Lecture Notes on Parallel Scientific Computing

Tao Yang
Department of Computer Science
University of California at Santa Barbara
tyang@cs.ucsb.edu
http://www.cs.ucsb.edu/~tyang
Last update: November 1997

Contents

1 Introduction

- Scientific computing uses computers to analyze and solve scientific and engineering problems based on mathematical models. Applications of scientific computing include computer graphics, image Processing, stock analysis, weather, prediction, seismic data processing, design simulation (airplanes/cars), computational medicine, astrophysics, and nuclear Engineering.
- An example of Parallel Computing. Summation of 8 numbers in parallel.
- Why Parallel Computing: solving large-scale problems. Many scientific problems can be reduced to the problem of solving linear equations. Time and space complexity of solving n equations on SUN Workstations (Sparc 10):

n	Time $2n^3/3$	Space $8n^2$ bytes
100	0.067 sec	80 KB
1000	1.1 min	8MB
10,000	18 hr	800MB
1,000,000	2 years	80TB

- Performance of current commercial machines. MF (Mega Flop, 1 million floating point operations per second). GF (GigaFlop)=1000MF.

	Peak	Benchmark
Cray Y-MP, 1 proc	952 MF	874 MF
Cray Y-MP, 4 proc	3.27 GF	
SGI Indigo 2		15 MF
SUN 4		2.7 MF
SUN SPARC 10		9.3 MF
nCUBE, 1000 nodes	2.4GF	
Intel/Delta, 512 nodes	20 GF	446 MF
Meiko, 1 node	40 MF	8 MF
Meiko, 64 nodes	2.56 GF	8 MF

- Applications of parallel computing: Large-scale scientific computing, Parallel data bases (credit cards), multi-media information servers, etc.
- Computer architectures: sequential machines, vector machines, parallel machines.
- Parallel architectures:
  - *Control mechanism.* SIMD type (single instruction stream multiple data stream, e.g. MasPar MP-1, CM-2). MIMD (multiple instruction stream multiple data stream, nCUBE 2, Intel Paragon, CM-5).
  - *Address-Space Organization.* Message passing architecture (Intel Paragon, nCUBE 2, CM-5, Cray T3D, IBM SP2). vs. Shared Address space Architecture (KSR-2, Stanford Dash). Nonuniform memory access (NUMA).
- Basic steps of parallel programming.
  1. Preparing parallelism (task partitioning, program dependence analysis).

  2. Mapping and scheduling of parallelism.
  3. Coding and debugging.
  4. Evaluating parallel performance.
- Performance evaluation.

$$Speedup = \frac{\text{Sequential time}}{\text{Parallel Time}}.$$

$$Efficiency = \frac{\text{Speedup}}{\text{No of Processors used}}.$$

- Performance is restricted by the availability of parallelism: Amdahl's Law.

  Let T_1 be the sequential time of a program. Assume that α is the fraction of this program that has to be executed *sequentially*. Only $(1-\alpha)$ fraction of this program can be executed concurrently. The shortest parallel time for this program is: $T_p = \alpha T_1 + (1-\alpha)\frac{T_1}{p}$.

  The speedup bound is

$$Speedup = \frac{T_1}{\alpha T_1 + (1-\alpha)T_1/p} = \frac{1}{\alpha + (1-\alpha)/p}.$$

  When $\alpha = 0$, $Speedup = p$. And when $\alpha = 0.5$, $Speedup < 2$!

2 Design and Implementation of Parallel Algorithms

2.1 A simple model of parallel computation

- *Representation of Parallel Computation: Task model.*

  A task is an indivisible unit of computation which may be an assignment statement, a subroutine or even an entire program. We assume that tasks are convex, which means that once a task starts its execution it can run to completion without interrupting for communications.

  Dependence. There exists a dependence between tasks. A task T_y depends on T_x, then there is a dependence edge from T_x to T_y. Task nodes and their dependence constitute a graph which is **a directed acyclic task graph** (DAG).

  Weights. Each task T_x can have a computation weight τ_x representing the execution time of this task. There is a cost $c_{x,y}$ in sending a message from one task T_x to another task T_y if they are assigned to different processors.

- *Execution of Task Computation.*

  Architecture model. Let us first assume message-passing architectures. Each processor has its own local memory. Processors are fully connected.

  Task execution. In the task computation, a task waits to receive all data before it starts its execution. As soon as the task completes its execution it sends the output data to all successors.

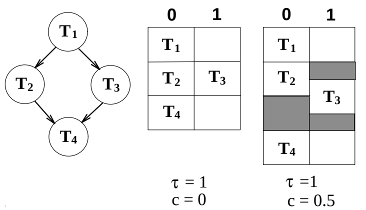
  Scheduling is defined by a processor assignment mapping, $PA(T_x)$, of the tasks T_x onto the p processors and by a starting time mapping, $ST(T_x)$, of all nodes onto the real positive numbers set. $CT(T_x) = ST(T_x) + \tau_x$ is defined as the completion time of task T_x in this schedule.

  Dependence Constraints. If a task T_y depends on T_x, T_y cannot start until the data produced by T_x is available in the processor of T_y. i.e.

$$ST(T_y) - ST(T_x) \geq \tau_x + c_{x,y}.$$

  Resource constraints Two tasks cannot be executed in the same processor, and time.

Fig. 1(a) shows a weighted DAG with all computation weights assumed to be equal to 1. Fig. 1(b) and (c) show the schedules with different communication weight assumptions. Both (b) and (c) use *Gantt charts* to represent schedules. A Gantt chart completely describes the corresponding schedule since it defines both $PA(n_j)$ and $ST(n_j)$. The PA and ST values for schedule (b) is summarized in Figure 1(d).



	T_1	T_2	T_3	T_4
PA	0	0	1	0
ST	0	1	1	2

(a) (b) (c) (d)

Figure 1: (a) A DAG with node weights equal to 1. (b) A schedule with communication weights equal to 0. (c) A schedule with communication weights equal to 0.5. (d) The PA/ST values for schedule (b).

Difficulty. Finding the shortest schedule for a general task graph is hard (known as NP-complete).

- *Evaluation of Parallel Performance.* Let p be the number of processors used.

$$\text{Sequential time} = \text{Summation of all task weights.}$$

$$\text{Parallel time} = \text{Length of the schedule.}$$

$$Speedup = \frac{\text{Sequential time}}{\text{Parallel Time}}, \quad Efficiency = \frac{\text{Speedup}}{p}.$$

- **Performance bound**.

  Let *the degree of parallelism* be the maximum size of independent task sets. Let *the critical path* be the path in the task graph with the longest length (including node computation weights only).

  Then

$$\text{Parallel time} \geq \text{Length of the critical path}, \quad \text{Parallel time} \geq \frac{\text{Sequential time}}{p}$$

$$\text{Speedup} \leq \frac{\text{Sequential time}}{\text{Length of the critical path}}, \quad \text{Speedup} \leq \text{Degree of parallelism}.$$

Example. For Figure 2, assume that $p = 2$, all task weights $\tau = 1$. Thus we have $Seq = 9$. We do not know the communication weights. One of the maximum independent set is $\{x_3, y, z\}$. Thus the degree of parallelism is 3. One critical path is $\{x_1, x_2, x_3, x_4, x_5\}$ with length 5 (noticed that the edge communication weights are not included). Thus

$$PT \geq max(Length(CP), \frac{\mathtt{seq}}{p}) = max(5, \frac{9}{2}) = 5.$$

$$\mathtt{Speedup} \leq \frac{\mathtt{Seq}}{5} = \frac{9}{5} = 1.8.$$

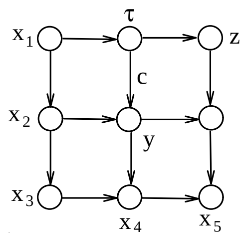


Figure 2: A DAG with node weights equal to 1.

2.2 SPMD parallel programming on Message-Passing Machines

SPMD programming style – single program multiple data.

- Data and program are distributed among processors, code is executed based on a predetermined schedule.
- Each processor executes the same program but operates on different data based on processor identification.

Data communication and synchronization are implemented via message exchange. Two types:

- *Synchronous* message passing, a processor sends a message and waits until the message is received by the destination.
- *Asynchronous* message passing. Sending is non-blocking and the processor that executes the sending procedure does not have to wait for an acknowledgment from the destination.

Programming Languages: C/Fortran + library functions.

Library functions:

- mynode()– return the processor ID. p processors are numbered as $0, 1, 2, \cdots, p-1$.
- numnodes()– return the number of processors allocated.
- *send(data,dest)*. Send data to a destination processor.
- *receive(addr)*. Executing *receive*() will get a specific or *any* message from the local communication buffer and store it in the space specified by *addr*.
- *broadcast(data)*: Broadcast a message to all processors.

Examples.

- SPMD code:
```
Print "hello";
```
  Execute in 4 processors. The screen is:
```
hello
hello
hello
hello
```

- SPMD code:

```
x=mynode();
If x > 0, then Print "hello from " x.
```

Screen:

```
hello from 1
hello from 2
hello from 3
```

- Program is:

```
x=3
For i = 0 to  p-1.
   y(i)= i*x;
Endfor
```

SPMD code:

```
int x, y, i;
x=3;
i=mynode();
y=i*x;
```

or

```
int x, y, i;
i=mynode();
if (i==0) then x=3; broadcast(x).
else receive(x).
y=i*x;
```

3 Issues in Processor Network and Communication

In this section we discuss several issues:

- *Network Embedding.* Difference parallel machines may use different networks to connect their processors. We address how a program running on one network can be simulated on another network.
- *Message routing for one-to-one communication.* We discuss how a message is routed in the network and present a cost function to model the communication delay.
- *Many-to-many communication.* We discuss how communication primitives such as broadcasting can be implemented in different architectures.

3.1 Network Embedding

Given two processor networks, $G = (V, E)$, $G'(V', E')$, we want to a mapping of nodes in G onto nodes in G's. We define it as:

$$v \quad \rightarrow \quad m(v).$$

where v is a node in the processor node set V in G.

Figure 3 gives an example of the embedding. Four processors in a linear processor array G are mapped onto a ring G'. The mapping function is:

$$m(1) = 0, m(2) = 2, m(3) = 3, m(4) = 1.$$

An embedding is called *perfect* if for any edge (n_1, n_2) in G, then $(m(n_1), m(n_2))$ is also an edge in G'. Figure 3 is a perfect embedding. It is not always possible to have a perfect mapping. If $(m(n_1), m(n_2))$ is not an edge in G', then this communication edge must be simulated using a path in G'. The longer this path is, the more overhead this communication will have. The goal of embedding is to minimize the length of such a path. Thus we measure the overall quality of embedding using the following terms.

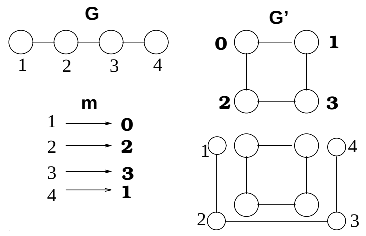


Figure 3: An example of processor embedding.

- The **distance** between two processors is the length of the shortest path connected these two processors.
- For edge (n_1, n_2) in G, Let *map_distance* to be the processor distance of $m(n_1)$ and $m(n_2)$ in G'. **Define** the *dilation* to be the maximum of the *map_distances* for all edges in G.

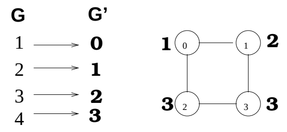


Figure 4: Another example of processor embedding.

Thus the goal of optimization is to minimize the dilation. For a perfect embedding such as Figure 3, the dilation of embedding is always 1. Figure 4 shows another mapping with dilation as 2. This is because Map_Distance(1,2) = 1, Map_Distance(2,3) = 2, Map_Distance(3,4) = 1.

Given an embedding from one network G to another network G', we can use its mapping function to simulate programs of G on G'. For example, given a SPMD program (called *prog*(x), x is the processor identification number) which is designed for a linear array of 8 processors, we need to execute it on a 2×4 mesh. Figure 5 shows an embedding of this linear array on the mesh.

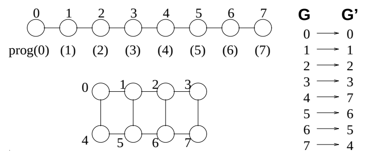


Figure 5: Execute a linear array SPMD program on a mesh.

We can write a SPMD code below on the mesh to run the linear array program, as shown in Figure 6. Notice that the processor identification numbers in the linear array is different from those in the mesh and a re-mapping in the SPMD code is needed. This code in Figure 6 enumerates all cases and if the number of processors is unknown, then this approach will not be feasible. If the number of processors is p, the mapping function $m(i)$ can be generalized as: If $i < \frac{p}{2}$ then $m(i) = i$, else $m(i) = \frac{3p}{2} - 1 - i$. Then we can write a generalized SPMD code as shown in Figure 7.

```
me = mynode();
if (me==0) call prog(0)
else if (me==1) call prog(1)
else if (me==2) call prog(2)
else if (me==3) call prog(3)
else if (me==4) call prog(7)
else if (me==5) call prog(6)
else if (me==6) call prog(5)
else if (me==7) call prog(4);
```

Figure 6: The simulation of a linear array code on a mesh.

```
me = mynode();
p = numnodes();
if ( me < p/2) call prog(me)
else call prog(3*p/2-1-me);
```

Figure 7: A generalized version for simulating a linear array code on a mesh of p processors.

3.2 Message routing for one-to-one sending

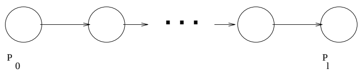


Figure 8: Routing a message from processor P_0 to P_l.

If an embedding is not perfect and two non-neighbor processors need to communicate, then a message routing is needed between these two processors. Figure 8 depicts a message to be sent from Processor P_0 to P_l with distance l. We discuss the types of message routing as follows.

1. **Store-Forward.** Each intermediate processor forwards the message to the next processor after it has received and stored the entire message.

2. **Wormhole (cut through) routing.** A channel is opened between the source and destination. The message is divided into small pieces (called flits). The flits are pipelined through the network channel. **Sending is fast** but the channel is used exclusively.

We define

- α is called the startup time, which is the time required to handle a message at the sending processor. That includes the cost of packaging this message (e.g. adding the header).
- t_h is the latency for a message to reach a neighbor processor. When a message is packed, it takes some time to deliver a message from the processor to the network and then to another processor. Notice that t_h is usually small compared to α in the current parallel computers.
- β is the data transmission speed between two processors. That is related to communication bandwidth (usually β=1/bandwidth). $\beta \times m$ will be the delay in delivering a message of the size m.

