Lecture Notes on Parallel Scientific Computing

Tao Yang
Department of Computer Science
University of California at Santa Barbara

Contents

1 Design and Implementation of Parallel Algorithms

1.1 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 distributed memory 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.

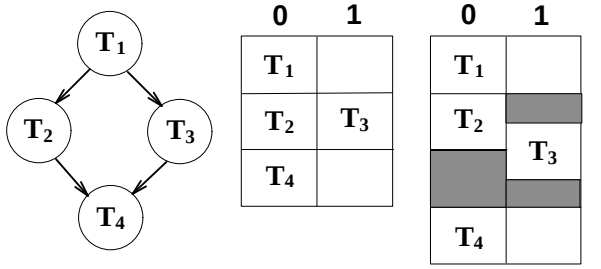
Scheduling is defined by a processor assignment mapping, $PA(T_x)$, of the tasks T_x onto the p processors and by a starting time mapping, $ST(T_x)$, of all nodes onto the real positive numbers set. $CT(T_x) = ST(T_x) + \tau_x$ is defined as the completion time of task T_x in this schedule.

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).



(a) (b) (c)

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

(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). The length of critical path is also called *the graph span*.

  Then the following conditions must be true. **Span law:**

$$\text{Parallel time} \geq \text{Length of the critical path},$$

  Work law:

$$\text{Parallel time} \geq \frac{\text{Sequential time}}{p}$$

  In addition,

$$\text{Speedup} \ \leq \ \text{Degree of parallelism.}$$

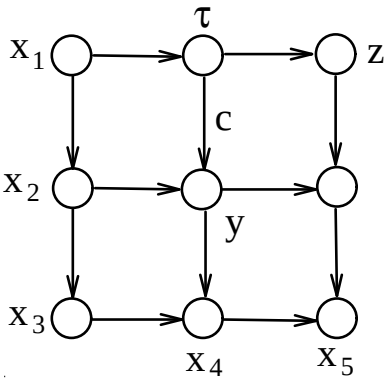


Figure 2: A DAG with node weights equal to 1.

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

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

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

1.2 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.

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;

2 Issues in Network Communication

2.1 Message routing for one-to-one sending

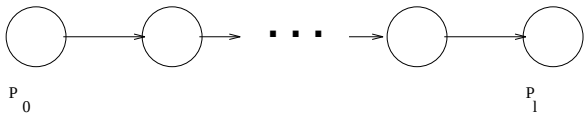


Figure 3: Routing a message from processor P_0 to P_l.

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 3 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.

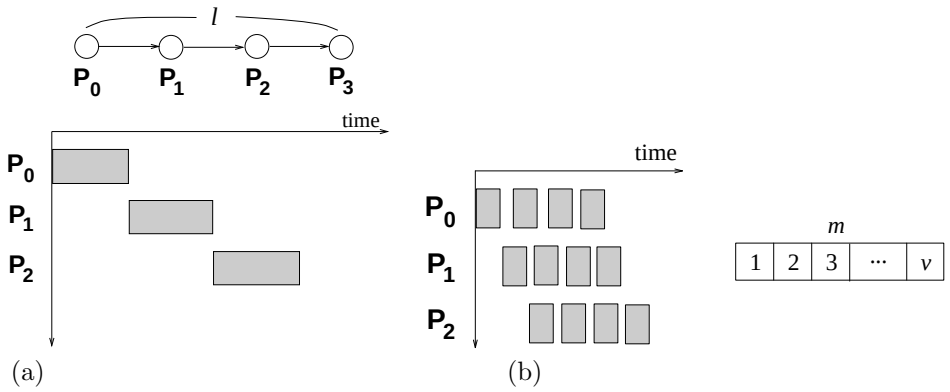


Figure 4: (a) An example of the store-forward routing model. (b) An example of wormhole model.

The store-forward model. Figure 4(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 4(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:

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

Thus

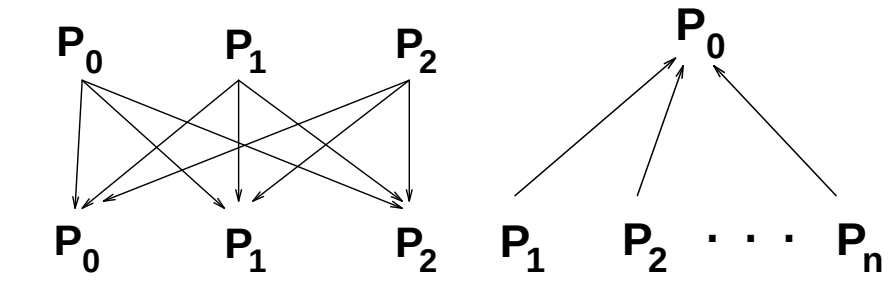
$$\begin{array}{rcl} t_{comm} & = & \alpha \ + \ m\beta \ + \frac{m\beta l}{v} + t_h(l + v) \\ & = & \alpha \ + \ m\beta \ + \ t_h(l + 2v) \\ & \approx & \alpha \ + \ \beta m. \end{array}$$

Thus the node distance (i.e., the number of hops between two nodes) does not play a significant role in the above communication cost.

