CSE5351: Parallel Processing

Part V
Algorithmic Paradigms

- Phase Parallel
- Synchronous and Asynchronous Iteration
- Divide and Conquer
- Pipeline
- Process Farm
- Work Pool
- Example Algorithm
Phase Parallel

- The phase-parallel model offers a paradigm that is widely used in parallel programming.
- The parallel program consists of a number of supersteps, and each superstep has two phases.
- In a computation phase, multiple processes each perform an independent computation $C$. 

![Diagram of phase-parallel model]

- Synchronous Interaction
- $C$
- $C$ ...
- $C$
In the subsequent interaction phase, the processes perform one or more synchronous interaction operations, such as a barrier or a blocking communication. Then the next superstep is executed.

This paradigm is also known as the *loosely synchronous paradigm*.

Two main shortcomings: Interaction is not overlapped with computation, and it is difficult to maintain balanced workload among the processes.
Synchronous and Asynchronous Iteration

- A special case of the phase-parallel paradigm is the *synchronous iteration paradigm*, where the supersteps are a sequence of iterations in a loop.

- The following code computes iteratively the vector function $x = f(x)$, where $x$ is an $n$-dimensional vector:

```c
parfor (i = 0; i < n; i++)
{
    for (j = 0; j < N; j++) {
        x[i] = f_j(x);
    }
    barrier;
}
```
Asynchronous Iteration

- In contrast, the *asynchronous iteration paradigm* allows a process to proceed to the next iteration, without waiting for the remaining processes to catch up.
Example

parfor ( i := 0; i < n; i++) {
    for ( j := 0; j < N; j++)
        x[i] = f_i(x);
        barrier;
}
Divide and Conquer

- The *parallel divide-and-conquer paradigm* is very similar to its sequential counterpart.

- A parent process divides its workload into several smaller pieces and assigns them to a number of child processes.
• The child processes then compute their workload in parallel and the results are merged by the parent.
• The dividing and the merging procedures are done recursively.
• This paradigm is very natural for computations such as quick sort.
• Its disadvantage is the difficulty in achieving load balance.
Pipeline

- With the *pipeline paradigm*, a number of processes form a virtual pipeline.
- A continuous data stream is fed into the pipeline, and the processes execute at different pipeline stages simultaneously in an overlapped fashion.
Process Farm

- This paradigm is also known as the *master-slave* paradigm.

- A master process executes the essentially sequential part of the parallel program and spawns a number of slave processes to execute the parallel workload.

- When a slave finishes its workload, it informs the master which assigns a new workload to the slave.

- This is a simple paradigm. A disadvantage is that the master could become the bottleneck.
Work Pool

- A number of processes are created. Initially, there may be just one piece of work in the pool.
- Any free process fetches a piece of work from the pool and executes it, producing zero, one, or more new work pieces to put into the pool.
- The parallel program ends when the work pool becomes empty.
- This paradigm facilitates load balancing, as the workload is dynamically allocated to free processes.
- The pool may be an unordered set, a queue, or a priority queue.
Software Development for Message-Passing Systems

The program development process for a distributed memory (message-passing) system consists of the following steps:

- Algorithm design
- Data and operation partitioning
- Granularity determination
- Load balancing
- Minimization of inter processor communication
Algorithm design

- First the programmer must “think parallel” about the problem and develop an efficient algorithm.

- A serial algorithm may or may not be parallelizable.

- Developing a parallel algorithm may require a new problem formulation.
Data and operation partitioning

- In data partitioning approach, first data is partitioned and then related operations are assigned to PEs.

- In the operation partitioning, operations are partitioned first and then corresponding data is allocated to PEs.

- Operation partitioning is more complex but is suitable for problems with irregular structure.

- Ideal situation: problem must be partitioned into as many independent tasks as possible trivially parallel.

- Basic Rule of Partitioning: Partitioning must be done along dependencies without cutting the dependencies.
Granularity determination

- With the selected partitioning style, the grain size of each partitioning must be determined.

- Usually, fine-grained partitions have more parallelism than coarse-grained partitions but have more dependencies.

Load balancing

- Load balancing must be done to reduce the waiting times of PEs when some other PEs are still busy.

- Load balancing may be required for both operations and data.
Minimization of inter processor communication

- The basic method is to allocate part of the data to the local memory of the PE that consumes it.