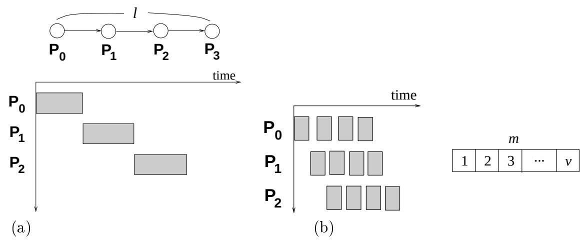


Figure 9: (a) An example of the store-forward routing model. (b) An example of wormhole model.

The store-forward model. Figure 9(a) depicts the store-forward model with $l = 3$, Each intermediate processor forwards the message to the next processor after it has received and stored the entire message. The total delay is

$$t_{comm} = \alpha + (t_h + m \times \beta) \times l.$$

For a neighbor communication, we simply model the cost as $t_{comm} \approx \alpha + \beta m$.

Wormhole/cut-through routing. In the wormhole communication model, A message of size m is divided into small v units and these units are pipelined through the communication channel. Figure 9(b) depicts the message pipelining between four processors ($l = 3$) with $v = 4$. The total communication delay is modeled as:

$$t_{comm} = \alpha \ + \ (\frac{m}{v}\beta + t_h)(l + v).$$

To get the shortest time delay, we derive the optimal package number v as:

$$\frac{dt}{dv} = 0, \quad -\frac{m\beta l}{v^2} + t_h = 0, \quad v = \sqrt{\frac{m\beta l}{t_h}}$$

Thus

$$\begin{aligned} t_{comm} &= \alpha \ + \ m\beta \ + \ \tfrac{m\beta l}{v} + t_h(l + v) \\ &= \alpha \ + \ m\beta \ + \ t_h(l + 2v) \\ &\approx \alpha \ + \ \beta m. \end{aligned}$$

The actual startup overhead depends on the software protocol design. On the lower level communication layer of the Meiko CS-2 machine, $\alpha \approx 10\mu s$, $\beta = 33$ MBytes per second and $t_h \approx 0.2\mu second$. In the NX/2 communication layer, $\alpha \approx 100\mu s$ and $\beta \approx 33MB/s$.

3.3 Basic Communication Operations

In most of scientific computing programs, program code and data are divided among processors and data exchanges between processors are often needed. Such exchanges require efficient communication schemes since communication delay affects the overall parallel performance. There are many communication patterns in a parallel program. We list some of them as follows:

- *One-one sending.* That is the simplest format of communication.
- *One-all broadcasting.* In this operation, one processor broadcasts the same message to different processors.
- *All-all broadcasting.* Every processor broadcasts a message to different processors. This operation is depicted in Figure10(a).

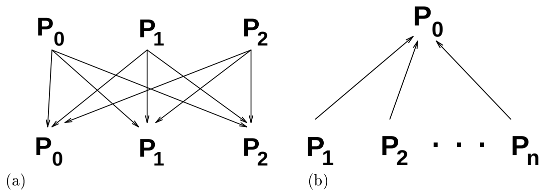


Figure 10: (a) An example of all-all broadcasting.(b) An example of accumulation (gather).

- *Accumulation (or called gather).* In this case, a processor receives a message from each of other processors and puts these messages together, as shown in Figure 10(b).
- *Reduction.* In this operation, each processor P_i holds one value V_i. Assume there are p processors. The global reduction computes the aggregate value of

$$V_0 \oplus V_1 \oplus \ldots \oplus V_p,$$

  and make it available to one processor (or to all processors). The operation $\oplus$ could stand for any reduction function, for example, the global sum.
- *One-all personalized communication or called single node scatter.* In this operation, one processor sends one personalized message to all of other processors. Different processors receive different messages.
- *All-all personalized communication.* In this operation, each processor sends one personalized message to all of other processors.

3.3.1 One-to-All Broadcast

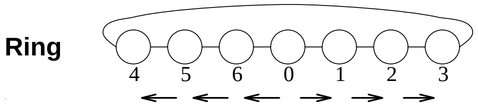


Figure 11: Broadcasting on a ring with the store-forward model.

We first discuss the implementation of broadcast on a ring using the store-forward routing model. Figure 11 depicts a broadcasting operation carried on a ring. The message is sent from processor 0 and let p be the number of processors, α be startup time, β - transmission speed and m be the size of the message. Then the total cost of this broadcasting is:

$$\frac{p}{2} \times (\alpha + \beta m)$$

If the architecture is a linear array instead of ring, then the worst-case cost is

$$p(\alpha + \beta m).$$

Figure 12 depicts a broadcasting operation carried on a mesh. Assume that the message is sent from the left and top processor. The communication is divided into two stages.

- At stage 1, the message is broadcasted in the first row. It costs $\sqrt{p}(\alpha + \beta m)$.
- At stage 2, the message is broadcasted in in all columns independently. The cost is: $\sqrt{p}(\alpha + \beta m)$.

The total cost is: $2\sqrt{p}(\alpha + \beta m)$.

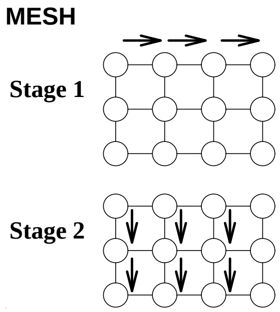


Figure 12: Broadcasting on a mesh with the store-forward model.

If the wormhole model is used, then the communication cost can be further reduced for this broadcasting case. Figure 13 shows a bisection method to broadcast in a linear array using the wormhole routing scheme. The bisection method is described as follows:

1. At the first, the message is directly sent to the center of the linear array. It costs $\alpha + \beta m$.
2. Then the message broadcasting is conducted independently in the left part and right part.
3. Repeat 1) and 2) recursively.

The bisection method takes about log p steps. The cost is $(\alpha + \beta m)\log p$. Notice there is no channel contention in wormhole routing since the broadcasting is conducted independently in subparts.

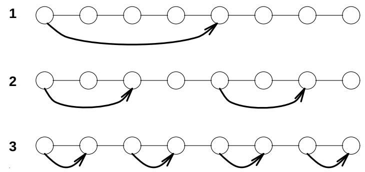


Figure 13: Broadcasting on a linear array with the wormhole model.

3.3.2 All-to-All Broadcast

We discuss briefly how an all-all broadcast algorithm can be designed on a ring with p processors.

- Each processor x forwards a message to its neighbor $(x+1) \bmod p$.
- After $p-1$ steps, all processors receive messages from all other processors. Thus the cost is $(p-1)(\alpha+\beta m)$.

Figure 14 depicts the method with $p=4$. At step 1, message 1 reaches its neighbor and at step 2, this message reach the next neighbor. This process is repeated for 3 steps.

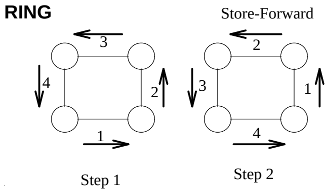


Figure 14: All-all broadcasting on a ring with the store-forward model.

3.3.3 One-to-all personalized broadcast

We discuss briefly on broadcasting a personalized message to each processor. Assume that the architecture is a linear array, the broadcast starts from the center of this linear array, as shown in Figure 15.

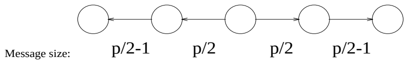


Figure 15: Personalized broadcasting on a linear with the store-forward model.

Then the center carries $p/2$ messages to the left part and $p/2$ messages to the right part. After the center sends these $p/2$ messages to its left neighbor, the total number of messages for this left neighbor to forward decreases by 1. Thus the cost of each step is listed as follows:

$$\begin{array}{llll} \textit{Step 1} & \alpha & + & \frac{p}{2}m\beta \\ \textit{Step 2} & \alpha & + & (\frac{p}{2}-1)m\beta \\ & & & \cdots \\ \textit{Step } p/2 & \alpha & + & 1m\beta \\ \texttt{Total cost} & & \approx & \frac{p}{2}\alpha + \frac{1}{2}(\frac{p}{2})^2 m\beta. \end{array}$$

4 Issues in Parallel Programming

We address the basic techniques in dependence analysis, program/data partitioning and mapping.

4.1 Dependence Analysis

Before executing a program in processors, we need to identify the inherent parallelism in this program. In this chapter, we discuss several graphical representations of program parallelism.

4.1.1 Basic dependence

We need to introduce the concept of dependence. We call a basic computation unit as a task. A task is a program fragment, which could be a basic assignment, a procedure or a logical test. A program consists of a sequence of tasks. When tasks are executed in parallel on different processors, the relative execution order of those tasks is different from the one in the sequential execution. It is mandatory to study orders between tasks that must be followed so that the semantic of this program does not change during parallel execution.

For example:

$S_1 : x = a + b$
$S_2 : y = x + c$

For this example, each statement is considered as a task. Assume statements are executed in separate processors. S_2 needs to use the value of x defined by S_1. If two processors share one memory, S_1 has to be executed first in one processor. The result of x is updated in the global memory. Then S_2 can fetch x from the memory and start its execution. In a distributed environment, after the execution of S_1, data x needs to be sent to the processor where S_2 is executed. This example demonstrates the importance of dependence relations between tasks. We formally define basic types of data dependence between tasks.

Definition: Let $IN(T)$ be the set of data items used by task T and $OUT(T)$ be the set of data items modified by task T.

- $OUT(T_1) \bigcap IN(T_2) \neq \emptyset$
  T_2 is *data flow-dependent (or called true-dependent)* on T_1. Example: $\boxed{\begin{array}{l} S_1 : A = x + B \\ S_2 : C = A * 3 \end{array}}$
- $OUT(T_1) \bigcap OUT(T_2) \neq \emptyset$
  T_2 is *output-dependent* on T_1. $\boxed{\begin{array}{l} S_1 : A = x + B \\ S_2 : A = 3 \end{array}}$
- $IN(T_1) \bigcap OUT(T_2) \neq \emptyset$
  T_2 is *anti-dependent* on T_1. $\boxed{\begin{array}{l} S_1 : B = A + 3 \\ S_2 : A = 3 \end{array}}$

Coarse-grain dependence graph. Tasks operate on a set of data items of large sizes and perform a large chunk of computations. An example of such a dependence graph is shown in Figure 16.

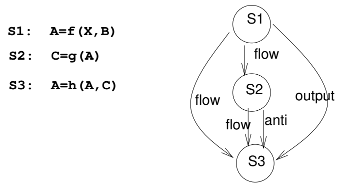


Figure 16: An example of a dependence graph. Functions f, g, h do not modify their input arguments.

4.1.2 Loop Parallelism

Loop parallelism can be modeled by the *iteration space* of a loop program which contains all iterations of a loop and data dependence between iteration statements. An example is in Fig. 17.

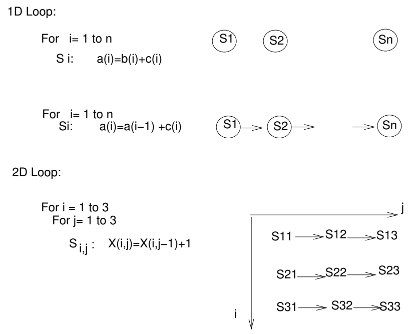


Figure 17: An example of a loop dependence graph (loop iteration space).

4.2 Program Partitioning

Purpose:

- Increase task grain size.
- Reduce unnecessary communication.
- Simplify the mapping of a large number of tasks to a small number of processors.

Two techniques are considered: loop blocking/unrolling and interior loop blocking. Loop interchange technique that assists partitioning will also be discussed.

4.2.1 Loop blocking/unrolling

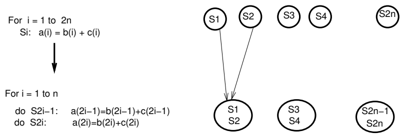


Figure 18: An example of 1D loop blocking/unrolling.

An example is Fig. 18. In general, sequential code:

```
For i = 1 to r*p
    Si : a(i) = b(i)+c(i)
```


Blocking this loop by a factor of r:

```
For j = 0 to p-1
    For i = r*j+1 to r*j+r
        a(i) = b(i)+c(i)
```

Assume there are p processors. A SPMD code for the above partitioning can be:

```
me=mynode();
For i = r*me+1 to r*me+r
    a(i) = b(i)+c(i)
```

4.2.2 Interior loop blocking

This technique is to block an interior loop and make it one task.

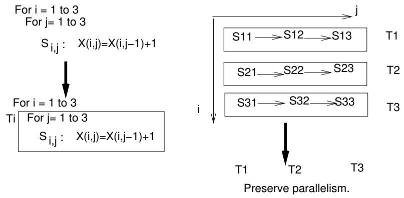


Figure 19: An example of interior loop blocking.

Grouping statements together may reduce available parallelism. We should try to preserve parallelism as much as possible in partitioning a loop program. One such an example is in Fig. 19. Fig. 20 shows an example of interior loop blocking that does not preserve parallelism. Loop interchange can be used before partitioning in assisting the exploitation of parallelism.

4.2.3 Loop interchange

Loop interchange is a program transformation that changes the execution order of a loop program as shown in Fig. 21. Loop interchange is not legal if the new execution order violates data dependence. An example of illegal interchange is in Fig. 22.

An example of interchanging triangular loops is shown below:

```
For i = 1 to 10                  For j = 2 to 10
    For j = i+1 to 10     --->       For i = 1 to j-1
        X(i,j)=X(i,j-1)+1                X(i,j)=X(i,j-1)+1
```

4.3 Data Partitioning

For distributed memory architectures, data partitioning is needed when there is no enough space for replication. Data structure is divided into *data units* and assigned to the local memories of the processors. A data unit can be a scalar variable, a vector or a submatrix block.

4.3.1 Data partitioning methods

1D array $\longrightarrow$ 1D processors.

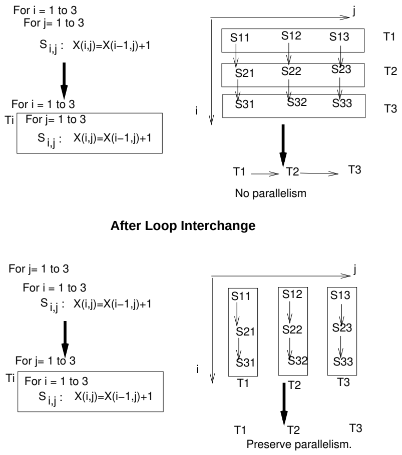


Figure 20: An example of partitioning that reduces parallelism and the role of loop interchange.

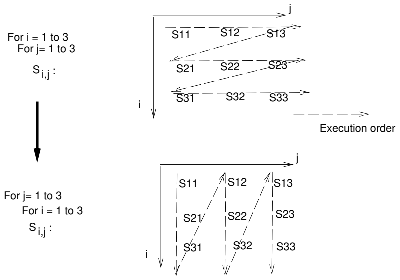


Figure 21: Loop interchange re-orders execution.