2.2 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 various communication patterns in a parallel program and we list popular communication opeations 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 Figure5(a).



(a) (b)

Figure 5: (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 5(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.

2.3 One-to-All Broadcast

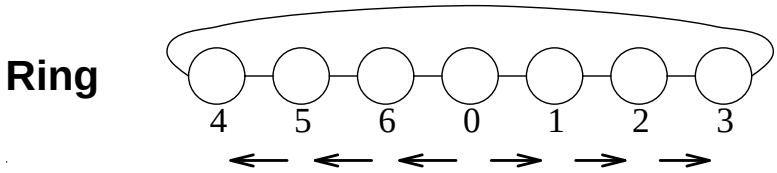


Figure 6: Broadcasting on a ring with the store-forward model.

We first discuss the implementation of broadcast on a ring using the store-forward routing model. Figure 6 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 7 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: $2\sqrt{p}(\alpha + \beta m)$.

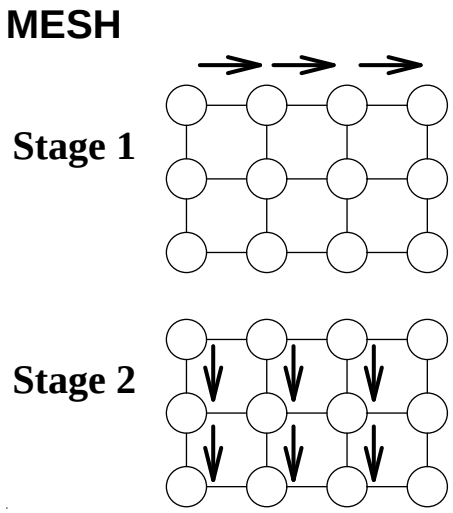


Figure 7: Broadcasting on a mesh with the store-forward model.

Fast direct node communication is allowed, then the communication cost can be further reduced for this broadcasting case. One architecture technique is called *wormhole routing* which allows pipelined fast communication between nodes even they are not directly connected. Figure 8 shows a bisection method to broadcast in a linear array under such an assumption. 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.

2.4 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 $(p - 1)(\alpha + \beta m)$.

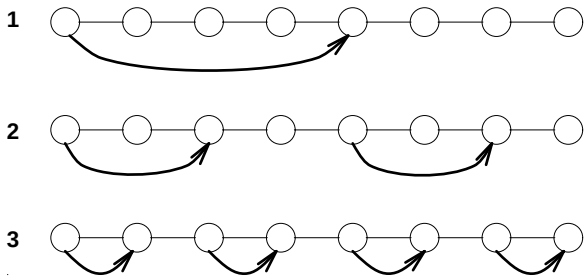


Figure 8: Broadcasting on a linear array with a wormhole model.

Figure 9 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.

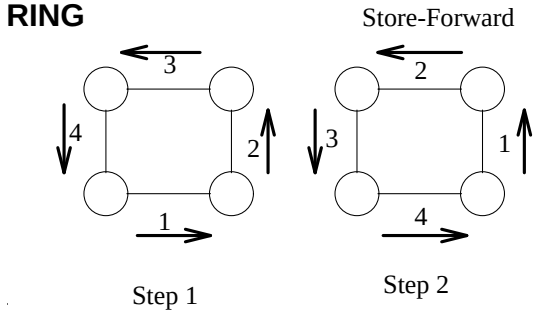


Figure 9: All-all broadcasting on a ring with the store-forward model.

2.5 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 10.

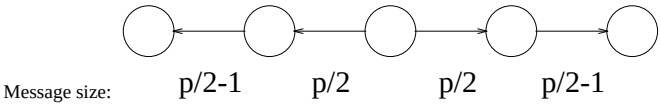


Figure 10: Personalized broadcasting on a linear with the store-forward model.

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:

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

3 Issues in Parallel Programming

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

3.1 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.

3.1.1 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:
  $S_1 : A = x + B$
  $S_2 : C = A * 3$
- $OUT(T_1) \bigcap OUT(T_2) \neq \emptyset$
  T_2 is *output-dependent* on T_1.
  $S_1 : A = x + B$
  $S_2 : A = 3$
- $IN(T_1) \bigcap OUT(T_2) \neq \emptyset$
  T_2 is *anti-dependent* on T_1.
  $S_1 : B = A + 3$
  $S_2 : A = 3$

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 11.