- If the data item is shared, then it must be transferred through the interconnection network. To reduce communication, it must be allocated to the PE that uses it most often.

- With heavy communication, PEs may get suspended and contention on the network may occur.

- Even when network traffic is not so heavy, “hot-spots” may occur if many PEs try to access a set of data in the same memory module.
Efficiency Loss

For multiprocessor system, the speed up is defined as:

\[ S = \frac{T_S}{T_P} \]

where \( T_S \) is the execution time of the best sequential program running on a single PE and \( T_P \) is the execution time of the parallel program. The efficiency is defined as

\[ \mu = \frac{S}{N} \]

where \( N \) is the number of PEs. The value of \( \mu \) is usually less than one. The efficiency loss may come from parallel algorithm, coding, and PE suspension.
Efficiency Loss From Algorithm Parallelization

- Although sequential algorithm may be parallelized directly, programmers may design new parallel algorithms.

- A parallel algorithm may require more overall computation to solve a given problem than a sequential algorithm. The efficiency loss from algorithm parallelization is defined as

\[
EL_A = \frac{T_A}{T_S}
\]

where \(T_A\) is the execution time of the parallel algorithm coded as a sequential program without any communication primitives.

- If the same algorithm can be used as sequential or parallel, the \(EL_A\) is equal to 1. However, most parallel algorithms are less efficient.
Efficiency Loss From Coding

- When a parallel algorithm is coded into a parallel program, overhead is introduced which causes efficiency loss. This overhead includes:
  - communication overhead
  - PE initializations
  - selection statements
  - duplication of operations

- Efficiency loss from coding is defined as

\[
EL_C = \frac{\sum_{i=1}^{N} T_R}{T_A}
\]

where \(T_R\) is the running time (not including suspension time) of PE \(i\) during execution. \(EL_C\) usually increases with the number of PEs.
Efficiency Loss From Coding (cont’d)

- Communication overhead includes time spent on message packing, initialization of message transfer and the actual communication time.

- There are two kinds of message packing

- Packing several elements of the same data type that are not stored contiguously. For example, the diagonal elements of a matrix

- Packing elements of the different data type. For example

F: Force; P: Position; T: Temperature
Efficiency Loss From Coding (cont’d)

- PE initialization includes getting identification parameters, setting topologies, and opening communication channels.

- Selection statements are used to select different code segments in each PE. For example, each of the four code segments may or may not be executed on a particular PE.
Efficiency Loss From Coding (cont’d)

- To reduce communication overhead and suspension time, some operations may be duplicated on different PEs. For example, consider the following two program segments.

\[
\begin{align*}
  a &= b \times c \\
  \text{for } i = 1 \text{ to } 4 \text{ do } x[i] &= y[i] \times a
\end{align*}
\]

- One solution is to execute part I on one PE and broadcast the results to all PEs. Second solution is to duplicate the execution of I on each PE.
Efficiency Loss From Processor Suspension

- This efficiency loss is defined as

\[
EL_P = \frac{\sum_{i=1}^{N} T_{P_i}}{N} - \frac{\sum_{i=1}^{N} T_{R_i}}{N}
\]

where \( T_{P_i} \) is the execution time of the parallel program on \( N \) PEs., and is given as

\[
T_P = T_R + T_B
\]

or \((i = 1, 2, \ldots, N)\), where \( T_{B_i} \) is the total suspension time for PE \( i \).
\[ EL_P = 1 + \frac{\sum_{i=1}^{N} T_{B_i}}{N} + \frac{\sum_{i=1}^{N} T_{R_i}}{N} \]
Efficiency Loss From Processor Suspension (cont’d)

- Thus the efficiency loss due to processors suspension depends on the ratio of the total suspension time and total running time for all PEs.

- Processor suspension can occur due to load imbalance. When the load is not balanced, a lightly loaded processor will become suspended.
Efficiency Loss From Processor Suspension (cont’d)

- Even when all PEs are equally loaded, processor suspension can still occur due to message dependencies.

- Complete load balancing means that all PEs are running without any suspension.
Overall Efficiency

The overall efficiency can be expressed as

$$
\mu = \frac{1}{E_L A E_L C E_L P}
$$

The speedup can be expressed as

$$
S = \frac{N}{E_L A E_L C E_L P}
$$

The speedup for the same algorithm is

$$
S_a = \frac{N}{E_L C E_L P}
$$

The speedup for the same code is

$$
S_c = \frac{N}{E_L P}
$$
A System of Linear Equations

- Many scientific and engineering problems involve the use of a large system of linear equations.