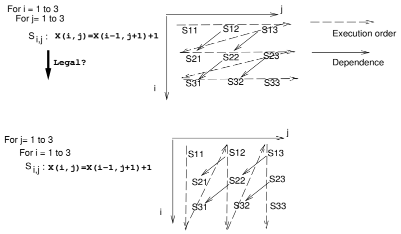
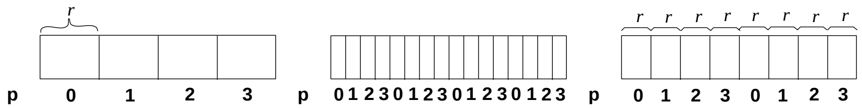


Figure 22: An illegal loop interchange.

Assume that data items are counted from $0, 1, \cdots n-1$, and processors are numbered from 0 to $p-1$. Let $r = \lceil \frac{n}{p} \rceil$. Three common methods for a 1D array are depicted in Fig. 23.



(a) Block. (b) Cyclic. (c) Block cyclic.

Figure 23: Data partitioning schemes.

- *1D block.* Data i is mapped to processor $\lfloor \frac{i}{r} \rfloor$.
- *1D cyclic.* Data i is mapped to processor $i \bmod p$.
- *1D block cyclic.* First the array is divided into a set of units using block partitioning (block size b). Then these units are mapped in a cyclic manner to p processors. Data i is mapped to processor $\lfloor \frac{i}{b} \rfloor \bmod p$.

2D array $\longrightarrow$ 1D processors.

Data elements are counted as (i, j) where $0 \le i, j \le \cdots n-1$. Processors are numbered from 0 to $p-1$. Let $r = \lceil \frac{n}{p} \rceil$.

- *Row-wise block.* Data (i, j) is mapped to processor $\lfloor \frac{i}{r} \rfloor$.
- *Column-wise block.* Data (i, j) is mapped to processor $\lfloor \frac{j}{r} \rfloor$.

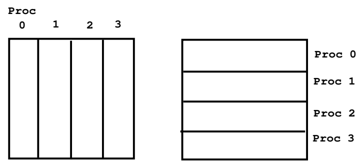


Figure 24: The left is the column-wise block and the right is the row-wise block.

- *Row-wise cyclic* Data (i, j) is mapped to processor i mod p.
- *Other partitioning.* Column-wise cyclic, Column-wise block cyclic. Row-wise block cyclic.

2D array $\longrightarrow$ 2D processors.

Data elements are counted as (i, j) where $0 \le i, j \le \cdots n-1$. Processors are numbered as (s, t) where $0 \le s, t \le \cdots q-1$ where $q = \sqrt{p}$. Let $r = \lceil \frac{n}{q} \rceil$.

- *(Block,Block).* Data (i, j) is mapped to processor $(\lfloor \frac{i}{r} \rfloor, \lfloor \frac{j}{r} \rfloor)$
- *(Cyclic,Cyclic).* Data (i, j) is mapped to processor $(i \bmod q,\ j \bmod q)$.

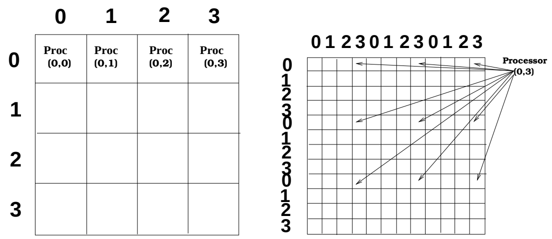


Figure 25: The left is the 2D block mapping (block,block) and the right is the 2D cyclic mapping (cyclic,cyclic).

- *Other partitioning.* (Block, Cyclic), (Cyclic, Block), (Block cyclic, Block cyclic).

4.3.2 Consistency between program and data partitioning

Given a computation partitioning and processor mapping, there are several choices available for data partitioning and mapping. How can we make a good choice of data partitioning and mapping? For distributed memory architectures large grain data partitioning is preferred because there is a high communication startup overhead in transferring a small size data unit. If a task requires to access a large number of distinct data units and data units are evenly distributed among processors, then there will be substantial communication overhead in fetching a large number of non-local data items for executing this task. Thus the following rule of thumb can be used to guide the design of program and data partitioning:

> **Consistency.** The program partitioning and data partitioning are consistent if sufficient parallelism is provided by partitioning and at the same time the number of distinct units accessed by each task is minimized.

The following rule is a simple heuristic used in determining data and program mapping.

> **"Owner computes rule"**: If a computation statement x modifies a data item i, then the processor that owns data item i executes statement x.

For example, sequential code:

```
For i = 0 to r*p-1
      S_i : a(i) = 3.
```


Blocking this loop by a factor of r

```
For j = 0 to p-1
    For i = r*j to r*j+r-1
        a(i) = 3.
```

Assume there are p processors. SPMD code is:

```
me=mynode();
For i = r*me to r*me+r-1
      a(i) = 3.
```

Data array $a(i)$ are distributed to processors such that if processor x executes $a(i) = 3$, then $a(i)$ is assigned to processor x. For example, we let processor 0 own data $a(0), a(1), \cdots, a(r-1)$. Otherwise if processor 0 does not have $a(0)$, this processor needs to allocate some temporal space to perform $a(0) = 3$ and send the result back the processor that owns $a(0)$, which leads to a certain amount of communication overhead.

The above SPMD code is for block mapping. For cyclic mapping, the code is:

```
me=mynode();
For i = me to r*p-1 step-size
p
      a(i) = 3.
```

A more general SPMD code structure for an arbitrary processor mapping method *proc_map*(i) (but with more code overhead) is

```
me=mynode();
For i =0 to rp-1
      if ( proc_map(i) == me) a(i) = 3.
```

For block mapping, *proc_map*$(i) = \lfloor \frac{i}{r} \rfloor$.

For cyclic mapping, *proc_map*$(i) = i \bmod p$.

4.3.3 Data indexing between global space and local space

There is one more problem with the previous program: statement "a(i)=3" uses "i" as the index function and the value of i is in a range from 0 to $r*p-1$. When a processor allocates r units, there is a need to translate the global index i to a local index which accesses the local memory at that processor. This is depicted in Fig.26. The correct code structure is shown below.

```
int a[r];
me=mynode();
For i =0 to rp-1
      if ( proc_map(i) == me) a(local(i)) = 3.
```

For block mapping, $local(i) = i \bmod r$. For cyclic mapping, $local(i) = \lfloor \frac{i}{p} \rfloor$.

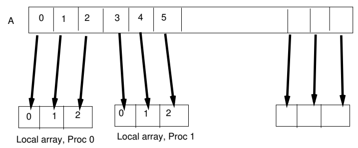


Figure 26: Global data vs local data index.

In summary, given data item i (i starts from 0).

- **1D Block.**
  Processor ID: $proc\_map(i) = \lfloor \frac{i}{r} \rfloor$.

  Local data address: $Local(i) = i \bmod r$.

  An example of mapping with p=2 and r=3 is:

Proc 0	Proc 1
$0 \to 0$	$3 \to 0$
$1 \to 1$	$4 \to 1$
$2 \to 2$	$5 \to 2$

- **1D Cyclic.**
  Processor ID: $proc\_map(i) = i \bmod p$.

  Local data address: $Local(i) = \lfloor \frac{i}{p} \rfloor$.

  An example of cyclic mapping with p=2 is:

proc 0	proc 1
$0 \to 0$	$1 \to 0$
$2 \to 1$	$3 \to 1$
$4 \to 2$	$5 \to 2$
$6 \to 3$	

4.4 A summary on program parallelization

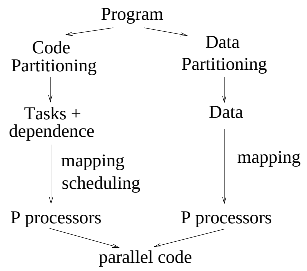


Figure 27: The process of program parallelization.

The process of program parallelization is depicted in Figure 27 and we will demonstrate this process in parallelizing following scientific computing algorithms.

- Matrix-vector multiplication.
- Matrix-matrix multiplication.
- Direct methods for solving a linear equation. e.g. Gaussian Elimination.
- Iterative methods for solving a linear equation. e.g. Jacobi.
- Finite-difference methods for differential equations.

5 Matrix Vector Multiplication

Problem: $y = A * x$ where A is a $n \times n$ matrix and x is a column vector of dimension n.

Sequential code:

for $i = 1$ **to** n **do**
  $y_i = 0$;
  for $j = 1$ to n **do**
    $y_i = y_i + a_{i,j} * x_j$;
  Endfor
Endfor

An example:

$$\begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix} * \begin{pmatrix} 1 \\ 2 \\ 3 \end{pmatrix} = \begin{pmatrix} 1*1+2*2+3*3 \\ 4*1+5*2+6*3 \\ 7*1+8*2+9*3 \end{pmatrix} = \begin{pmatrix} 14 \\ 32 \\ 50 \end{pmatrix}$$

The sequential complexity is $2n^2\omega$ where ω is the time for an addition or multiplication.

Partitioned code:

for $i = 1$ **to** n **do**
  S_i : $y_i = 0$;
    for $j = 1$ to n **do**
      $y_i = y_i + a_{i,j} * x_j$;
    Endfor
Endfor

Dependence task graph is shown in Fig. 28.

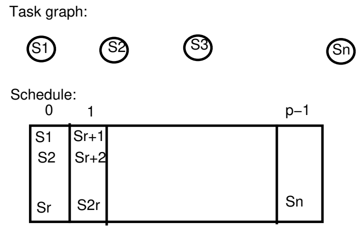


Figure 28: Task graph and a schedule for matrix vector multiplication.

Task schedule on p processors is shown in Fig. 28.

Mapping function of tasks S_i. For the above schedule: $proc\_map(i) = \lfloor \frac{i-1}{r} \rfloor$ where $r = \lceil \frac{n}{p} \rceil$.

Data partitioning based on the above schedule: matrix A is divided into n rows $A_1, A_2, \cdots A_n$.

Data mapping: Row A_i is mapped to processor $proc\_map(i)$, the same as task i. The indexing function is: $local(i) = (i-1) \bmod r$. Vectors x and y are replicated to all processors.

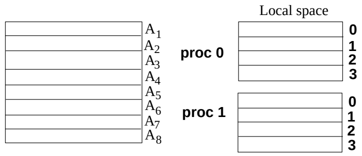


Figure 29: An illustration of data mapping for matrix-vector multiplication.

SPMD parallel code:

```
int x[n], y[n], a[r][n];
me=mynode();
for i = 1 to n do
    if proc_map(i) = me, then do S_i:
        S_i :   y[i] = 0;
                for j = 1 to n do
                    y[i] = y[i] + a[local(i)][j] * x[j];
                Endfor
Endfor
```

The parallel time is $PT = \frac{n}{p} \times 2n\omega$ since each task S_i costs $2n\omega$, ignoring the overhead of computing $local(i)$. Thus $PT = \frac{2n^2\omega}{p}$.

6 Matrix-Matrix Multiplication

6.1 Sequential algorithm

Problem: $C = A * B$ where A and B are $n \times n$ matrices.

Sequential code:

```
for i = 1 to n do
    for j = 1 to n do
        sum = 0;
        for k = 1 to n do
            sum = sum + a(i,k) * b(k,j);
        Endfor
        c(i,j) = sum;
    Endfor
Endfor
```

An example:

$$\begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix} * \begin{pmatrix} 5 & 7 \\ 6 & 8 \end{pmatrix} = \begin{pmatrix} 1*5+2*6 & 1*7+2*8 \\ 3*5+4*6 & 3*7+4*8 \end{pmatrix}$$

Time Complexity. Each multiplication or addition counts one time unit ω.

$$\text{No of operations} = \sum_{i=1}^{n}\sum_{j=1}^{n}\sum_{k=1}^{n} 2\omega = 2n^3\omega$$

6.2 Parallel algorithm with sufficient memory

Partitioned code:

```
for i = 1 to n do
  T_i :  for j = 1 to n do
             sum = 0;
             for k = 1 to n do
                 sum = sum + a(i,k) * b(k,j);
             Endfor
             c(i,j) = sum;
         Endfor
Endfor
```

Since tasks T_i $(1 \leq i \leq n)$ are independent, we use the following mapping for the parallel code:

- Matrix A is partitioned using row-wise block mapping
- Matrix C is partitioned using row-wise block mapping
- Matrix B is duplicated to all processors
- Task T_i is mapped to the processor of row i in matrix A.

The SPMD code with parallel time $2n^3\omega/p$ is:

```
For i = 1 to n
    if proc_map(i)==me do T_i;
Endfor
```

A more detailed description of the algorithm is:

```
for i = 1 to n do
  if proc_map(i)==me do
      for j = 1 to n do
          sum = 0;
          for k = 1 to n do
              sum = sum + a(local(i),k) * b(k,j);
          Endfor
          c(local(i),j) = sum;
      Endfor
  endif
Endfor
```

6.3 Parallel algorithm with 1D partitioning

The above algorithm assumes that B can be replicated to all processors. In practice, that costs too much memory. We describe an algorithm in which all of matrices A, B, and C are uniformly distributed among processors.

Partitioned code:

```
for i = 1 to n do
  for j = 1 to n do
    T_i,j :  sum = 0;
             for k = 1 to n do
                 sum = sum + a(i,k) * b(k,j);
             Endfor
             c(i,j) = sum;
  Endfor
Endfor
```

Data access: Each task $T_{i,j}$ reads row A_i and column B_j to write data element $c_{i,j}$.

Task graph: There are n^2 independent tasks:
$$\begin{array}{llll} T_{1,1} & T_{1,2} & \cdots T_{1,n} \\ T_{2,1} & T_{2,2} & \cdots T_{2,n} \\ \cdots & & \\ T_{n,1} & T_{n,2} & \cdots T_{n,n} \end{array}$$

Mapping.

- Matrix A is partitioned using row-wise block mapping
- Matrix C is partitioned using row-wise block mapping
- Matrix B is partitioned using column-wise block mapping
- Task $T_{i,j}$ is mapped to the processor of row i in matrix A. Cluster 1: $\boxed{T_{1,1}\ T_{1,2}\ \cdots T_{1,n}}$
  Cluster 2: $\boxed{T_{2,1}\ T_{2,2}\ \cdots T_{2,n}}$
  $\cdots$
  Cluster n: $\boxed{T_{n,1}\ T_{n,2}\ \cdots T_{n,n}}$

Parallel algorithm:

```
For j = 1 to n
    Broadcast column B_i to all processors
    Do tasks T_{1,j}, T_{2,j}, ..., T_{n,j} in parallel.
Endfor
```

Parallel time analysis. Each multiplication or addition counts one time unit ω. Each task $T_{i,j}$ costs $2n\omega$. Also we assume that each broadcast costs $(\alpha + \beta n)\log p$.

$$PT = \sum_{j=1}^{n}((\alpha + \beta n)\log p + \frac{n}{p}2n\omega) = n(\alpha + \beta n)\log p + \frac{2n^3\omega}{p}.$$

6.4 Fox's algorithm for 2D data partitioning

Naturally another option we have for program partitioning is to treat each interior statement as one task. Let us first ignore the initialization statement. We have the following partitioning.

```
for i = 1 to n do
  for j = 1 to n do
    for k = 1 to n do
      T_{i,j,k} : [ c_{i,j} = c_{i,j} + a_{i,k} * b_{k,j}; ]
    Endfor
  Endfor
Endfor
```

Then the iteration space is 3D. All tasks $T_{i,j,k}$ $(1 \le k \le n)$ that modify the same $c_{i,j}$ will have a chain dependence, which corresponds to a summation of n terms for $c_{i,j}$. There are no dependence between those chains.