3.1.2 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. 12.

3.2 Program Partitioning

Purpose:

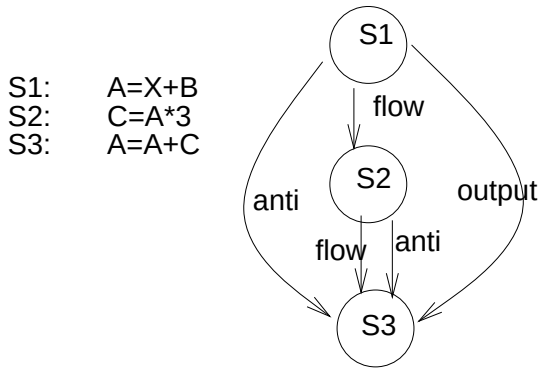


Figure 11: An example of a dependence graph. Functions f, g, h do not modify their input arguments.

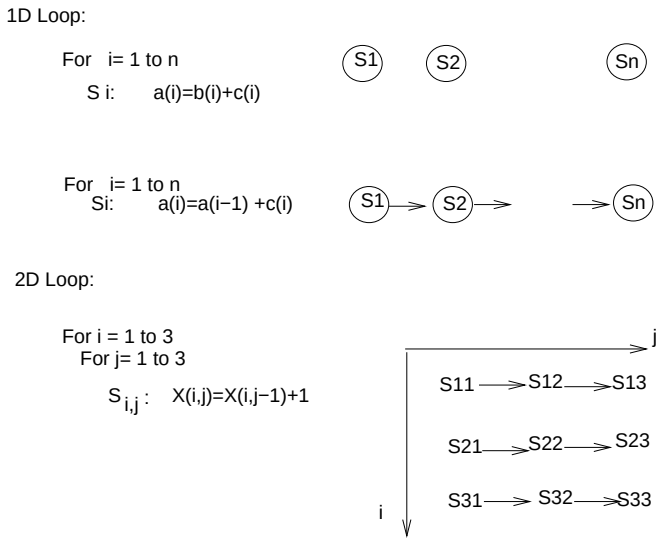


Figure 12: An example of a loop dependence graph (loop iteration space).

- 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.

3.2.1 Loop blocking/unrolling

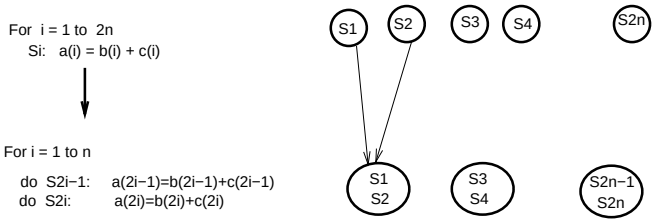


Figure 13: An example of 1D loop blocking/unrolling.

An example is Fig. 13. In general, sequential code:

```
For i = 1 to r*p
      S_i : 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)
```

3.2.2 Interior loop blocking

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

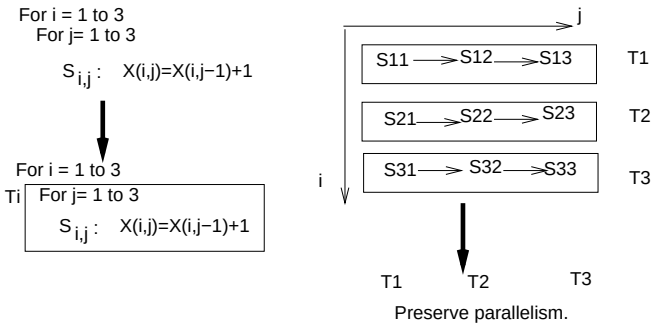


Figure 14: An example of interior loop blocking.

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. 14. Fig. 15 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.