- A linear equation in the $n$ variables $x_1, x_2, x_3, \ldots, x_n$ is an equation that can be expressed as $a_1x_1 + a_2x_2 + a_3x_3 + \ldots + a_nx_n = b$ where $a_1, a_2, a_3, \ldots, a_n$ and $b$ are constants.

- A finite set of linear equations in the variables $x_1, x_2, x_3, \ldots, x_n$ is called a system of linear equations or a linear system.

- A set of numbers $s_1, s_2, s_3, \ldots, s_n$ is a solution to a system of linear equations if and only if making the substitutions $x_1 = s_1, x_2 = s_2, x_3 = s_3, \ldots, x_n = s_n$ satisfies every equation in the linear system.
A Linear System

A system of \( n \) linear equations in \( n \) variables:

\[
\begin{align*}
    a_{11}x_1 + a_{12}x_2 + a_{13}x_3 + \ldots + a_{1n}x_n &= b_1 \\
    a_{21}x_1 + a_{22}x_2 + a_{23}x_3 + \ldots + a_{2n}x_n &= b_2 \\
    a_{31}x_1 + a_{32}x_2 + a_{33}x_3 + \ldots + a_{3n}x_n &= b_3 \\
    \quad \vdots \\
    a_{n1}x_1 + a_{n2}x_2 + a_{n3}x_3 + \ldots + a_{nn}x_n &= b_n
\end{align*}
\]

- We are given a set of \( N \) equations.
- Each equation has \( N \) variables, denoted by \( x \).
- An \( a_{ij} \) is called a coefficient where \( i \) is the equation number and \( j \) is the variable number.
A Linear System (cont’d)

- A linear system is usually expressed as $Ax = b$ where $A$ is an $n$ by $n$ matrix containing the $a_{i,j}$s and $x$ and $b$ are $n$-element vectors storing $x_i$s and $b_i$s, respectively.

- Solution of the linear system depends on how nonzero elements of matrix $A$ are located in the matrix.

- In general, a sequential algorithm having time complexity of $O(n^3)$ can solve a system of linear equations.

- An $n$ by $n$ matrix $A$ is upper triangular if

  $$i > j \Rightarrow a_{ij} = 0$$

- An $n$ by $n$ matrix $A$ is lower triangular if

  $$i < j \Rightarrow a_{ij} = 0$$
Solution of Simultaneous Equations (cont’d)

The augmented matrix of the above equations is given as follows:

\[
\begin{array}{cccc}
  a_{11}x_1 & a_{12}x_2 & a_{13}x_3 & \cdots & a_{1n}x_n &= b_1 \\
  a_{21}x_1 & a_{22}x_2 & a_{23}x_3 & \cdots & a_{2n}x_n &= b_2 \\
  a_{31}x_1 & a_{32}x_2 & a_{33}x_3 & \cdots & a_{3n}x_n &= b_3 \\
  \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\
  a_{n1}x_1 & a_{n2}x_2 & a_{n3}x_3 & \cdots & a_{nn}x_n &= b_n \\
\end{array}
\]

We can solve the set of equations if we are able to convert the augmented matrix into a triangular form.

\[
\begin{array}{cccc}
  t_{11} & t_{12} & t_{13} & \cdots & t_{1n} & z_1 \\
  t_{22} & t_{23} & t_{24} & \cdots & t_{2n} & z_2 \\
  t_{33} & t_{34} & t_{35} & \cdots & t_{3n} & z_3 \\
  \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\
  t_{nn} & & & & t_{nn} & z_n \\
\end{array}
\]
An Example of Linear Equation Solver

Consider the following set of equations:

\[
\begin{align*}
2x_1 + x_2 + x_3 &= 7 \\
4x_1 + 4x_2 + 3x_3 &= 21 \\
6x_1 + 7x_2 + 4x_3 &= 32
\end{align*}
\]

Multiply the first equation by 2 and subtract it from the second equation, and also multiply the first equation by 3 and subtract it from the third equation.

\[
\begin{align*}
2x_1 + x_2 + x_3 &= 7 \\
2x_2 + x_3 &= 7 \\
4x_2 + x_3 &= 11
\end{align*}
\]

Next multiply the second equation by 2 and subtract the result from the third equation to obtain
Example (cont’d)

The process of finding the values of $x$’s is called back substitution.