The Fox's algorithm is used to support 2D block partitioning of matrices A, B and C. This algorithm requires a re-organization of the sequential code. There are two basic ideas: reordering the sequential addition and submatrix-based partitioning.

6.4.1 Reorder the sequential additions.

We examine the summation sequence for each result element $c_{i,j}$,

$$c_{i,j} = a_{i,1} * b_{1,j} + a_{j,2} * b_{2,j} + \cdots + a_{i,n} * b_{n,j}.$$

It can be re-ordered as:

$$c_{i,j} = a_{i,i} * b_{i,j} + a_{i,i+1} * b_{i+1,j} + \cdots + a_{i,n} * b_{n,j} + a_{i,1} * b_{1,j} + a_{j,2} * b_{2,j} + \cdots + a_{i,i-1} * b_{i-1,j}.$$

i.e.

$$c_{i,j} = \sum_{k=1}^{n} a_{i,(i+k-2) \bmod n+1} * b_{(i+k-2) \bmod n+1,j}.$$

Assume that we use a 2D data mapping on $n \times n$ processors. Processor (i,j) owns data $a_{i,j}$, $b_{i,j}$ and $c_{i,j}$. We will execute the computation for $c_{i,j}$ in n stages. At stage k $(k = 1, \cdots, n)$, for all $1 \le i,j \le n$, do

$$c_{i,j} = c_{i,j} + a_{i,(i+k-2) \bmod n+1} * b_{(i+k-2) \bmod n+1,j}$$

We use an example for 3×3 processors to demonstrate the computation sequence.

$$\begin{bmatrix} c_{11} = a_{11} * b_{11} & c_{12} = a_{11} * b_{12} & c_{13} = a_{11} * b_{13} \\ c_{21} = a_{22} * b_{21} & c_{22} = a_{22} * b_{22} & c_{23} = a_{22} * b_{23} \\ c_{31} = a_{33} * b_{31} & c_{32} = a_{33} * b_{32} & c_{33} = a_{33} * b_{33} \end{bmatrix}^{k=1} \Longrightarrow \begin{bmatrix} c_{11} += a_{12} * b_{21} & c_{12} += a_{12} * b_{22} & c_{13} += a_{12} * b_{23} \\ c_{21} += a_{23} * b_{31} & c_{22} += a_{23} * b_{32} & c_{23} += a_{23} * b_{33} \\ c_{31} += a_{31} * b_{11} & c_{32} += a_{31} * b_{12} & c_{33} += a_{31} * b_{13} \end{bmatrix}^{k=2}$$

$$\Longrightarrow \begin{bmatrix} c_{11} += a_{13} * b_{31} & c_{12} += a_{13} * b_{32} & c_{13} += a_{13} * b_{33} \\ c_{21} += a_{21} * b_{11} & c_{22} += a_{21} * b_{12} & c_{23} += a_{21} * b_{13} \\ c_{31} += a_{32} * b_{21} & c_{32} += a_{32} * b_{22} & c_{33} += a_{32} * b_{23} \end{bmatrix}^{k=3}$$

Then we examine what elements of a are used during the above 3 stages so we can determine the communication patterns.

$$\begin{bmatrix} a_{11} & a_{11} & a_{11} \\ a_{22} & a_{22} & a_{22} \\ a_{33} & a_{33} & a_{33} \end{bmatrix}^{k=1} \Longrightarrow \begin{bmatrix} a_{12} & a_{12} & a_{12} \\ a_{23} & a_{23} & a_{23} \\ a_{31} & a_{31} & a_{31} \end{bmatrix}^{k=2} \Longrightarrow \begin{bmatrix} a_{13} & a_{13} & a_{13} \\ a_{21} & a_{21} & a_{21} \\ a_{32} & a_{32} & a_{32} \end{bmatrix}^{k=3}$$

Thus the communication pattern is that at each stage, $a_{i,(i+k-2) \bmod n+1}$ is broadcasted to every processor in row i. We also examine what elements of b are used for each grid point at each stage.

$$\begin{bmatrix} b_{11} & b_{12} & b_{13} \\ b_{21} & b_{22} & b_{23} \\ b_{31} & b_{32} & b_{33} \end{bmatrix}^{k=1} \Longrightarrow \begin{bmatrix} b_{21} & b_{22} & b_{23} \\ b_{31} & b_{32} & b_{33} \\ b_{11} & b_{12} & b_{13} \end{bmatrix}^{k=2} \Longrightarrow \begin{bmatrix} b_{31} & b_{32} & b_{33} \\ b_{11} & b_{12} & b_{13} \\ b_{21} & b_{22} & b_{23} \end{bmatrix}^{k=3}$$

For b, studying the element movement of column 1 from Stage 1, 2 to 3, we can observe that essentially a ring pipelining for all-all broadcast is conducted in every column direction.

Parallel algorithm:

```
For k = 1 to n
    On each processor (i, j), set t = (k + i − 2) mod n + 1.
    At each row, a_{i,t} at processor (i, t) is broadcasted to other processors in the same row (i, j) where 1 ≤ j ≤ n.
    Do c_{i,j} = c_{i,j} + a_{i,t} * b_{t,j} in parallel for all processors.
    Every processor (i,j) sends b_{i,t} to processor (i − 1, j).
Endfor
```

6.4.2 Submatrix partitioning

In practice, the number of processors is much less than n^2. Also it does not make sense to utilize n^2 processors since the code suffers too much communication overhead for fine-grain computation. To increase the granularity of computation, we map the $n \times n$ grid to processors using the 2D block method.

First we partition all matrices $(A, B, C$ of size $n \times n)$ in a submatrix style. Each processor (i,j) is assigned a submatrix of size $n/q \times n/q$ where $q = \sqrt{p}$. Let $r = n/q$. The submatrix partitioning of A can be demonstrated using the following example with $n = 4$ and $q = 2$.

$$A = \begin{pmatrix} a_{11} & a_{12} & a_{13} & a_{1n} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \\ a_{41} & a_{42} & a_{43} & a_{44} \end{pmatrix} \Longrightarrow \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix}$$

where

$$A_{11} = \begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix}, A_{12} = \begin{pmatrix} a_{13} & a_{14} \\ a_{23} & a_{24} \end{pmatrix}, A_{21} = \begin{pmatrix} a_{31} & a_{32} \\ a_{41} & a_{42} \end{pmatrix}, A_{22} = \begin{pmatrix} a_{33} & a_{34} \\ a_{43} & a_{44} \end{pmatrix}.$$

Then the sequential algorithm can be re-organized as

```
for i = 1 to q do
  for j = 1 to q do
    C_{i,j} = 0;
    for k = 1 to q do
      C_{i,j} = C_{i,j} + A_{i,k} * B_{k,j};
    Endfor
  Endfor
Endfor
```

Then we use the same idea of re-ordering:

$$C_{i,j} = A_{i,i} * B_{i,j} + A_{i,i+1} * B_{i+1,j} + \cdots + A_{i,n} * B_{n,j} + A_{i,1} * B_{1,j} + A_{j,2} * B_{2,j} + \cdots + A_{i,i-1} * B_{i-1,j}.$$

Parallel algorithm:

```
For k = 1 to q
    On each processor (i, j), t = (k + i - 2) mod q + 1.
    At each row, A_{i,t} at processor (i, t) is broadcasted to other processors in the same row (i, j) where 1 <= j <= q.
    Do C_{i,j} = C_{i,j} + A_{i,t} * B_{t,j} in parallel for all processors.
    Every processor (i,j) sends B_{i,t} to processor (i - 1, j).
Endfor
```

Parallel time analysis. A submatrix multiplication $(A_{i,t} * B_{t,j})$ costs $2r^3\omega$. A submatrix addition costs $r^2\omega$. Each inner most statement costs $2r^3\omega$. Also we assume that each broadcast of a submatrix costs $(\alpha + \beta r^2)\log q$.

$$PT = \sum_{k=1}^{q}((\alpha + \beta r^2)(1 + \log q) + 2r^3\omega) = q(\alpha + \beta(n/q)^2)(1 + \log q) + 2(n/q)^3\omega = (\sqrt{p}\alpha + \beta\frac{n^2}{\sqrt{p}})(1 + \log\sqrt{p}) + \frac{2n^3\omega}{p}.$$

This algorithm has communication overhead much smaller than the 1D algorithm presented in the previous subsection.

7 Gaussian Elimination for Solving Linear Systems

7.1 Gaussian Elimination without Partial Pivoting

7.1.1 The Row-Oriented GE sequential algorithm

The Gaussian Elimination method for solving linear system $Ax = b$ is listed below. Assume that column $n + 1$ of A stores column b. Loop k controls the elimination steps. Loop i controls i-th row accessing and loop j controls j-th column accessing.

Forward Elimination:
```
For k = 1 to n - 1
  For i = k + 1 to n
    a_{i,k} = a_{i,k}/a_{k,k};
    For j = k + 1 to n + 1
      a_{i,j} = a_{i,j} - a_{i,k} * a_{k,j};
    Endfor
  Endfor
Endfor
```

Notice that since the lower triangle matrix elements become zero after elimination, their space is used to store multipliers ($a_{i,k} = a_{i,k}/a_{k,k}$).

Backward Substitution: Note that x_i uses the space of $a_{i,n+1}$.
```
For i = n to 1
  For j = i + 1 to n
    x_i = x_i - a_{i,j} * x_j;
  Endfor
  x_i = x_i/a_{i,i};
Endfor
```

An example: Given the following matrix system:

$$\begin{array}{lrcl} (1) & 4x_1 - 9x_2 + 2x_3 & = & 2 \\ (2) & 2x_1 - 4x_2 + 4x_3 & = & 3 \\ (3) & -x_1 + 2x_2 + 2x_3 & = & 1 \end{array}$$

We eliminate the coefficients of x_1 for equations (2) and (3) and make them zero.

$$\begin{array}{lrcll} (2)\text{-}(1)*\frac{2}{4}: & 0.5x_2 + 3x_3 & = & 2 & (4) \\ (3)\text{-}(1)*\text{-}\frac{1}{4}: & -\frac{1}{4}x_2 + \frac{5}{2}x_3 & = & \frac{3}{2} & (5) \end{array}$$

Then we eliminate the coefficients of x_2 for equation (5).

$$(5)\text{-}(4)*\text{-}\tfrac{1}{2}: \quad 4x_3 = \tfrac{5}{2}$$

Now we have an upper triangular system:
$$\begin{array}{rcl} 4x_1 - 9x_2 + 2x_3 & = & 2 \\ \frac{1}{2}x_2 + 3x_3 & = & 2 \\ 4x_3 & = & \frac{5}{2} \end{array}$$

Given this upper triangular system, backward substitution performs the following operations:

$$\begin{array}{rcl} x_3 & = & \frac{5}{8} \\ x_2 & = & \frac{2-3x_3}{\frac{1}{2}} = \frac{1}{4} \\ x_1 & = & \frac{2+9x_2-2x_3}{4} = \frac{3}{4} \end{array}$$

The forward elimination process can be expressed using an augmented matrix $(A \mid b)$ where b is treated as the n+1 column of A.

$$\begin{pmatrix} 4 & -9 & 2 & 2 \\ 2 & -4 & 4 & 3 \\ -1 & 2 & 2 & 1 \end{pmatrix} \xRightarrow[]{\substack{(2)=(2)-(1)*\frac{2}{4} \\ (3)=(3)-(1)*\frac{-1}{4}}} \begin{pmatrix} 4 & -9 & 2 & 2 \\ 0 & \frac{1}{2} & 3 & 2 \\ 0 & \frac{-1}{4} & \frac{5}{2} & \frac{3}{2} \end{pmatrix} \longrightarrow \begin{pmatrix} 4 & -9 & 2 & 2 \\ 0 & 1/2 & 3 & 2 \\ 0 & 0 & 4 & 5/2 \end{pmatrix}$$

Time complexity: Each of division, multiplication and subtraction counts one time unit ω.

#Operations in forward elimination:

$$\sum_{k=1}^{n-1}\sum_{i=k+1}^{n}\left(1 + \sum_{j=k+1}^{n+1} 2\right)\omega = \sum_{k=1}^{n-1}\sum_{i=k+1}^{n}(2(n-k)+3)\omega \approx 2\omega\sum_{k=1}^{n-1}(n-k)^2 \approx \frac{2n^3}{3}\omega$$

#Operations in backward substitution:

$$\sum_{i=1}^{n}(1+\sum_{j=i+1}^{n}2)\omega \approx 2\omega\sum_{i=1}^{n}(n-i) \approx n^2\omega$$

Thus the total number of operations is about $\frac{2n^3}{3}\omega$. The total space needed is about n^2 double-precision numbers.

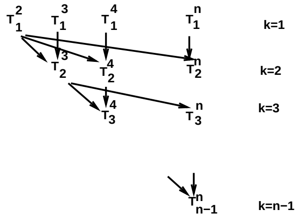
7.1.2 The row-oriented parallel algorithm

A program partitioning for the forward elimination part:

For $k = 1$ **to** $n-1$
  For $i = k+1$ **to** n
    T_k^i : $a_{ik} = a_{ik}/a_{kk}$
      For $j = k+1$ **to** $n+1$
        $a_{ij} = a_{ij} - a_{ik} * a_{kj}$
      EndFor

The **data access pattern** of each task is: T_k^i : Read rows A_k, A_i
Write row A_i.

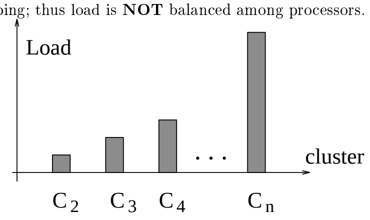
Dependence Graph.



Thus for each step k, tasks $T_k^{k+1}\ T_k^{k+2}\ \ldots\ T_k^n$ are independent. We can have the following algorithm design for the row-oriented GE algorithm:

For $k = 1$ **to** $n-1$
  Do $T_k^{k+1}\ T_k^{k+2}\ \ldots\ T_k^n$ in parallel on p processors.