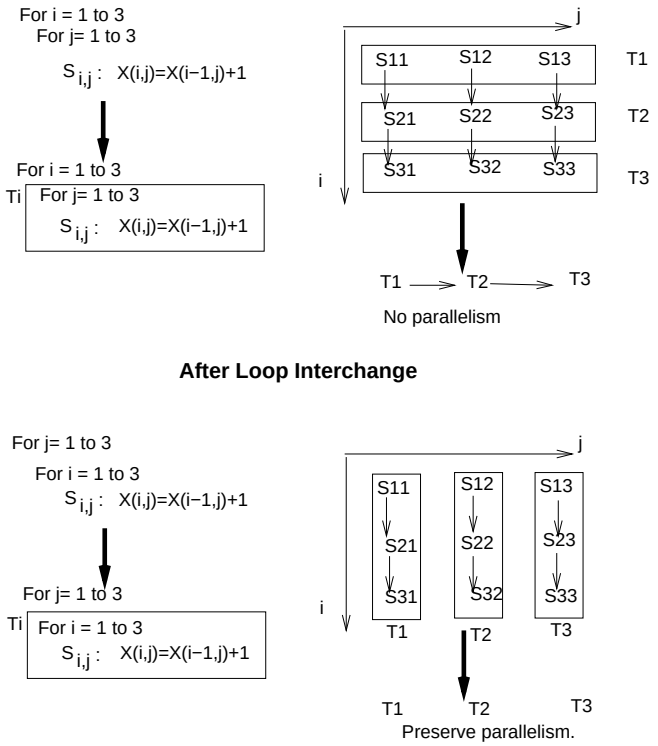


Figure 15: An example of partitioning that reduces parallelism and the role of loop interchange.

3.2.3 Loop interchange

Loop interchange is a program transformation that changes the execution order of a loop program as shown in Fig. 16. Loop interchange is not legal if the new execution order violates data dependence. An example of illegal interchange is in Fig. 17.