There are various methods of triangularization.

\[
\begin{align*}
2x_1 + x_2 + x_3 &= 7 \\
2x_2 + x_3 &= 7 \\
-x_3 &= -3
\end{align*}
\]
Back Substitution

- Back substitution is an algorithm to solve a linear system $Ax = b$ where $A$ is upper triangular.
- The back substitution algorithm solves the system in $O(n^2)$ time.
- Suppose we want to solve the system

\[
\begin{align*}
1x_1 + 1x_2 - 1x_3 + 4x_4 &= 8 \\
-2x_2 - 3x_3 + 1x_4 &= 5 \\
2x_3 - 3x_4 &= 0 \\
2x_4 &= 4
\end{align*}
\]

- The last equation can be solved directly since it has only a single unknown. That means $x_4 = 2$
- After that, we can simplify other equations by removing their $x_4$ and by adjusting their $b$ terms.
Back Substitution (cont’d)

Now the equations look like:

\[ \begin{align*}
1x_1 + 1x_2 - 1x_3 &= 0 \\
-2x_2 + 3x_3 &= 3 \\
2x_3 &= 6 \\
2x_4 &= 4
\end{align*} \]

From the third equation, we can obtain \( x_3 = 3 \).

After we substitute the value of \( x_3 \) in the two equations above.

\[ \begin{align*}
1x_1 + 1x_2 &= 3 \\
-2x_2 &= 12 \\
2x_3 &= 6 \\
2x_4 &= 4
\end{align*} \]

From here \( x_2 = -6 \).
Back Substitution (cont’d)

- After substituting the value of $x_2$.

\[
\begin{align*}
1x_1 & = 9 \\
-2x_2 & = 12 \\
2x_3 & = 6 \\
2x_4 & = 4
\end{align*}
\]
The Algorithm for Back Substitution

Back.Substitution(sequential):

Global $n$ {size}
    $a[1..n][1..n]$
    $b[1..n]$
    $x[1..n]$

$i$ {column index}

$j$ {row index}

begin
    for $i ← n$ down to 1 do
        $x[i] ← b[i]/a[i][i]$
        for $j ← 1$ to $i − 1$ do
            $b[j] ← b[j] − x[i] * a[j][i]$
            $a[j][i] ← 0$ {optional}
        endfor
    endfor
end
The Task Graph for Sequential Algorithm

Subscript

A

x

Versions of b

0 0 0 0 7

0 0 0 0 5

0 0 0 0 3

0 0 0 0 1
The Parallel Algorithm (UMA Shared-Memory System)

Back.Substitution(sequential):

Global
n \{size\}
p \{number of processes\}
a[1..n][1..n]
b[1..n]
x[1..n]
i \{column index\}

Local
j \{process identifier\}
k \{row index\}

begin

for i ← n down to 1 do
    x[i] ← b[i]/a[i][i]
    forall P_j where i <= j <= p do
        for k ← j to i — 1 step p do
            b[k] ← b[k] — x[i] * a[k][i]
        endfor
    endforall
endfor

end
Speedup (UMA Shared-Memory System)
Gaussian Elimination with Partial Pivoting

3x₁ + 2x₂ - 1x₃ = 1
6x₁ + 6x₂ + 2x₃ = 12
3x₁ - 2x₂ + x₃ = 11

Step 1: In the leftmost column find an entry that has the largest absolute value. This is called the pivot entry.

\[
\begin{array}{cccc}
3 & 2 & -1 & 1 \\
6 & 6 & 2 & 12 \\
3 & -2 & 1 & 11 \\
\end{array}
\]

Step 2: Perform a row interchange, if necessary, to bring the pivot entry to the top of the column.
Step 3: If the pivot entry is $a$, multiply the top row by $1/a$.

\[
\begin{bmatrix}
1 & 1 & 1/3 & 2 \\
3 & 2 & -1 & 1 \\
3 & -2 & 1 & 11
\end{bmatrix}
\]