Task and data mapping. We first group tasks into a set of clusters using "owner computes rule". Each task T_k^i that modifies the same row i is in the same cluster C_i. Row i and cluster C_i will be mapped to the same processor. We discuss how rows should be mapped to processors. If block mapping is used, we profile the computation load of clusters $C_2, C_3, \ldots, C_n$ below. Processor 0 will get the smallest amount of load and the last processor gets the most of computation load for block mapping; thus load is **NOT** balanced among processors. Cyclic mapping should be used.



Parallel Algorithm:

Proc 0 broadcasts Row 1
For $k = 1$ **to** $n-1$
  Do $T_k^{k+1} \ldots T_k^n$ in parallel.
  Broadcast row $k+1$.
Endfor

SPMD Code:

me=mynode();
For $i = 1$ **to** n
  if proc_map(i)==me, initialize Row i;
Endfor
If proc_map(1)==me, broadcast Row 1 **else** receive it;
For $k = 1$ **to** $n-1$
  For $i = k+1$ **to** n
    If proc_map(i)==me, do T_k^i.
  EndFor
  If proc_map(k+1)==me, then broadcast Row $k+1$ **else** receive it.
EndFor

7.1.3 The column-oriented algorithm

The column-oriented algorithm essentially interchanges loops i and j of the GE program in Section 7.1.1.

Forward elimination part:

For $k = 1$ **to** $n-1$
  For $i = k+1$ **to** n
    $a_{i,k} = a_{i,k}/a_{k,k}$
  EndFor
  For $j = k+1$ **to** $n+1$
    For $i = k+1$ **to** n
      $a_{i,j} = a_{i,j} - a_{i,k} * a_{k,j}$
    EndFor
  EndFor
EndFor

Given the first step of forward elimination in the previous example,

$$\begin{pmatrix} 4 & -9 & 2 & 2 \\ 2 & -4 & 4 & 3 \\ -1 & 2 & 2 & 1 \end{pmatrix} \xrightarrow[(3)=(3)-(1)*\frac{-1}{4}]{(2)=(2)-(1)*\frac{2}{4}} \begin{pmatrix} 4 & -9 & 2 & 2 \\ 0 & \frac{1}{2} & 3 & 2 \\ 0 & \frac{-1}{4} & \frac{5}{2} & \frac{3}{2} \end{pmatrix}$$

One can mark the data access (writing) sequence for row-oriented elimination below. Notice that $\boxed{1}$ computes and stores the multiplier $\frac{2}{4}$ and $\boxed{5}$ stores $\frac{-1}{4}$.

$$\begin{pmatrix} \boxed{1} & \boxed{2} & \boxed{3} & \boxed{4} \\ \boxed{5} & \boxed{6} & \boxed{7} & \boxed{8} \end{pmatrix}$$

Then the data writing sequence for column-oriented elimination is:

$$\begin{pmatrix} \boxed{1} & \boxed{3} & \boxed{5} & \boxed{7} \\ \boxed{2} & \boxed{4} & \boxed{6} & \boxed{8} \end{pmatrix}$$

Notice that $\boxed{1}$ computes and stores the multiplier $\frac{2}{4}$ and $\boxed{2}$ stores $\frac{-1}{4}$.

Column-oriented backward substitution. We change the loops of i and j in the row-oriented backward substitution code. Notice again that column $n+1$ stores the solution vector x.

For $j = n$ **to** 1
  $x_j = x_j / a_{j,j}$;
  For $i = j-1$ **to** 1
   $x_i = x_i - a_{i,j} x_j$;
  Endfor
EndFor

For example, given a upper triangular system:
$$\begin{aligned} 4x_1 - 9x_2 + 2x_3 &= 2 \\ 0.5x_2 + 3x_3 &= 2 \\ 4x_3 &= \tfrac{5}{2}. \end{aligned}$$

The row-oriented algorithm performs:
$$\begin{aligned} &x_3 = \tfrac{5}{8} \\ &x_2 = 2 - 3x_3 \\ &x_2 = \tfrac{x_2}{0.5} \\ &x_1 = 2 + 9x_2 \\ &x_1 = x_1 - 2x_3 \\ &x_1 = \tfrac{x_1}{4}. \end{aligned}$$

The column-oriented algorithm performs:
$$\begin{aligned} &x_3 = \tfrac{5}{8} \\ &x_2 = 2 - 3x_3 \\ &x_1 = 2 - 2x_3 \\ &x_2 = \tfrac{x_2}{0.5} \\ &x_1 = x_1 + 9x_2 \\ &x_1 = \tfrac{x_1}{4}. \end{aligned}$$

Partitioned forward elimination: **For** $k = 1$ **to** $n-1$

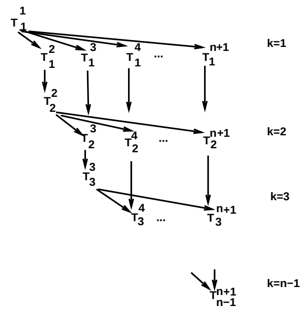
 T_k^k : **For** $i = k+1$ **to** n
   $a_{ik} = a_{ik}/a_{kk}$
  Endfor

 For $j = k+1$ **to** $n+1$

  T_k^j : **For** $i = k+1$ **to** n
    $a_{ij} = a_{ij} - a_{ik} * a_{kj}$
   Endfor

 Endfor
Endfor

Task graph:



Backward substitution on p processors

Partitioning: **For** $j = n$ **to** 1

 S_j^x : $x_j = x_j / a_{j,j}$;
  For $i = j-1$ **to** 1
   $x_i = x_i - a_{i,j} x_j$;
  Endfor

EndFor

Dependence:

$S_n^x \longrightarrow S_{n-1}^x \longrightarrow \cdots \longrightarrow S_1^x$.

Parallel algorithm: Execute all these tasks (S_j^x, $j = n, \cdots, 1$) gradually on the processor that owns x (column $n+1$).

The code is:

If owner(column x)==me then
 For $j = n$ **to** 1
  Receive column j if not available.
  Do S_j^x.
 EndFor
Else If owner(column j)==me, send column j to the owner of column x.

8 Gaussian elimination with partial pivoting

8.1 The sequential algorithm

Pivoting will avoid the problem when $a_{k,k}$ is too small or zero. We just need to change the forward elimination algorithm by adding the follow statements: at stage k, interchange rows such that $|a_{k,k}|$ is the maximum in the lower portion of the column k. Notice that b is stored in column $n+1$ and interchanging should also be applied to elements of b.

Example of GE with Pivoting:

$$\left(\begin{array}{ccc|c} 0 & 1 & 1 & 2 \\ 3 & 2 & -3 & 2 \\ 1 & 5 & -1 & 5 \end{array}\right) \overset{(1)\leftrightarrow(2)}{\Longrightarrow} \left(\begin{array}{ccc|c} 3 & 2 & -3 & 2 \\ 0 & 1 & 1 & 2 \\ 1 & 5 & -1 & 5 \end{array}\right) \overset{(3)-(1)*\frac{1}{3}}{\Longrightarrow} \left(\begin{array}{ccc|c} 3 & 2 & -3 & 2 \\ 0 & 1 & 1 & 2 \\ 0 & \frac{13}{3} & 0 & \frac{13}{3} \end{array}\right)$$

$$\overset{(2)\leftrightarrow(3)}{\Longrightarrow} \left(\begin{array}{ccc|c} 3 & 2 & -3 & 2 \\ 0 & \frac{13}{3} & 0 & \frac{13}{3} \\ 0 & 1 & 1 & 2 \end{array}\right) \overset{(3)-(2)*\frac{3}{13}}{\Longrightarrow} \left(\begin{array}{ccc|c} 3 & 2 & -3 & 2 \\ 0 & \frac{13}{3} & 0 & \frac{13}{3} \\ 0 & 0 & 1 & 1 \end{array}\right) \qquad \begin{aligned} x_1 &= 1 \\ x_2 &= 1 \\ x_3 &= 1 \end{aligned}$$

The backward substitution does not need any change.

Row-oriented forward elimination: **For** $k = 1$ **to** $n-1$
 Find m such that $|a_{m,k}| = \max_{n \geq i \geq k}\{|a_{i,k}|\}$;
 If $a_{m,k} = 0$, No unique solution, stop;
  Swap row(k) with row(m);
 For $i = k+1$ **to** n
  $a_{ik} = a_{ik}/a_{kk}$;
  For $j = k+1$ to $n+1$
   $a_{i,j} = a_{i,j} - a_{i,k} * a_{k,j}$;
  Endfor
 Endfor
Endfor

Column-oriented forward elimination: **For** $k = 1$ **to** $n-1$

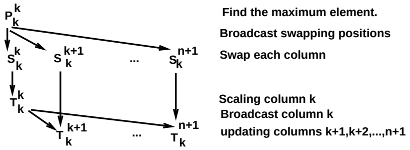
Find m such that $|a_{m,k}| = \max_{n \geq i \geq k}\{|a_{i,k}|\}$;
If $a_{m,k} = 0$, No unique solution, stop;
Swap row(k) with row(m);
For $i = k+1$ **to** n
  $a_{i,k} = a_{i,k}/a_{k,k}$
EndFor
For $j = k+1$ **to** $n+1$
  For $i = k+1$ **to** n
    $a_{i,j} = a_{i,j} - a_{i,k} * a_{k,j}$
  EndFor
EndFor
EndFor

8.2 Parallel column-oriented GE with pivoting

Partitioned forward elimination: **For** $k = 1$ **to** $n-1$

P_k^k: Find m such that $|a_{m,k}| = \max_{i \geq k}\{|a_{i,k}|\}$;
  If $a_{m,k} = 0$, No unique solution, stop.
For $j = k$ **to** $n+1$
  S_k^j : Swap $a_{k,j}$ with $a_{m,j}$;
Endfor
T_k^k: **For** $i = k+1$ **to** n
    $a_{i,k} = a_{i,k}/a_{k,k}$
  Endfor
For $j = k+1$ **to** $n+1$
  T_k^j **For** $i = k+1$ **to** n
      $a_{i,j} = a_{i,j} - a_{i,k} * a_{k,j}$
    Endfor
Endfor

Dependence structure for iteration k: The above partitioning produces the following dependence structure.

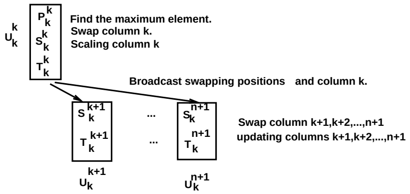


We can further merging tasks and combine small messages as:

- Define task U_k^k as performing P_k^k, S_k^k, and T_k^k.

- Define task U_k^j as performing S_k^j, and T_k^j $(k+1 \leq j \leq n+1)$.

Then the new graph has the following structure.



Parallel algorithm for GE with pivoting:

For $k = 1$ **to** $n-1$
  The owner of column k does U_k^k and broadcasts the swapping positions and column k.
  Do $U_k^{k+1} \ldots U_k^n$ in parallel.
Endfor

9 Iterative Methods for Solving Ax = b

9.1 Iterative methods

We use the following example to demonstrate the Jacobi iterative method. Given,

$$
\begin{array}{lrcl}
(1) & 6x_1 - 2x_2 + x_3 & = & 11 \\
(2) & -2x_1 + 7x_2 + 2x_3 & = & 5 \\
(3) & x_1 + 2x_2 - 5x_3 & = & -1
\end{array}
$$

We reformulate it as:

$$
\Longrightarrow \begin{array}{rcl}
x_1 & = & \frac{11}{6} - \frac{1}{6}(-2x_2 + x_3) \\
x_2 & = & \frac{5}{7} - \frac{1}{7}(-2x_1 + 2x_3) \\
x_3 & = & \frac{1}{5} - \frac{1}{-5}(x_1 + 2x_2)
\end{array}
\Longrightarrow \begin{array}{rcl}
x_1^{(k+1)} & = & \frac{1}{6}(11 - (-2x_2^{(k)} + x_3^{(k)})) \\
x_2^{(k+1)} & = & \frac{1}{7}(5 - (-2x_1^{(k)} + 2x_3^{(k)})) \\
x_3^{(k+1)} & = & \frac{1}{-5}(-1 - (x_1^{(k)} + 2x_2^{(k)}))
\end{array}
$$

We start from an initial approximation $x_1 = 0, x_2 = 0, x_3 = 0$. Then we can get a new set of values for x_1, x_2, x_3. We keep doing this until the difference from iteration k and $k+1$ is small, that means the error is small. The values of x_i for a number of iterations are listed below.

Iter	0	1	2	3	4	$\cdots$	8
x_1	0	1.833	2.038	2.085	2.004	$\cdots$	2.000
x_2	0	0.714	1.181	1.053	1.001	$\cdots$	1.000
x_3	0	0.2	0.852	1.080	1.038	$\cdots$	1.000

Iteration 8 actually delivers the final solution with error $< 10^{-3}$. Formally we stop when $\| \vec{x}^{(k+1)} - \vec{x}^{(k)} \| < 10^{-3}$ where $\vec{x}^k$ is a vector of the values of x_1, x_2, and x_3 after iteration k. We need to define **norm** $\| \vec{x}^{(k+1)} - \vec{x}^{(k)} \|$.

The general iterative method can be re-formulated as:

Assign an initial value to $\vec{x}^{(0)}$
k=0
Do
  $\vec{x}^{(k+1)} = H * \vec{x}^{(k)} + d$
until $\| \vec{x}^{(k+1)} - \vec{x}^{(k)} \| < \varepsilon$

For the above example, we have:

$$\begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix}^{k+1} = \begin{bmatrix} 0 & \frac{2}{6} & -\frac{1}{6} \\ \frac{2}{7} & 0 & -\frac{2}{7} \\ \frac{1}{5} & \frac{2}{5} & 0 \end{bmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix}^{k} + \begin{pmatrix} \frac{11}{6} \\ \frac{5}{7} \\ \frac{1}{5} \end{pmatrix}$$

9.2 Norms and Convergence

9.2.1 Norms of vectors and matrices

Norm of a vector. One application of vector norms is in error control, e.g. $\| Error \| < \varepsilon$. In the rest of discussion, we sometime simply use notation x instead of $\vec{x}$. Given $x = (x_1, x_2, \cdots x_n)$:

$$\| x \|_1 = \sum_{i=1}^{n} | x_i |, \quad \| x \|_2 = \sqrt{\sum | x_i |^2}, \quad \| x \|_\infty = max \mid x_i \mid .$$

Example:

$$x = (-1, 1, 2)$$

$$\| x \|_1 = 4, \quad \| x \|_2 = \sqrt{1 + 1 + 2^2} = \sqrt{6}, \quad \| x \|_\infty = 2.$$

Properties of norm:

$$\| x \| > 0, \quad \| x \| = 0 \quad \text{if } x = 0.$$

$$\| \alpha x \| = | \alpha | \| x \|, \quad \| x + y \| \leq \| x \| + \| y \| .$$

Norm of a matrix. Given a matrix of dimension $n \times n$, a matrix norm has the following property:

$$\| A \| > 0, \quad \| A \| = 0 \text{ if A=0 } .$$

$$\| \alpha A \| = | \alpha | \| A \|, \quad \| A + B \| \leq \| A \| + \| B \| .$$

$$\| AB \| \leq \| A \| \| B \|, \quad \| Ax \| \leq \| A \| \| x \| \text{ where x is vector } .$$

Define

$$\| A \|_\infty = \max_{1 \leq i \leq n} \sum_{j=1}^{n} a_{ij} \quad \textbf{max sum row}$$

$$\| A \|_1 = \max_{1 \leq j \leq n} \sum_{i=1}^{n} a_{ij} \quad \textbf{max sum column}$$

Example:

$$A = \begin{bmatrix} 5 & 9 \\ -2 & 1 \end{bmatrix}$$

$$\| A \|_\infty = max(5 + 9, 2 + 1) = 14, \quad \| A \|_1 = max(5 + 2, 9 + 1) = 10$$

9.2.2 Convergence of iterative methods

<u>Definition:</u> Let x^* be the exact solution and x^k be the solution vector after step k. Sequence $x^0, x^1, x^2, \cdots, x^n$ converges to the solution x^* with respect to norm $\| . \|$ if $\| x^k - x^* \| < \varepsilon$ when k is very large. Namely $k \to \infty, \| x^k - x^* \| \to 0$.