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
```

3.3 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.

3.3.1 Data partitioning methods

1D array $\longrightarrow$ 1D processors.

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. 18.

- *1D block*. Data i is mapped to processor $\lfloor \frac{i}{r} \rfloor$.

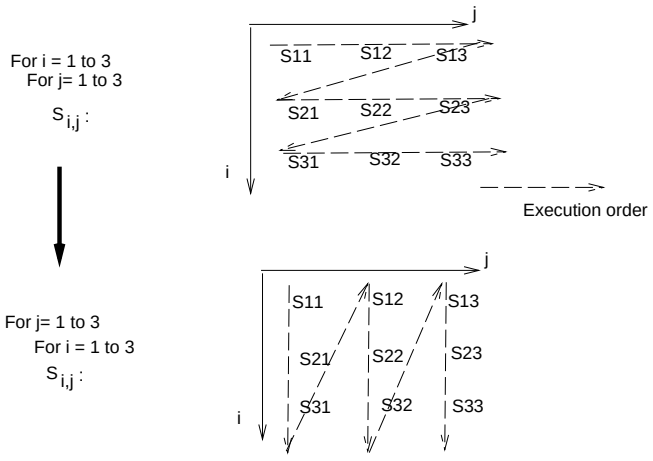


Figure 16: Loop interchange re-orders execution.

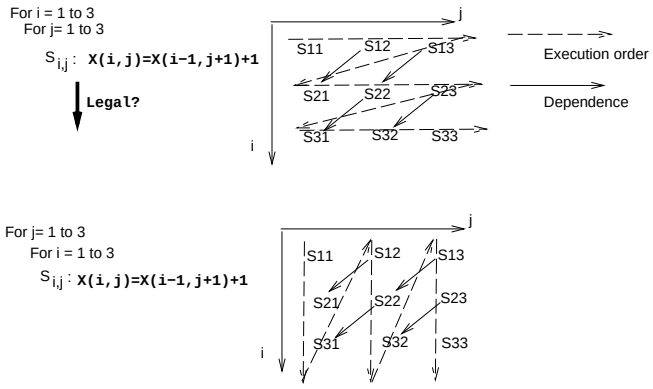
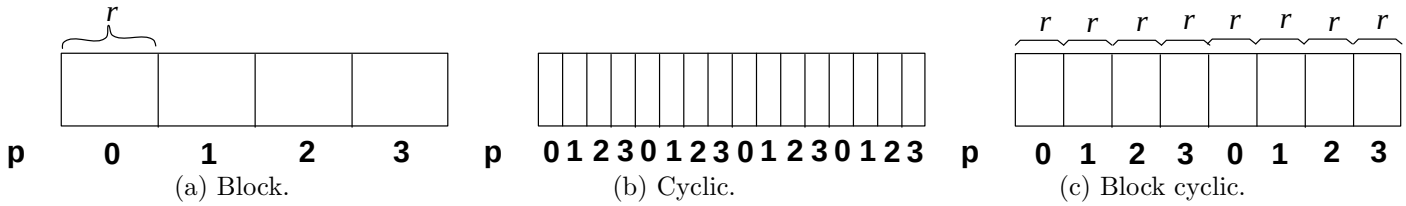


Figure 17: An illegal loop interchange.



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

Figure 18: Data partitioning schemes.

- *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 $0 \leq i, j \leq \cdots n-1$. 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 $\lfloor \frac{j}{r} \rfloor$.

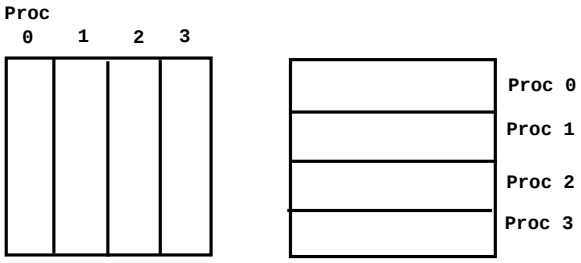


Figure 19: The left is the column-wise block and the right is the row-wise block.

- *Row-wise cyclic* Data (i, j) is mapped to processor $i \bmod p$.
- *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 $0 \leq i, j \leq \cdots n-1$. Processors are numbered as (s, t) where $0 \leq s, t \leq \cdots q-1$ 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 ($i \bmod q$, $j \bmod q$).

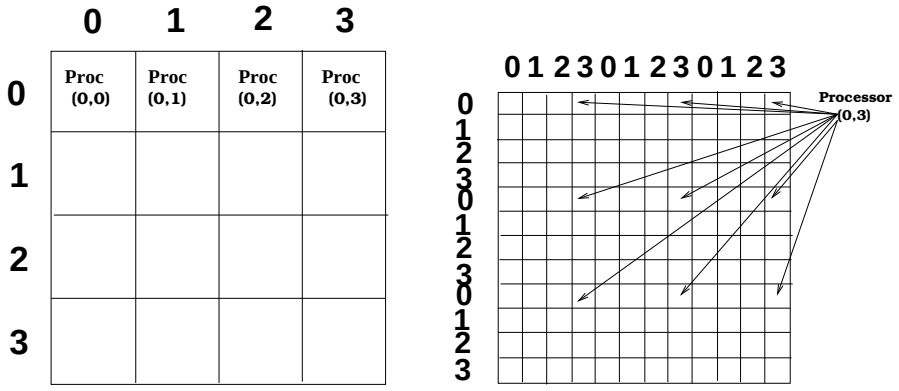


Figure 20: The left is the 2D block mapping (block,block) and the right is the 2D cyclic mapping (cyclic,cyclic).

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

3.3.2 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, $proc\_map(i) = i \bmod p$.

3.3.3 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.21. 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, $local(i) = \lfloor \frac{i}{p} \rfloor$.

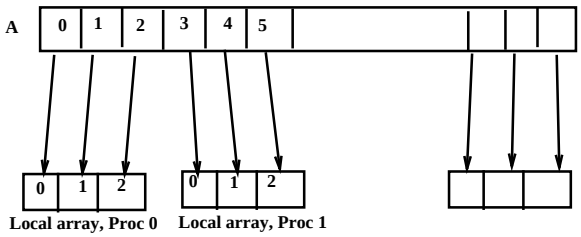


Figure 21: Global data vs local data index.

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

3.4 A summary on program parallelization

The process of program parallelization is depicted in Figure 22 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.

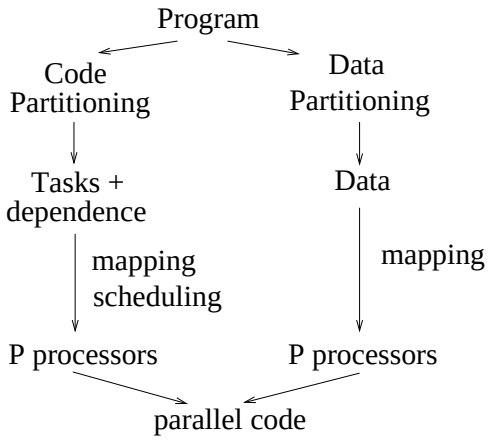


Figure 22: The process of program parallelization.

- Iterative methods for solving a linear equation. e.g. Jacobi.
- Finite-difference methods for differential equations.

4 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. 23.

Task schedule on p processors is shown in Fig. 23.

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.

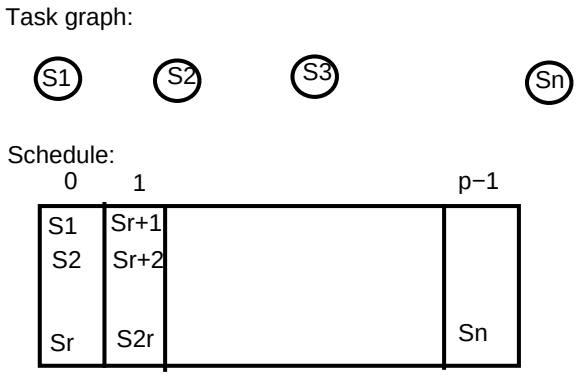


Figure 23: Task graph and a schedule for matrix vector multiplication.

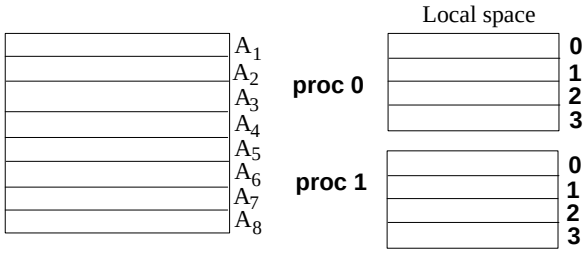


Figure 24: An illustration of data mapping for matrix-vector multiplication.

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 $PT = \frac{2n^2\omega}{p}$.

5 Matrix-Matrix Multiplication

5.1 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.

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

5.2 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 $(1 \le i \le n)$ 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
```