Step 4: Add suitable multiples of top row to the rows below so that in the column located in step 1, all entries below the top become zeros. (-3 times the first row of the previous matrix).
Step 5: Cover the top row in the matrix and begin again with step 1., applied to the submatrix that remains.
The first and second rows of the submatrix are interchanged.

\[
\begin{pmatrix}
1 & 1 & 1/3 & 2 \\
0 & -5 & 0 & 5 \\
0 & -1 & -2 & -5 \\
\end{pmatrix}
\]

The first row of the submatrix is multiplied by -1/5.

\[
\begin{pmatrix}
1 & 1 & 1/3 & 2 \\
0 & 1 & 0 & -1 \\
0 & -1 & -2 & -5 \\
\end{pmatrix}
\]

The first row of the submatrix is added to the second row.
The first row of the new submatrix is multiplied by \(-1/2\).

\[
\begin{array}{cccc}
1 & 1 & 1/3 & 2 \\
0 & 1 & 0 & -1 \\
0 & 0 & -2 & -6 \\
\end{array}
\]

Step 6: Solve the corresponding system of equations by back-substitution.

\[
\begin{array}{cccc}
1 & 1 & 1/3 & 2 \\
0 & 1 & 0 & -1 \\
0 & 0 & 1 & 3 \\
\end{array}
\]

\[
\begin{align*}
x_3 &= 3; \\
x_2 &= -1; \\
x_1 &= 2
\end{align*}
\]
Gaussian Elimination

- It is an algorithm for solving the linear system $Ax = b$ when $A$ has nonzero elements in arbitrary locations.

- Gaussian elimination reduces $Ax = b$ to upper triangular system $Tx = c$.

- Back substitution algorithm can be applied to the upper triangular matrix to solve for $x$.

- The Gaussian elimination algorithm consists of $N$ iterations.
Gaussian Elimination

- In iteration $i$, all elements in column $i$ for each row $j$ below row $i$ are driven to 0 by subtracting a multiple of row $i$ from row $j$.

- In other words, the nonzero elements below the diagonal in column $i$ are eliminated by replacing each row $j$, where $i + 1 \leq j \leq n$, with the sum of row $j$ and $-\frac{a_{ji}}{a_{ii}}$.
Gaussian Elimination with Partial Pivoting

- In a simple algorithm, row $i$ is the pivot row, i.e., the row used to drive to zero all nonzero elements below the diagonal in column $i$.

- But this approach is not very stable.

- A variant of Gaussian elimination is called Gaussian elimination with partial pivoting.

- In this version, in step $i$, rows $i$ through $n$ are searched for the row whose column $i$ element has the largest absolute value.

- This row is swapped with row $i$.

- Now the algorithm uses this row to reduce to zero the nonzero elements of column $i$ in rows $i+1$ through $n$. 

Serial Algorithm for Gaussian Elimination
Parallelization of Gaussian Elimination

- Assume that $n$ is multiple of $p$.
- We can develop a parallel algorithm by distributing the elements of matrix $a$ and $b$ to individual processors.
- There are various possible ways for partition the data.
  - A row-oriented decomposition
  - A column-oriented decomposition
  - A column-scattered decomposition
  - A block-oriented decomposition
A Block-Oriented Distribution of Data
The Flow of Data During Parallel Execution
SIMD Algorithms and Programming

- SIMD algorithms are called data-parallel because their parallelism comes from simultaneous operations across large sets of data, rather than from multiple threads of control.

- The algorithms use $O(N)$ processors to solve problems of size $N$.

- Each data element is mapped to one processor.

- Typical algorithms of size $N$ can be solved in $O(\log N)$ time.

- Programs are described using virtual processors rather than hardware processors.
SIMD Algorithms

- The processor array executes commands in SIMD fashion with a single stream coming from the front-end. These instructions act on multiple data items, on the order of one (or fewer) per processor.

- Most instructions are executed conditionally, that is each processor has state bits that determine which instructions the processor will execute.

- A processor whose state bit is set is said to be selected. The state bit is called context flag and the set of selected processors is called context within which instructions are executed.

- For example, the front-end may cause all odd-numbered processors to have their context flag set, and even-numbered processors to have their flags clear.
Examples of Data-Parallel Algorithms

Sum of an Array of Numbers

• The sum of \( n \) numbers can be computed in time \( O(\log n) \) by organizing the addends at the leaves of a binary tree and performing the sums at each level of the tree in parallel.