A condition for convergence: Let error $e^k = x^k - x^*$. Since

$$x^{k+1} = Hx^k + d, \quad x^* = Hx^* + d.$$

Then

$$x^{k+1} - x^* = H(x^k - x^*), \quad e^{k+1} = He^*.$$

We have

$$\| e^{k+1} \| \leq \| He^k \| \leq \| H \| \| e^k \| \leq \| H \|^2 \| e^{k-1} \| \leq \cdots \leq \| H \|^{k+1} \| e^0 \|$$

Then if $\| H \| < 1$, $\Longrightarrow$ The method converges.

9.3 Jacobi Method for Ax = b

For each iteration:

$$x_i^{k+1} = \frac{1}{a_{ii}} (b_i - \sum_{j>i} a_{ij} x_j^k) \quad i = 1, \cdots n$$

$$\textbf{Example:} \quad \begin{array}{lrcl} (1) & 6x_1 - 2x_2 + x_3 & = & 11 \\ (2) & -2x_1 + 7x_2 + 2x_3 & = & 5 \\ (3) & x_1 + 2x_2 - 5x_3 & = & \text{-1} \end{array} \Longrightarrow \begin{array}{lcl} x_1^{(k+1)} & = & \frac{1}{6}(11 - (-2x_2^{(k)} + x_3^{(k)})) \\ x_2^{(k+1)} & = & \frac{1}{7}(5 - (-2x_1^{(k)} + 2x_3^{(k)})) \\ x_3^{(k+1)} & = & \frac{1}{-5}(-1 - (x_1^{(k)} + 2x_2^{(k)})) \end{array}$$

Jacobi method in a matrix-vector form

$$\begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix}^{k+1} = \begin{bmatrix} 0 & \frac{2}{6} & -\frac{1}{6} \\ \frac{2}{7} & 0 & -\frac{2}{7} \\ \frac{1}{5} & \frac{2}{5} & 0 \end{bmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix}^{k} + \begin{pmatrix} \frac{11}{6} \\ \frac{5}{7} \\ \frac{1}{5} \end{pmatrix}$$

In general:

$$A = \begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{pmatrix}, \quad D = \begin{pmatrix} a_{11} & & & \\ & a_{22} & & \\ & & \ddots & \\ & & & a_{nn} \end{pmatrix}, \quad B = A - D.$$

Then from $Ax = b$,

$$(D + B)x = b.$$

Thus $Dx = -Bx + b$, then

$$x^{k+1} = -D^{-1}Bx^k + D^{-1}b$$

i.e.

$$H = -D^{-1}B, \quad d = D^{-1}b.$$

9.4 Parallel Jacobi Method

$$x^{k+1} = -D^{-1}Bx^k + D^{-1}b$$

Parallel solution:

- Distribute rows of B and the diagonal elements of D to processors.
- Perform computation based on the owner-computes rule.
- Perform all-all broadcasting after each iteration.

9.5 Gauss-Seidel Method

The basic idea is to utilize new solutions as soon as they are available. For example,

$$
\begin{array}{llcl}
(1) & 6x_1 - 2x_2 + x_3 & = & 11 \\
(2) & -2x_1 + 7x_2 + 2x_3 & = & 5 \\
(3) & x_1 + 2x_2 - 5x_3 & = & -1
\end{array}
$$

$$
\Longrightarrow \text{Jacobi method.} \quad
\begin{array}{lcl}
x_1^{k+1} & = & \frac{1}{6}(11 - (-2x_2^k + x_3^k)) \\
x_2^{k+1} & = & \frac{1}{7}(5 - (-2x_1^k + 2x_3^k)) \\
x_3^{k+1} & = & \frac{1}{-5}(-1 - (x_1^k + 2x_2^k))
\end{array}
$$

$$
\Longrightarrow \text{Gauss-Seidel method.} \quad
\begin{array}{lcl}
x_1^{k+1} & = & \frac{1}{6}(11 - (-2x_2^k + x_3^k)) \\
x_2^{k+1} & = & \frac{1}{7}(5 - (-2x_1^{k+1} + 2x_3^k)) \\
x_3^{k+1} & = & \frac{1}{-5}(-1 - (x_1^{k+1} + 2x_2^{k+1}))
\end{array}
$$

We show a number of iterations for the GS method, which converges faster than Jacobi's method.

	0	1	2	3	4	5
x_1	0	1.833	2.069	1.998	1.999	2.000
x_2	0	1.238	1.002	0.995	1.000	1.000
x_3	0	1.062	1.015	0.998	1.000	1.000

Formally the GS method is summarized as below. For each iteration:

$$x_i^{k+1} = \frac{1}{a_{ii}}(b_i - \sum_{j<i} a_{ij}x_j^{k+1} - \sum_{j\neq i} a_{ij}x_j^{(k)}), \quad i = 1, \cdots, n.$$

Matrix-vector Format. Let A=D+L+U where L is the lower triangular part of A. U is the upper triangular part.

$$D\vec{x}^{k+1} = b - L\vec{x}^{k+1} - U\vec{x}^k$$

$$(D + L)\vec{x}^{k+1} = b - U\vec{x}^k$$

$$\vec{x}^{k+1} = -(D + L)^{-1}U\vec{x}^k + (D + L)^{-1}b$$

$$
\textbf{Example:} \quad
\begin{array}{lcl}
x_1^{k+1} & = & \frac{1}{6}(11 - (-2x_2^k + x_3^k)) \\
x_2^{k+1} & = & \frac{1}{7}(5 - (-2x_1^{k+1} + 2x_3^k)) \\
x_3^{k+1} & = & \frac{1}{-5}(-1 - (x_1^{k+1} + 2x_2^{k+1}))
\end{array}
$$

$$
\begin{array}{llllccc}
6x_1^{k+1} & & & = & 2x_2^k & -x_3^k & +11 \\
-2x_1^{k+1} & +7x_2^{k+1} & & = & & -2x_3^k & +5 \\
x_1^{k+1} & +2x_2^{k+1} & -5x_3^{k+1} & = & & & -1
\end{array}
$$

$$
\begin{bmatrix} 6 & 0 & 0 \\ -2 & 7 & 0 \\ 1 & 2 & -5 \end{bmatrix}
\begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix}^{k+1}
=
\begin{bmatrix} 0 & 2 & -1 \\ 0 & 0 & -2 \\ 0 & 0 & 0 \end{bmatrix}
\begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix}^{k}
+
\begin{pmatrix} 11 \\ 5 \\ -1 \end{pmatrix}
$$

$$
H = \begin{bmatrix} 6 & 0 & 0 \\ -2 & 7 & 0 \\ 1 & 2 & -5 \end{bmatrix}^{-1} *
\begin{bmatrix} 0 & 2 & -1 \\ 0 & 0 & -2 \\ 0 & 0 & 0 \end{bmatrix}, \quad
d = \begin{bmatrix} 6 & 0 & 0 \\ -2 & 7 & 0 \\ 1 & 2 & -5 \end{bmatrix}^{-1} *
\begin{pmatrix} 11 \\ 5 \\ -1 \end{pmatrix}.
$$

More on convergence. We can actually judge the convergence of Jacobi and GS by examining A but not H.

We call a matrix A as **strictly diagonally dominant** if

$$| a_{ii} | > \sum_{j=1, j\neq i}^{n} | a_{ij} | \qquad \text{i=1,2, ...,n}$$

Theorem: If A is strictly diagonally dominant, then both Gauss-Seidel and Jacobi methods converge.

Example:

$$
\begin{pmatrix} 6 & -2 & 1 \\ -2 & 7 & 2 \\ 1 & 2 & -5 \end{pmatrix} x = \begin{pmatrix} 11 \\ 5 \\ -1 \end{pmatrix}
$$

A is strictly diagonally dominant:

$$| 6 | > 2 + 1, \quad 7 > 2 + 2, \quad 5 > 1 + 2$$

Then both Jacobi and G.S. methods will converge.

9.6 The SOR method

SOR stands for Successive Over Relaxation. The rate of convergence can be improved (accelerated) by the SOR method:

Step 1. Use the Gauss-Seidel method: $x^{k+1} = Hx^k + d$.

Step 2. Do a correction: $x^{k+1} = x^k + w(x^{k+1} - x^k)$.

10 Numerical Differentiation

Topics for the rest of the quarter: Finite-difference methods for solving ODE/PDEs.

1. Use numerical differentiation to approximate (partial) derivatives.
2. Set up a set of linear equations (usually sparse).
3. Solve the linear equations.

In this section, we will discuss how to approximate derivatives.

10.1 First-derivative formulas

We discuss formulas for computing the first derivative f'(x). The definition of the first derivative is:

$$f'(x) = \lim_{h\to 0} \frac{f(x + h) - f(x)}{h}$$

Thus the **forward difference method** is:

$$f'(x) \approx \frac{f(x + h) - f(x)}{h}$$

We can derive it using the Taylor's expansion:

$$f(x + h) = f(x) + hf'(x) + \frac{h^2}{2}f''(z)$$

where z is between x and $x + h$. Then

$$f'(x) = \frac{f(x + h) - f(x)}{h} - \frac{h}{2}f''(z).$$

Truncation error is $O(h)$.

There are other formulas:

- **Backward difference:**

$$f'(x) = \frac{f(x) - f(x-h)}{h} + \frac{h}{2} f''(z).$$

- **Central difference:**

$$f(x+h) = f(x) - hf'(x) + \frac{h^2}{2} f''(x) + \frac{h^3}{6} f^{(3)}(z_1),$$

$$f(x-h) = f(x) - hf'(x) + \frac{h^2}{2} f''(x) - \frac{h^3}{6} f^{(3)}(z_2).$$

Then

$$f(x+h) - f(x-h) = 2hf'(x) + \frac{h^3}{6}(f^{(3)}(z_1) - f^{(3)}(z_2)).$$

Thus the method is:

$$f(x) = \frac{f(x+h) - f(x-h)}{2h} + O(\frac{h^2}{3!}).$$

10.2 Central difference for second-derivatives

$$f(x+h) = f(x) + hf'(x) + \frac{h^2}{2} f''(x) + \frac{h^3}{3!} f^{(3)}(x) + \frac{h^4}{4!} f^{(4)}(z_1)$$

$$f(x-h) = f(x) - hf'(x) + \frac{h^2}{2} f''(x) - \frac{h^3}{3!} f^{(3)}(x) + \frac{h^4}{4!} f^{(4)}(z_2)$$

$$f(x+h) + f(x-h) = 2f(x) + h^2 f''(z) + O(\frac{h^4}{4!}).$$

Thus:

$$f''(x) = \frac{f(x+h) + f(x-h) - 2f(x)}{h^2} + O(\frac{h^2}{4!}).$$

10.3 Example

We approximate $f'(x) = (\cos x)'$ at $x = \frac{\pi}{6}$ using the forward difference formula.

i	h	$\frac{f(x+h)-f(x)}{h}$	Error E_i	$\frac{E_{i-1}}{E_i}$
1	0.1	-0.54243	0.04243	
2	0.05	-0.52144	0.02144	1.98
3	0.025	-0.51077	0.01077	1.99

From this case, we can see that this truncation error E_i is proportional to h. If h is halved, $\Longrightarrow$, the error is also halved.

We approximate $f''(x) = (\cos x)''$ at $x = \frac{\pi}{6}$. Using the central difference formula $f''(x) \approx (f(x+h) - 2f(x) + f(x-h))/h^2$, we have:

i	h	$f''(x) \approx$	Error E_i	$\frac{E_{i-1}}{E_i}$
1	0.5	-0.84813289	0.0178929	
2	0.25	-0.86152424	0.04504	3.97
3	0.125	-0.86489835	0.0127	3.99
4	0.0625	-0.86574353	0.002819	4.00

The truncation error Ei is proportional to h^2.

11 ODE and PDE

ODE stands for Ordinary Differential Equations. PDE stands for Partial Differential Equations.

An ODE Example: Growth of Population

- Population : $N(t)$, t is time.
- Assumption : Population in a state grows continuously with time at a birth rate (λ) proportional to the number of persons. Let δ be the average number of persons moving-in to this state (after subtracting the moving-out number.)

$$\frac{d\,N(t)}{dt} = \lambda N(t) + \delta.$$

- Let $\lambda = 0.01$. $\delta = 0.045$ million.

$$\frac{d\,N(t)}{dt} = 0.01N(t) + 0.045.$$

- If $N(1990) = 1million$, what are $N(1991), N(1992), N(1993), \cdots, N(1999)$?

Other Examples

-

$$f'(x) = x \quad f(0) = 0.$$

What are $f(0.1), f(0.2), \cdots, f(0.5)$?

-

$$f"(x) = 1, \quad f(0) = 1, f(1) = 1.5.$$

What are $f(0.2), f(0.4), f(0.6), f(0.8)$?

- **The Laplace PDE.**

$$\frac{\partial^2 U(x,y)}{\partial x^2} + \frac{\partial^2 U(x,y)}{\partial y^2} = 0.$$

The domain is a 2D region. Usually we know some boundary values or condition and we need to find values for some points within this region.

11.1 Finite Difference Method

1. Discretize a region or interval of variables, or domain of this function.
2. For each point in the discretized domain, setup an equation using a numerical differentiation formula.
3. Solve the linear equations.

Example. Given:

$$y'(x) = 1, \quad y(0) = 0$$

Domain x: $0,\ 1h,\ 2h, \cdots,\ nh$.

Find: $y(h), y(2h), \cdots, y(nh)$.

Method: At each point $x = ih$, $\quad 1 = y'(ih) \approx \frac{y((i+1)h) - y(ih)}{h}$.

