.1 Gauss Elimination Method

The Gauss Elimination Method is the most used direct solver for the linear system:

$$Ax = b$$

where A is a known, square, positive definite and dense system.

The general procedure for Gauss elimination is to factor the A matrix into an upper-triangular matrix:

$$Ux = y$$

Then use back substitution to obtain the solution x.

4.4.2.1.1.Gauss Elimination Procedure:

The Gaussian elimination algorithm can be written in algorithmic form as shown:

For $k = 1,...,n-1$

  For $i = k+1,...,n$

$$l_{ik} = \frac{a_{ik}}{a_{kk}}$$

  For $j = k+1,...,n$

$$a_{ij} = a_{ij} - l_{ik}a_{kj}$$

$$b_i = b_i - l_{ik}b_k$$

(a) Forward Reduction

For $k = n, n-1,...,1$

$$x_k = b_k$$

  For $i = k+1,...,n$

$$x_k = x_k - a_{ki}x_i$$

$$x_k = \frac{x_k}{a_{kk}}$$

(b) Back Substitution

5. REFERENCES

1. K. Hwang, Z. Xu, " Scalable Parallel Computing", Boston: WCB/McGraw-Hill, c1998.

2. I. Foster, " Designing and Building Parallel Programs", Reading, Mass: Addison-Wesley, c1995.

3. D. J. Evans, "Parallel SOR Iterative Methods", Parallel Computing, Vol.1, pp. 3-8, 1984.

4. L. Adams, "Reordering Computations for Parallel Execution", Commun. Appl. Numer. Methods, Vol.2, pp 263-271, 1985.

5. K. P. Wang and J. C. Bruch, Jr., "A SOR Iterative Algorithm for the Finite Difference and Finite Element Methods that is Efficient and Parallelizable", Advances in Engineering Software, 21(1), pp. 37-48, 1994.

6. K. P. Wang and J. C. Bruch, Jr., "An Efficient Iterative Parallel Finite Element Computational Method", The Mathematics of Finite Elements and Applications, edited by J. R. Whiteman, John Wiley and Sons, Inc., Chapter 12, pp. 179-188, 1994.