• We assume that the number of elements to be summed is an integral power of two.

• There are as many processors as elements.

• The statement for all \( k \) in parallel do \( s \) od causes all processors to execute the same instruction \( s \) in a synchronized fashion, but the variable \( k \) has a different value for each processor. \( k \) is the index of that processor within the array of processors.
Sum of an Array of Numbers

for $j := 1 \text{ to } \log n$ do
  for all $k$ in parallel do
    if $((k+1) \mod 2^j) = 0$ then
      $x[k] := x[k - 2^{j-1}] + x[k]$
    fi
  od
od

On CM-1, an optimized version of this algorithm with 64k elements takes 200 microseconds.
Sum of an Array of Numbers (cont’d)
All Partial Sums of an Array

- It is also called “sum-prefix” operation because it computes sums over all prefixes of the array.

- It appears that computing the partial sums is an inherently serial process, because one must add up the first $k$ elements before adding $k+1$.

- In the previous example, we notice that most of the processors are idle most of the time. During iteration $j$, only $n/2^j$ processors are active.

- The partial-sums algorithm replaces each $X_k$ by $\sum_{i=0}^{k} x_i$ that is sum of all the elements preceding it and including $x_k$. 
All Partial Sums of an Array

for $j := 1$ to $\log n$ do

    for all $k$ in parallel do

        if ($(k \geq 2^{j-1})$) then

            $x[k] := x[k - 2^{j-1}] + x[k]$

        fi

    od

od

- During step $j$, $n - 2^{j-1}$ processors are busy.
- After step $j$, element number $k$ has become $\sum_{i}^{k} x_i$ where $a$

$$a = \max(0, k - 2^j + 1)$$
Sum of an Array of Numbers (cont’d)

\[
\begin{array}{cccccccc}
  & x_0 & x_1 & x_2 & x_3 & x_4 & x_5 & x_6 & x_7 \\
\hline
x_0 & \sum_{i=0}^{1} X_i & \sum_{i=1}^{2} X_i & \sum_{i=2}^{3} X_i & \sum_{i=3}^{4} X_i & \sum_{i=4}^{5} X_i & \sum_{i=5}^{6} X_i & \sum_{i=6}^{7} X_i \\
\hline
x_0 & \sum_{i=0}^{1} X_i & \sum_{i=0}^{2} X_i & \sum_{i=0}^{3} X_i & \sum_{i=1}^{4} X_i & \sum_{i=2}^{5} X_i & \sum_{i=3}^{6} X_i & \sum_{i=4}^{7} X_i \\
\hline
x_0 & \sum_{i=0}^{1} X_i & \sum_{i=0}^{2} X_i & \sum_{i=0}^{3} X_i & \sum_{i=0}^{4} X_i & \sum_{i=0}^{5} X_i & \sum_{i=0}^{6} X_i & \sum_{i=0}^{7} X_i \\
\end{array}
\]
Counting and Enumerating Active Processors

**Count:** How many processors are active in a given context

**Enumerate:** Assign a distinct integer to each active processor

- Both operations are implemented using sum and sum-prefix algorithms.
- count is implemented by having each processor examine its context flag and compute the integer 1 if the flag is set and 0 if the flag is clear. This is done by the unconditional operation.
- Then we perform the sum operation.
- Every processor computes integer 1 or 0 in the same manner, but then we perform an unconditional sum-prefix.
Counting and Enumerating Active Processors

- We then perform conditional operation such that all the selected processors have received distinct integers and deselected processors are simply ignored.

- Then every selected processor subtracts 1 from its received value. Therefore, if there are $n$ selected processors, then they will receive integers from 0 to $n - 1$ instead of 1 to $n$. 
Radix Sort

- Each key (unsigned integers) has a binary representation.
- maxint is the value of the largest representable key.
- We first examine the least significant bit of each key.
- Next we examine the next higher significant bit and so on.
- We examine the most significant bit last.
Radix Sort

for $j := 1$ to $1 + \lceil \log \maxint \rceil$ do
    for all $k$ in parallel do
        if the bit with weight $2^{j-1}$ is zero then
            $y[k] := \text{enumerate}$
            $c := \text{count}$
        fi
        if the bit with weight $2^{j-1}$ is one then
            $y[k] := \text{enumerate} + c$
        fi
        $x[y[k]] := x[k]$
    od
od
Finding the end of a Linked List

- We are given a linked list, in which each cell has a next pointer that points to the next cell in the list, while the last cell has the special value `null` in its next component.