5.3 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:
$T_{1,1}$ $T_{1,2}$ $\cdots T_{1,n}$
$T_{2,1}$ $T_{2,2}$ $\cdots T_{2,n}$
$\cdots$
$T_{n,1}$ $T_{n,2}$ $\cdots T_{n,n}$

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: $T_{1,1}$ $T_{1,2}$ $\cdots T_{1,n}$
Cluster 2: $T_{2,1}$ $T_{2,2}$ $\cdots T_{2,n}$
$\cdots$
Cluster n: $T_{n,1}$ $T_{n,2}$ $\cdots T_{n,n}$

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 $(\alpha + \beta n) \log p$.

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

5.3.1 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), $t = (k + i) \bmod q + 1$.
  At each row, $A_{i,t}$ at processor (i, t) is broadcasted to other processors in the same row (i, j) where $1 \leq j \leq q$.
  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.

6 Gaussian Elimination for Solving Linear Systems

6.1 Gaussian Elimination without Partial Pivoting

6.1.1 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.

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

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.

6.1.2 The row-oriented parallel algorithm

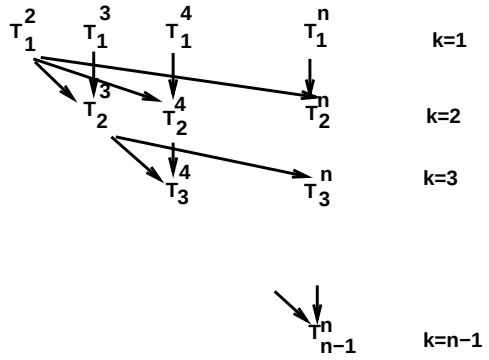
A program partitioning for the forward elimination part:

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.

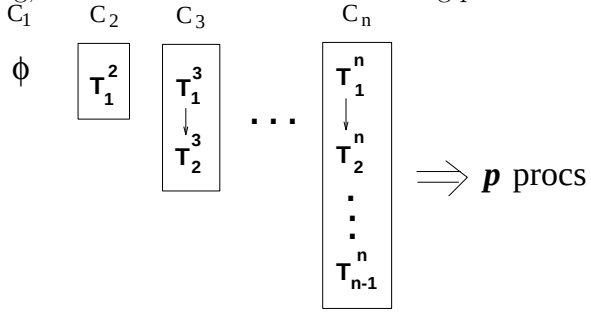
Dependence Graph.

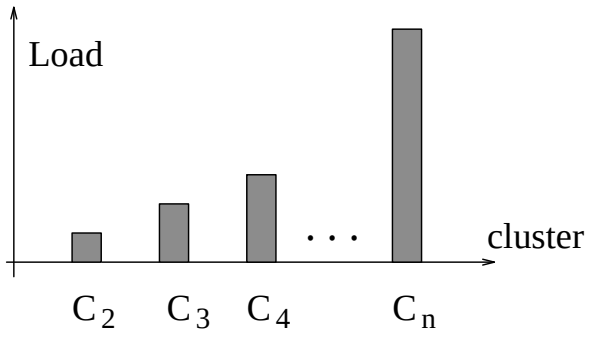


Thus for each step k, tasks $T_k^{k+1}\ T_k^{k+2}\ \ldots\ T_k^n$ are independent. We can have the following algorithm design for the row-oriented GE algorithm:

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

6.1.3 The column-oriented algorithm

The column-oriented algorithm essentially interchanges loops i and j of the GE program in Section 6.1.1.