Thus

$$y((i+1)h) - y(ih) = h \quad \text{for } i = 0, 1, \cdots n-1.$$

$$y(h) - y(0) = h$$

$$y(2h) - y(h) = h$$

$$\vdots$$

$$y(nh) - y((n-1)h) = h$$

$\Longrightarrow$

$$\begin{pmatrix} 1 & & & & \\ -1 & 1 & & & \\ & -1 & 1 & & \\ & & & \ddots & \\ & & & -1 & 1 \end{pmatrix} \begin{pmatrix} y(h) \\ y(2h) \\ \vdots \\ y((n-1)h) \\ y(nh) \end{pmatrix} = \begin{pmatrix} h \\ h \\ \vdots \\ h \\ h \end{pmatrix}$$

Example. Given:

$$y''(x) = 1, \quad y(0) = 0, \quad y(1) = 0$$

Domain: $0, x_1, x_2, \cdots, x_n, 1$.

Let $x_i = i * n$ and $y_i = y(i * h)$ where $h = \frac{1}{n+1}$.

Find: $y_1, y_2, \cdots, y_n$.

Method: At each point x_i,

$$1 = y''(x_i) \approx \frac{y_{i+1} - 2y_i + y_{i-1}}{h^2}$$

Thus

$$y_{i+1} - 2y_i + y_{i-1} = h^2 \quad \text{For } i = 1, \cdots n.$$

Then

$$y_0 - 2y_1 + y_2 = h^2$$

$$y_1 - 2y_2 + y_3 = h^2$$

$$\vdots$$

$$y_{n-1} - 2y_n + y_{n+1} = h^2$$

i.e.

$$\begin{pmatrix} -2 & 1 & & & \\ 1 & -2 & 1 & & \\ & 1 & -2 & 1 & \\ & & & \ddots & 1 \\ & & & 1 & -2 \end{pmatrix} \begin{pmatrix} y_1 \\ y_2 \\ \vdots \\ y_{n-1} \\ y_n \end{pmatrix} = \begin{pmatrix} h^2 \\ h^2 \\ \vdots \\ h^2 \\ h^2 \end{pmatrix}$$

11.2 GE for solving linear tridiagonal systems

$$\begin{bmatrix} a_1 & b_1 & & & \\ c_2 & a_2 & b_2 & & \\ & c_3 & a_3 & b_3 & \\ & & & \ddots & \\ & & & c_n & a_n \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ \vdots \; x_n \end{bmatrix} = \begin{bmatrix} d_1 \\ d_2 \\ d_3 \\ \vdots \\ d_n \end{bmatrix}$$

Forward Elimination:

For i=2 to n-1
  $temp = \frac{a_{i,i-1}}{a_{i-1,i-1}}$
  $a_{ii} = a_{ii} - a_{i-1,i} * temp$
  $b_i = b_i - b_{i-1} * temp$
EndFor

Backward Substitution: $x_n = \frac{b_n}{c_{nn}}$
For i=n-1 to 1
  $x_i = (b_i - x_{i+1} * a_{i,i+1})/a_{ii}$
EndFor

Parallel solutions for tridiagonal systems. There is no parallelism in the above GE method. We discuss an algorithm called the Odd-Even Reduction method (or called Cyclic Reduction).

Basic idea: The basic idea is to eliminate the odd-numbered variables from the n equations and formulate $n/2$ equations. This reduction process is modeled as a tree dependence structure and can be parallelized. For example:

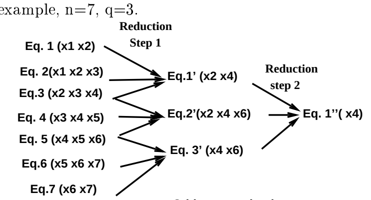
$$\begin{array}{lrcl} (1) & a_1x_1 + b_1x_2 & = & d_1 \\ (2) & c_2x_1 + a_2x_2 + b_2x_3 & = & d_2 \\ (3) & c_3x_2 + a_3x_3 + b_3x_4 & = & d_3 \end{array}$$

$$\Longrightarrow a_2'x_2 + b_2'x_4 = d_2'.$$

$$\begin{array}{lrcl} (3) & c_3x_2 + a_3x_3 + b_3x_4 & = & d_3 \\ (4) & c_4x_3 + a_4x_4 + b_4x_5 & = & d_4 \\ (5) & c_5x_4 + a_5x_5 + b_5x_6 & = & d_5 \end{array}$$

$$\Longrightarrow c_3'x_2 + a_4'x_4 + b_5'x_6 = d_4.$$

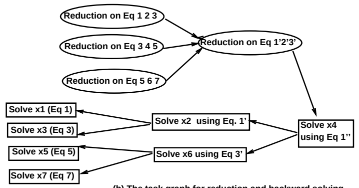
Odd-even reduction part: In general, given $n = 2^q - 1$ equations, one reduction step eliminates all odd variables and reduces the system to $2^{q-1} - 1$ equations. Recursively applying such reduction in $\log n$ steps (q-1), we finally have one equation with one variable. For example, n=7, q=3.



(a) Odd–even reduction process.

Backward substitution part: After one equation with one variable is derived, the solution for this variable can be found. Then backward substitution process is applied recursively in $\log n$ steps to find all solutions.

A task graph for odd-even reduction and back substitution is:



(b) The task graph for reduction and backward solving.

11.3 PDE: Laplace's Equation

We demonstrate the sequential and parallel solutions for a PDE that models a steady-state heat-flow problem on a rectangular $10cm \times 20cm$ metal sheet. One edge maintains temperature of 100 degree, other three edges maintain 0 degree as shown in Figure 30. What are the steady-state temperatures at interior points?

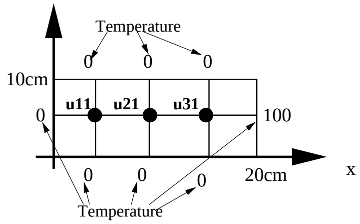


Figure 30: The problem domain for a Laplace equation in modeling the steady-state temperature.

The mathematical model is the Laplace equation:

$$\frac{\partial^2 u(x,y)}{\partial x^2} + \frac{\partial^2 u(x,y)}{\partial y^2} = 0$$

with the boundary condition:

$$u(x,0) = 0, \quad u(x,10) = 0.$$

$$u(0,y) = 0, \quad u(20,y) = 100.$$

We divide the region into a grid with gap h at each axis. At each point (ih, jh), let $u(ih, jh) = u_{i,j}$. The goal is to find the value of all points $u_{i,j}$.

Numerical Solution: Use the approximating formula for numerical differentiation:

$$f''(x) \approx \frac{f(x+h) + f(x-h) - 2f(x)}{h^2}$$

Thus

$$\frac{\partial^2 u(x_i, y_i)}{\partial x^2} \approx \frac{u_{i+1,j} - 2u_{i,j} + u_{i-1,j}}{h^2}$$

$$\frac{\partial^2 u(x_i, y_i)}{\partial y^2} \approx \frac{u_{i,j+1} - 2u_{i,j} + u_{i,j-1}}{h^2}$$

Then

$$u_{i+1,j} - 2u_{ij} + u_{i-1,j} + u_{i,j+1} - 2u_{i,j} + u_{i,j-1} = 0$$

or

$$4u_{i,j} - u_{i+1,j} - u_{i-1,j} - u_{i,j+1} - u_{i,j-1} = 0$$

Example. For the case in Figure 30, Let $u_{11} = x_1, u_{21} = x_2, u_{31} = x_3$.

$$\text{At } u_{11}, \quad 4x_1 - 0 - 0 - x_2 = 0$$

$$\text{At } u_{21}, \quad 4x_2 - x_1 - 0 - x_3 - 0 = 0$$

$$\text{At } u_{31}, \quad 4x_3 - x_2 - 0 - 100 - 0 = 0$$

$$\begin{bmatrix} 4 & -1 & 0 \\ -1 & 4 & -1 \\ 0 & -1 & 4 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 100 \end{bmatrix}$$

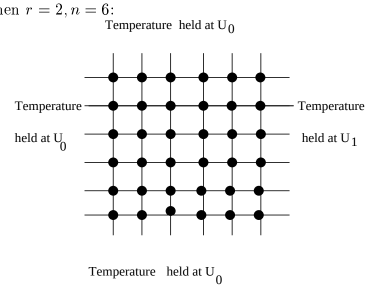
Solutions:

$$x_1 = 1.786, \quad x_2 = 7.143, \quad x_3 = 26.786$$

A general grid. Given a general $(n+2) \times (n+2)$ grid, we have n^2 equations:

$$4u_{i,j} - u_{i+1,j} - u_{i-1,j} - u_{i,j+1} - u_{i,j-1} = 0$$

for $1 \le i, j \le n$. For example, when $r = 2, n = 6$:



We order the unknowns as $(u_{11}, u_{12}, \cdots, u_{1n}, u_{21}, u_{22}, \cdots, u_{2n}, \cdots, u_{n1}, \cdots, u_{nn})$.

For $n = 2$, the ordering is:

$$\begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix} = \begin{bmatrix} u_{11} \\ u_{12} \\ u_{21} \\ u_{22} \end{bmatrix}$$

The matrix is:

$$\begin{bmatrix} 4 & -1 & -1 & 0 \\ -1 & 4 & 0 & -1 \\ -1 & 0 & 4 & -1 \\ 0 & -1 & -1 & 4 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix} = \begin{bmatrix} u_{01} + u_{10} \\ u_{20} + u_{31} \\ u_{02} + u_{13} \\ u_{32} + u_{23} \end{bmatrix}$$

In general, the left side matrix is:

$$\begin{bmatrix} T & -I & & & \\ -I & T & -I & & \\ & -I & T & -I & \\ & & \ddots & \ddots & \ddots \\ & & & -I & T \end{bmatrix}_{n^2 \times n^2} \quad T = \begin{bmatrix} 4 & -1 & & & \\ -1 & 4 & -1 & & \\ & -1 & 4 & -1 & \\ & & \ddots & \ddots & \ddots \\ & & & -1 & 4 \end{bmatrix}_{n \times n} \quad I = \begin{bmatrix} 1 & & & & \\ & 1 & & & \\ & & 1 & & \\ & & & \ddots & \\ & & & & 1 \end{bmatrix}_{n \times n}$$

The matrix is very sparse, and a direct method for solving this system takes too much time.

The Jacobi Iterative Method. Given

$$4u_{i,j} - u_{i+1,j} - u_{i-1,j} - u_{i,j+1} - u_{i,j-1} = 0$$

for $1 \le i, j \le n$. The Jacobi program is:

Repeat
  For i=1 to n
    For j=1 to n
      $u^{new}_{i,j} = 0.25(u_{i+1,j} + u_{i-1,j} + u_{i,j+1} + u_{i,j-1})$.
    EndFor
  EndFor
Until $\| u^{new} - u \| < \epsilon$

Parallel Jacobi Method. Assume we have a mesh of $n \times n$ processors. Assign $u_{i,j}$ to processor $p_{i,j}$.

The SPMD Jacobi program at processor $p_{i,j}$:

Repeat
  Collect data from four neighbors: $u_{i+1,j}$, $u_{i-1,j}$, $u_{i,j+1}$, $u_{i,j-1}$ from $p_{i+1,j}$, $p_{i-1,j}$, $p_{i,j+1}$, $p_{i,j-1}$.
  $u^{new}_{i,j} = 0.25(u_{i+1,j} + u_{i-1,j} + u_{i,j+1} + u_{i,j-1})$.
  $diff_{i,j} = |u^{new}_{ij} - u_{ij}|$
  Do a global reduction to get the maximum of $diff_{i,j}$ as M.
Until $M < \epsilon$.

Performance evaluation.

- Each computation step takes $\omega = 5$ operations for each grid point.
- There are 4 data items to be received. The size of each item is 1 unit. Assume sequential receiving and then communication costs $4(\alpha + \beta)$ for these 4 items.
- Assume that the global reduction takes $(\alpha + \beta) \log n$.
- The sequential time $Seq = K\omega n^2$ where K is the number of steps for convergence.
- Assume $\omega = 0.5, \beta = 0.1, \alpha = 100, n = 500, p^2 = 2500$.

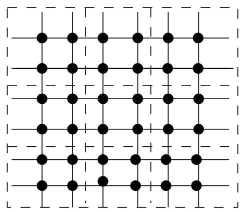
The parallel time $PT = K(\omega + (4 + \log n)(\alpha + \beta))$.

$$\text{Speedup} = \frac{\omega * n^2}{\omega + (4 + \log n)(\alpha + \beta)} \approx 192.$$

$$\text{Efficiency} = \frac{Speedup}{n^2} = 7.7\%.$$

In practice, the number of processors is much less than n^2. Also the above analysis shows that it does not make sense to utilize n^2 processors since the code suffers too much communication overhead for fine-grain computation. To increase the granularity of computation, we map the $n \times n$ grid to processors using 2D block method.

Grid partitioning. Assume that the grid is mapped to a $p \times p$ mesh where $p << n$. Let $\gamma = \frac{n}{p}$. An example of a mapping with $\gamma = 2, n = 6$ is shown below.



Code partitioning. Re-write the kernel part of the sequential code as:

For $bi = 1$ to p
 For $bj = 1$ to p
  For $i = (b_i - 1)\gamma + 1$ to $b_i\gamma$
   For $j = (b_j - 1)\gamma + 1$ to $b_j\gamma$
    $u^{new}_{i,j} = 0.25(u_{i+1,j} + u_{i-1,j} + u_{i,j+1} + u_{i,j-1})$.
    EndFor
   EndFor
  EndFor
 EndFor

Parallel SPMD code. On processor p_{b_i,b_j}:

Repeat
  Collect the data from its four neighbors.
  For $i = (b_i - 1)\gamma + 1$ to $b_i\gamma$
   For $j = (b_j - 1)\gamma + 1$ to $b_j\gamma$
    $u^{new}_{i,j} = 0.25(u_{i+1,j} + u_{i-1,j} + u_{i,j+1} + u_{i,j-1})$.
   EndFor
  EndFor
  Compute the local maximum $diff_{b_i,b_j}$ for the difference between old values and new values.
  Do a global reduction to get the maximum $diff_{b_i,b_j}$ as M.
Until $M < \epsilon$.

Performance evaluation.

- At each processor, each computation step takes $\omega\omega^2$ operations.
- The communication cost for each step on each processor is $4(\alpha + \omega\beta)$.
- Assume that the global reduction takes $(\alpha + \beta) \log p$.
- The number of steps for convergence is K.
- Assume $\omega = 0.5, \beta = 0.1,\ \alpha = 100,\ n = 500, \omega = 100, p^2 = 25$.

$$PT = K(\omega^2\omega + (4 + \log p)(\alpha + r\beta))$$

$$\text{Speedup} = \frac{\omega\omega^2 p^2}{\omega^2\omega + (4 + \log p)(\alpha + \omega\beta)} \approx 21.2.$$

$$\text{Efficiency} = 84\%.$$

A MPI Parallel Programming on Meiko CS-2 at UCSB

A.1 Introduction to Meiko CS-2

CS-2 is a parallel machine manufactured by the Meiko company. The Department of Computer Science at UCSB purchased a 64-processor CS-2 in June 1994.