- Each cell has an additional pointer called `chum` that may be used for temporary purposes.

- Each processor sets its `chum` component equal to its next component.

- Each processor then repeatedly replaces its `chum` by its `chum’s chum`.

- If a processor’s `chum` is `null`, then it remains `null`. 
Finding the end of a Linked List

for all $k$ in parallel do

$\text{chum}[k] := \text{next}[k]$

while $\text{chum}[k] \neq \text{null}$

and $\text{chum}[\text{chum}[k]] \neq \text{null}$ do

$\text{chum}[k] := \text{chum}[\text{chum}[k]]$

od

od
Finding the End of a Linked List
All Partial Sums of a Linked List

for all \( k \) in parallel do

\[ \text{chum}[k] := \text{next}[k] \]

while \( \text{chum}[k] \neq \text{null} \) do

\[ \text{value}[\text{chum}[k]] := \text{value}[k] + \text{value}[\text{chum}[k]] \]

\[ \text{chum}[k] := \text{chum}[\text{chum}[k]] \]

od

od
All Partial Sums of a Linked List (cont’d)
Matching UP Elements of Two Linked Lists

- This algorithm takes into account two linked lists.
- It matches up the corresponding elements in two linked lists.
- The algorithm assigns to each list cell a pointer to its “friend” in the other list.
- Again the algorithm takes logarithmic time.
Matching UP Elements of Two Linked Lists

for all \( k \) in parallel do
  friend[\( k \)] := null
od

friend[list1] := list2
friend[list2] := list1

for all \( k \) in parallel
  chum[\( k \)] := next[\( k \)]
while chum[\( k \)] != null do
  if friend[\( k \)] != null then
    friend[chum[\( k \)] := chum[friend[\( k \)]]
    chum[\( k \)] := chum[chum[\( k \)]]
  fi
od
Matching UP Components of Two Lists (cont’d)
Matching UP Components of Two Lists (cont’d)
SIMD Algorithm for Matrix Multiplication

- We are given a matrix $A = (a_{ij})$ and $B = (b_{ij})$.
- The result of multiplying $A$ and $B$ is a matrix $C$.
- We assume the SIMD network is a 2D torus with bidirectional links.
- Assume $A$ and $B$ are 64 by 64 and the network is 8 by 8.
- The matrix $A$, $B$ and $C$ are divided into 8 by 8 element sub blocks.
## Distribution of Matrices