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,

$$\begin{pmatrix} 4 & -9 & 2 & 2 \\ 2 & -4 & 4 & 3 \\ -1 & 2 & 2 & 1 \end{pmatrix} \overset{\substack{(2)=(2)-(1)*\frac{2}{4} \\ (3)=(3)-(1)*\frac{-1}{4}}}{\Longrightarrow} \begin{pmatrix} 4 & -9 & 2 & 2 \\ 0 & \frac{1}{2} & 3 & 2 \\ 0 & \frac{-1}{4} & \frac{5}{2} & \frac{3}{2} \end{pmatrix}$$

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** $n - 1$

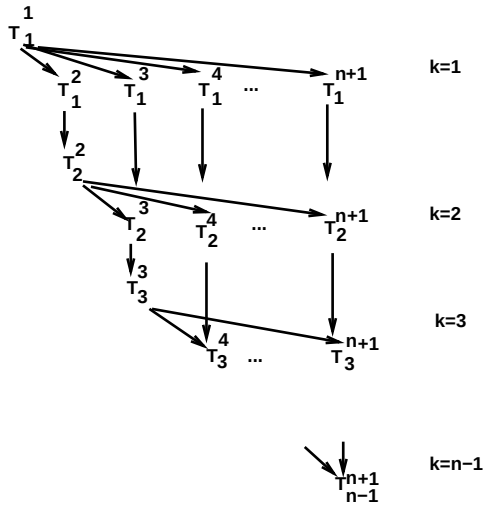
T_k^k : **For** $i = k + 1$ **to** n
  $a_{ik} = a_{ik} / a_{kk}$
Endfor