Hardware architecture. The Meiko CS-2 is essentially a collection of SPARCstation 10s with a fast network interconnection. Each node has a SUN 40MHz SPARC Viking superscalar processor for computation with a peak performance 40M flop per second. Each node also has a Sparc-like processor to deal with communication, 32MB memory and 1MB external cache. The structure of the CS-2 data communication network (called Elan) is a multi-stage packet-switching fat-tree. The Elan network provides 40MB/sec node-to-node sustainable bandwidth with $10\mu s$ latency.

Software. All nodes run the Solaris 2.3 operating system and each node can be regarded as a SUN workstation. Any binary code that runs in a SUN workstation should be able to run in each CS-2 node. The OS at each node also has extensions to support the Elan communication network. The Message passing paradigms supported:

- NX/2 - Intel communication interface.
- MPI - The new standard communication interface.
- PVM - software for supporting parallel programming on Ethernet-connected workstations.
- CSN - traditional Meiko communication library.
- EW - The low-level Meiko interface to the network (Elan Widgets).

Other software supported are: Parallel debuggers (Totalview from BBN), Adaptor and HPF (when available), and /usr/compsci/*/bin standard CS software. Some useful system utilities: prun and rinfo (commands for parallel running of programs) and pandora (a graphical program for viewing machine state).

Configuration. The machine is configured with a single login node (m.cs.ucsb.edu), seven server nodes, and 56 diskless clients. Client nodes named m000-m055. Server nodes named m096, m100, m104, ..., m120. The machine meiko.cs.ucsb.edu is not actually the Meiko itself, but a SPARCstation 10 file server. The bulk of the disk space available on the Meiko resides here, and this is the site of user home directories. For performance and security reasons, CS home directories are not mounted on the Meiko CS-2. However on the file server, both standard home directories and Meiko home directories are available. The Meiko home directories are in /fs/meiko-user or /fs/meiko-xuser. Meiko system files are in /fs/meiko-sys.

The Meiko CS-2 principal login node is called m.cs.ucsb.edu (m124). Current client partitions: servers, p32, p16, p4 p2a p2b. To see the status of the partitions, run the command 'rinfo' on the Meiko login node.

Disk configuration. m124 node contains shared system files (/opt) and a user accessible partition (/export/tmphome). /export/tmphome IS NOT BACKED UP. It may be cleared at anytime. Server nodes contain root/swap/usr for self and 8 clients. Server nodes also contain a 7x122MB parallel file system partition. The /usr/compsci file system is mounted on m124. This file system contains the standard CS binaries, including X11 and GNU applications.

Documentation. Investigate directories in /opt/MEIKOcs2 for postscript documentation, application examples, include files, libraries, parallel packages, etc. Investigate directories in /opt/MEIKOcs2 for postscript documentation. Also check /fs/meiko-sys on file server. SPARC compiler docs in /opt/SUNWspro. Meiko answerbook in /opt/Mkab001. The answerbook is intended to be used under OpenWindows, which supports some additional features over standard X11. However you can also read the postscript files with ghostview (/usr/compsci/X11/bin/gv). Directory /fs/meiko-user/shared contains user-contributed examples and programs.

You can use: man -M /opt/MEIKOcs2/man rinfo command to view manual page of a command (say rinfo). After setting up the manual path (see below), you can directly type “man rinfo”. The MPI man pages are in /fs/meiko-user/shared/mpi/man.

More Help Problems should be reported to cs-meiko@cs. This is the formal way to report some problems, or difficulty you are having. Theron Lewis is the system manager for the Meiko CS-2 facility at UCSB.

A.2 Basic Steps

A.2.1 Basic issues

You need to first login to a machine on the Internet (e.g. in CSIL or the ECI lab), then login to the Meiko. Before logging in, make sure that you have disabled access control, so that X clients can connect from Meiko. Just type ”xhost m” or add Meiko to your .xhosts file. From the shell prompt of your workstation type rlogin or telnet m, then enter your login name and your password if required.

To tell X applications to display on your login terminal, set the environment variable DISPLAY. Type setenv DISPLAY “workstation_name:0.0”.

Setup commands have been included in the sample .cshrc/.login files as described below (110Bcshrc.cshrc and 110Blogin.login files). Basically it includes /opt/MEIKOcs2/bin in the path, and includes /opt/SUNWspro/bin to get access to the Solaris node compilers for C, C++, and Fortran.

You can type: ls /opt/MEIKOcs2/man/man1 to see manual pages available. Type: man -M /opt/MEIKOcs2/man command to view manual page. The 110B login setup file includes /opt/MEIKOcs2/man in the man path:

```
setenv MANPATH /opt/MEIKOcs2/man:/fs/meiko-user/shared/mpi/man:${MANPATH}
```

Thus you can simply type a command such as “man prun” for viewing a manual page without specifying the manual path.

A.2.2 Login

Command: telnet m or rlogin m.

First Time. Copy the following two files to your home directory.

```
cp /fs/meiko-user/tyang/cs110b97/110Bcshrc .cshrc
cp /fs/meiko-user/tyang/cs110b97/110Blogin .login
```

The above sample cshrc/login files will setup the necessary paths and environment variables for you to do 110B programming assignments. Then you re-login again.

A.2.3 Where to store files

Normally each user is assigned two directories. For example, my home directory is /fs/meiko-user/tyang in which I edit and compile the code, another directory is /export/tmphome/tyang where I execute the program. The files under /fs/meiko-user are backed up every few days. But the files under /export/tmphome are not backed up. The Meiko parallel nodes can only access /export/tmphome thus the binary code has to be executed under that directory. Your own exeuction directory is linked to ∼/Mtmp.

A.2.4 Run sample MPI programs

Directory /fs/meiko-user/tyang/cs110b97/mpi/example contains a set of MPI programs. Files under that sample directory are:

```
Makefile  --  Makefile for sample C programs. You can modify for your code.
greetings.c   -- C hello program. It demonstrates how SPMD and simple communication work.
cpi.c   -- Compute pi=3.1415... with numerical integration.
cpilog.c   -- C program for computing pi. It also asks for the input of interval information.
pi3.f   -- Fortran program for computing pi.
```

To run the sample program, you can first copy those samples under your own home directory. To compile the greeting code, you type "make greetings". Then you move the binary *greetings* to $\sim /Mtmp$.

Then type "rinfo" to get the current status of the machine. For example, rinfo may return the following information in the screen:

```
PARTITION     NPROC     PROCS    STATUS        TIME
      p32        32      0-31      down
      p16        16     32-47        up     7:47:41
    login         1       124      down
      p2a         2     48-49        up     7:47:48
      p2b         2     50-51        up     7:47:50
       p4         4     52-55        up     6:41:43

 RESOURCE     NPROC    STATUS      USER        GPID        TIME       LIMIT
   p2a.13         2      free                           6:26:38
   p2b.28         2      free                           5:36:25
   p4.142         4    in-use     tyang    124.3534        0:02        none

     USER     NPROC    STATUS PARTITION        GPID        TIME     COMMAND
    tyang         4   running        p4    124.3534        0:02  matrix_para
```

The above screen information indicates that partition p32 is down, p16 is working and is free, and partition p4 is working but is occupied by user tyang. Then you can use partition p16 to run this program. Partition p16 contains 16 nodes. We can select 4 nodes.

```
cd ~/Mtmp
prun -pp16 -n4 greetings
```

A.2.5 Edit and run your own program

The best is that you edit files in a CSIL machine, then copy files to meiko directory (use ftp or rcp). In compiling a program, you can follow the Makefile in the sample directory or use the following commands.

```
cc -O -I/fs/meiko-user/shared/mpi/include -c cpi.c
cc -O -o cpi cpi.o -L/fs/meiko-user/shared/mpi/lib/meiko/meiko
    -lmpi -L/opt/MEIKOcs2/lib -lew -lelan -lm
or
f77 -O -o pi3 pi3.f -L/fs/meiko-user/shared/mpi/lib/meiko/meiko
     -lmpi -L/opt/MEIKOcs2/lib -lew -lelan
```


To execute, be sure you are logged onto one of the server nodes (eg. m124.cs). make sure that your current directory is under ∼/Mtmp and the compiled binary code is also under that directory.

Killing your job If your program does not work or it hangs, use control-C to kill it. Don't leave your program running before you logout, otherwise other people cannot use the partition you are occupying.

Every time when you logout, you should use rinfo and ps to check if you have any job running.

A.2.6 Other useful information

The ftp utility:

```
ftp m
ftp> cd <directory>
ftp> binary
ftp> get <filename>
ftp> put <filename>
ftp> bye
```

The remote-file copying command: For example, you can add a machine name dagwood in your .rhosts file in machine m, then you can use "rcp filename m:" on dagwood. After adding m in your .rhosts file of machine dagwood, then at m you can use " rcp filename dagwood:". An example of file .rhosts:

```
dagwood tyang
m tyang
```

A.3 More MPI sample programs

/fs/meiko-user/shared/mpi contains more MPI information and examples. All sample programs from the text book "Parallel Programming with MPI" are in directory ∼tyang/cs110b97/mpi/ppmpi_c. The following is a *partial* list of the MPI programs from this book and "Page" refers to the first page in the text book that discusses about the corresponding code.

```
Page  File(s) and Description
----  ----------------------
 41   chap3/greetings.c -- greetings program
 55   chap4/serial.c -- serial trapezoidal rule
 57   chap4/trap.c -- parallel trapezoidal rule, first version
 61   chap4/get_data.c -- parallel trap. rule, reads and distributes input using linear for loops.
 67   chap5/get_data1.c -- parallel trap. rule, uses hand-coded, tree-
          structured broadcast to distribute input.
 70   chap5/get_data2.c -- parallel trap. rule, uses 3 calls to MPI_Bcast
          to distribute input.
 74   chap5/reduce.c -- parallel trap. rule, uses 3 calls to MPI_Bcast
          to distribute input and MPI_Reduce to compute final sum.
 75   chap5/serial_dot.c -- serial dot product
 76   chap5/parallel_dot.c -- parallel dot product
 78   chap5/parallel_dot1.c -- parallel dot product using MPI_Allreduce
 78   chap5/serial_mat_vect.c -- serial matrix-vector product
 83   chap5/parallel_mat_vect.c -- parallel matrix-vector product
 90   chap6/count.c -- send a subarray using count parameter
 93   chap6/get_data3.c -- parallel trap. rule, builds derived datatype for use with distribution of input
 96   chap6/send_row.c -- send row of a matrix
 97   chap6/send_col.c -- use derived datatype to send a column of a matrix
```

98 chap6/send_triangle.c -- use derived datatype to send upper triangle of a matrix
100 chap6/send_col_to_row.c -- send a row of a matrix on one process to a column on another
100 chap6/get_data4.c -- parallel trap. rule, use MPI_Pack/Unpack in distribution of input
104 chap6/sparse_row.c -- use MPI_Pack/Unpack to send a row of sparse matrix
113 chap7/serial_mat_mult.c -- serial matrix multiplication of two square matrices
118 chap7/comm_create.c -- build a communicator using MPI_Comm_create
118 chap7/comm_test.c -- tests communicator built using MPI_Comm_create
120 chap7/comm_split.c -- builds a collection of communicators using MPI_Comm_split
121 chap7/top_fcns.c -- builds and tests basic Cartesian topology functions
125 chap7/fox.c -- uses Fox's algorithm to multiply two square matrices
140 chap8/cache_test.c -- cache and retrieve a process rank attribute
143 chap8/cio.c, cio.h, cio_test.c, vsscanf.c, vsscanf.h, Makefile.cio --
functions for basic collective I/O
154 chap8/stdin_test.c -- test whether an MPI implementation allows input from stdin.
154 chap8/arg_test.c -- test whether an MPI implementation allows
processes access to command line arguments
157 chap8/cfopen.c -- open a file
157 chap8/multi_files.c -- each process opens and writes to a different file.
165 chap8/ub.c -- build a derived type that uses MPI_UB
166 chap8/cyclic_io.c, cyclic_io.h, sum.c, Makefile.sum -- functions for
array I/O using cyclic distribution
180 chap9/bug.c -- bugged serial insertion sort
188 chap9/mat_mult.c -- nondeterministic matrix multiplication
192 chap9/comm_time_0.c -- initial ring pass program
200 chap9/comm_time_1.c -- added first debugging output
202 chap9/comm_time_2.c -- ring pass started by process 0
204 chap9/comm_time_2a.c -- printf added to Print_results
205 chap9/comm_time_3.c -- fixed incorrect calculation of *order_ptr
206 chap9/comm_time_3a.c -- two different message sizes
206 chap9/comm_time_4.c -- removed printf from Print_results
208 chap9/comm_time_5.c -- checking behavior of message-passing functions by adding Cprintf's
209 chap9/comm_time_6.c -- relocate pesky Cprintf's
210 chap9/comm_time_7.c -- remove debug output, change number of tests
211 chap9/err_handler.c -- test changing default error handler in MPI to MPI_ERRORS_RETURN

218 chap10/serial_jacobi.c -- serial version of Jacobi's method
223 chap10/parallel_jacobi.c -- parallel version of Jacobi's method
226 chap10/sort_1.c, sort_1.h -- level 1 version of sort program
231 chap10/sort_2.c, sort_2.h -- add Get_list_size, Allocate_list, and Get_local_keys
234 chap10/sort_3.c, sort_3.h -- add Redistribute_keys, finish
Allocate_list, add Insert, Local_sort, and Print_list
237 chap10/sort_4.c, sort_4.h -- add Find_alltoall_send_params,
Find_cutoff, and Find_recv_displacements. Allow user input list size
255 chap11/parallel_trap.c -- parallel trapezoidal rule with code for taking timings.
267 chap12/ping_pong.c -- two process ping-pong
268 chap12/send.c, bcast.c -- simple example showing MPI's profiling interface
283 chap13/ag_ring_blk.c -- ring allgather using blocking send/recv
292 chap13/ag_cube_blk.c -- hypercube allgather using blocking send/recv
298 chap13/ag_ring_nblk.c -- ring allgather using nonblocking communications
299 chap13/ag_cube_nblk.c -- hypercube allgather using nonblocking communications
301 chap13/ag_ring_pers.c -- ring allgather using persistent communication requests
305 chap13/ag_ring_syn.c -- ring allgather using synchronous sends
307 chap13/ag_ring_rdy.c -- ring allgather using ready mode sends
309 chap13/ag_ring_buf.c -- ring allgather using buffered mode sends