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.
```

7 Gaussian elimination with partial pivoting

7.1 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:

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

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

The backward substitution does not need any change.

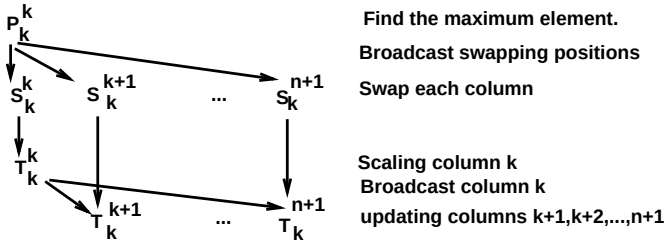
Row-oriented forward elimination: **For** $k = 1$ **to** $n-1$
Find m such that $|a_{m,k}| = \max_{n \geq i \geq k}\{|a_{i,k}|\}$;
If $a_{m,k} = 0$, No unique solution, stop;
Swap row(k) with row(m);
For $i = k+1$ **to** n
$a_{ik} = a_{ik}/a_{kk}$;
For $j = k+1$ to $n+1$
$a_{i,j} = a_{i,j} - a_{i,k}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

7.2 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

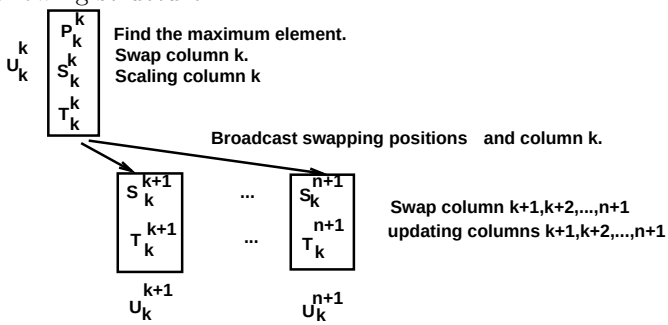
Dependence structure for iteration k: The above partitioning produces the following dependence structure.



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 $(k+1 \le j \le n+1)$.

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

8 Iterative Methods for Solving Ax = b

8.1 Iterative methods

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

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

We reformulate it as:

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

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:

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

8.2 Norms and Convergence

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 $x = (x_1, x_2, \cdots x_n)$:

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

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:

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

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

Convergence of iterative methods

Definition: 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 $\| \cdot \|$ 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.

8.3 Jacobi Method for Ax = b

For each iteration:

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

Example:

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

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.

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

8.4 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.

8.5 Gauss-Seidel Method

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

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

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

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

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.

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

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

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

Example:

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

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

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

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

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.

8.6 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: $x^{k+1} = x^k + w(x^{k+1} - x^k)$.

9 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.

9.1 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:
$$f(x) = \frac{f(x+h) - f(x-h)}{2h} + O(\frac{h^2}{3!}).$$

9.2 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:

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

9.3 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.

10 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.

10.1 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

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

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

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

$$\vdots$$

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

$\Longrightarrow$

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

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

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

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.

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

10.2 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:

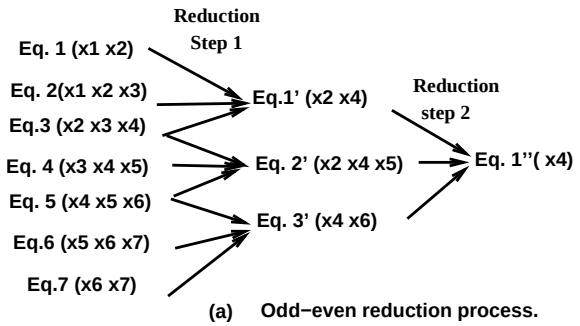
$$\begin{array}{lllll} (1) & a_1x_1 + b_1x_2 & = & d_1 & \\ (2) & c_2x_1 + a_2x_2 + b_2x_3 & = & d_2 & \\ (3) & c_3x_2 + a_3x_3 + b_3x_4 & = & d_3 & \end{array}$$

$$\Longrightarrow a'_2x_2 + b'_2x_4 = d'_2.$$

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

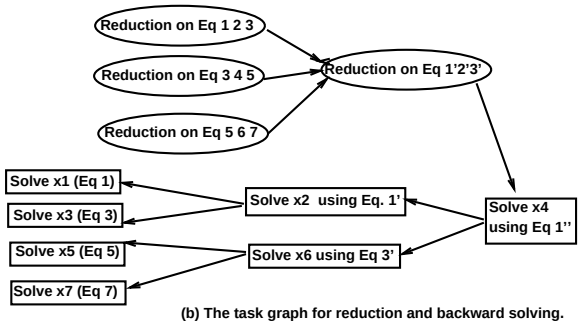
$$\Longrightarrow c'_3x_2 + a'_4x_4 + b'_5x_6 = d_4.$$

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.



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:



10.3 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 25. What are the steady-state temperatures at interior points?

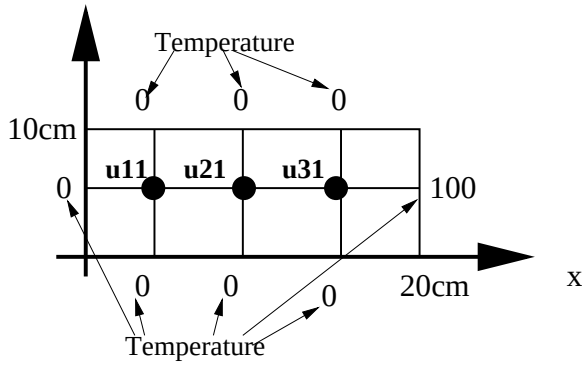


Figure 25: The problem domain for a Laplace equation in modeling the steady-state temperature.

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

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

This is called **5-point stencil computation** since 5 points are involved within each equation.

Example. For the case in Figure 25, Let $u_{11} = x_1, u_{21} = x_2, u_{31} = x_3$.

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

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

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

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

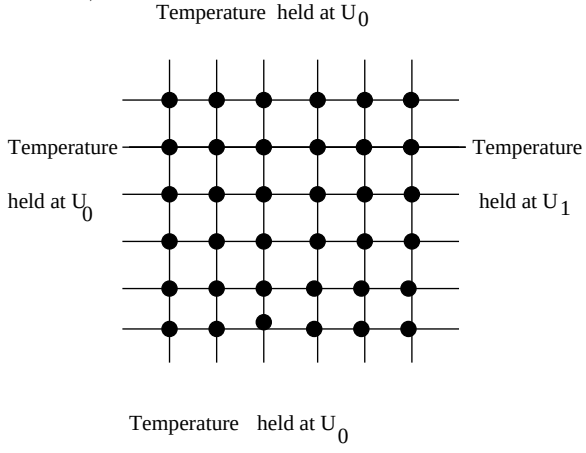
Solutions:

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

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

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

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 $p_{i+1,j}$, $p_{i-1,j}$, $p_{i,j+1}$, $p_{i,j-1}$.
$u_{i,j}^{new} = 0.25(u_{i+1,j} + u_{i-1,j} + u_{i,j+1} + u_{i,j-1})$.
$diff_{i,j} = |u_{ij}^{new} - u_{ij}|$
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 $\omega = 0.5, \beta = 0.1, \alpha = 100, n = 500, p^2 = 2500$.

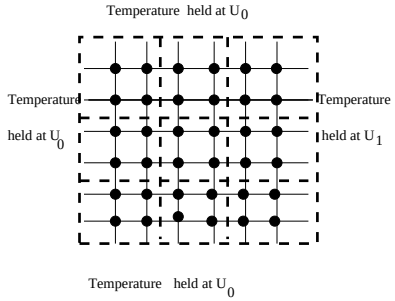
The parallel time $PT = K(\omega + (4 + \log n)(\alpha + \beta))$.

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

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

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 $b_j\gamma$
    $u_{i,j}^{new} = 0.25(u_{i+1,j} + u_{i-1,j} + u_{i,j+1} + u_{i,j-1})$.
   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 $b_j\gamma$
   $u_{i,j}^{new} = 0.25(u_{i+1,j} + u_{i-1,j} + u_{i,j+1} + u_{i,j-1})$.
  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 $\omega = 0.5, \beta = 0.1,\ \alpha = 100,\ n = 500, \omega = 100, p^2 = 25$.

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

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

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