<table>
<thead>
<tr>
<th>( \mathbf{A} )</th>
<th>( \mathbf{B} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( A_{00} )</td>
<td>( B_{00} )</td>
</tr>
<tr>
<td>( A_{01} )</td>
<td>( B_{01} )</td>
</tr>
<tr>
<td>( A_{02} )</td>
<td>( B_{02} )</td>
</tr>
<tr>
<td>( A_{03} )</td>
<td>( B_{03} )</td>
</tr>
<tr>
<td>( A_{04} )</td>
<td>( B_{04} )</td>
</tr>
<tr>
<td>( A_{05} )</td>
<td>( B_{05} )</td>
</tr>
<tr>
<td>( A_{06} )</td>
<td>( B_{06} )</td>
</tr>
<tr>
<td>( A_{07} )</td>
<td>( B_{07} )</td>
</tr>
<tr>
<td>( A_{10} )</td>
<td>( B_{10} )</td>
</tr>
<tr>
<td>( A_{11} )</td>
<td>( B_{11} )</td>
</tr>
<tr>
<td>( A_{12} )</td>
<td>( B_{12} )</td>
</tr>
<tr>
<td>( A_{13} )</td>
<td>( B_{13} )</td>
</tr>
<tr>
<td>( A_{14} )</td>
<td>( B_{14} )</td>
</tr>
<tr>
<td>( A_{15} )</td>
<td>( B_{15} )</td>
</tr>
<tr>
<td>( A_{16} )</td>
<td>( B_{16} )</td>
</tr>
<tr>
<td>( A_{17} )</td>
<td>( B_{17} )</td>
</tr>
<tr>
<td>( A_{20} )</td>
<td>( B_{20} )</td>
</tr>
<tr>
<td>( A_{21} )</td>
<td>( B_{21} )</td>
</tr>
<tr>
<td>( A_{22} )</td>
<td>( B_{22} )</td>
</tr>
<tr>
<td>( A_{23} )</td>
<td>( B_{23} )</td>
</tr>
<tr>
<td>( A_{24} )</td>
<td>( B_{24} )</td>
</tr>
<tr>
<td>( A_{25} )</td>
<td>( B_{25} )</td>
</tr>
<tr>
<td>( A_{26} )</td>
<td>( B_{26} )</td>
</tr>
<tr>
<td>( A_{27} )</td>
<td>( B_{27} )</td>
</tr>
<tr>
<td>( A_{30} )</td>
<td>( B_{30} )</td>
</tr>
<tr>
<td>( A_{31} )</td>
<td>( B_{31} )</td>
</tr>
<tr>
<td>( A_{32} )</td>
<td>( B_{32} )</td>
</tr>
<tr>
<td>( A_{33} )</td>
<td>( B_{33} )</td>
</tr>
<tr>
<td>( A_{34} )</td>
<td>( B_{34} )</td>
</tr>
<tr>
<td>( A_{35} )</td>
<td>( B_{35} )</td>
</tr>
<tr>
<td>( A_{36} )</td>
<td>( B_{36} )</td>
</tr>
<tr>
<td>( A_{37} )</td>
<td>( B_{37} )</td>
</tr>
<tr>
<td>( A_{40} )</td>
<td>( B_{40} )</td>
</tr>
<tr>
<td>( A_{41} )</td>
<td>( B_{41} )</td>
</tr>
<tr>
<td>( A_{42} )</td>
<td>( B_{42} )</td>
</tr>
<tr>
<td>( A_{43} )</td>
<td>( B_{43} )</td>
</tr>
<tr>
<td>( A_{44} )</td>
<td>( B_{44} )</td>
</tr>
<tr>
<td>( A_{45} )</td>
<td>( B_{45} )</td>
</tr>
<tr>
<td>( A_{46} )</td>
<td>( B_{46} )</td>
</tr>
<tr>
<td>( A_{47} )</td>
<td>( B_{47} )</td>
</tr>
<tr>
<td>( A_{50} )</td>
<td>( B_{50} )</td>
</tr>
<tr>
<td>( A_{51} )</td>
<td>( B_{51} )</td>
</tr>
<tr>
<td>( A_{52} )</td>
<td>( B_{52} )</td>
</tr>
<tr>
<td>( A_{53} )</td>
<td>( B_{53} )</td>
</tr>
<tr>
<td>( A_{54} )</td>
<td>( B_{54} )</td>
</tr>
<tr>
<td>( A_{55} )</td>
<td>( B_{55} )</td>
</tr>
<tr>
<td>( A_{56} )</td>
<td>( B_{56} )</td>
</tr>
<tr>
<td>( A_{57} )</td>
<td>( B_{57} )</td>
</tr>
<tr>
<td>( A_{60} )</td>
<td>( B_{60} )</td>
</tr>
<tr>
<td>( A_{61} )</td>
<td>( B_{61} )</td>
</tr>
<tr>
<td>( A_{62} )</td>
<td>( B_{62} )</td>
</tr>
<tr>
<td>( A_{63} )</td>
<td>( B_{63} )</td>
</tr>
<tr>
<td>( A_{64} )</td>
<td>( B_{64} )</td>
</tr>
<tr>
<td>( A_{65} )</td>
<td>( B_{65} )</td>
</tr>
<tr>
<td>( A_{66} )</td>
<td>( B_{66} )</td>
</tr>
<tr>
<td>( A_{67} )</td>
<td>( B_{67} )</td>
</tr>
<tr>
<td>( A_{70} )</td>
<td>( B_{70} )</td>
</tr>
<tr>
<td>( A_{71} )</td>
<td>( B_{71} )</td>
</tr>
<tr>
<td>( A_{72} )</td>
<td>( B_{72} )</td>
</tr>
<tr>
<td>( A_{73} )</td>
<td>( B_{73} )</td>
</tr>
<tr>
<td>( A_{74} )</td>
<td>( B_{74} )</td>
</tr>
<tr>
<td>( A_{75} )</td>
<td>( B_{75} )</td>
</tr>
<tr>
<td>( A_{76} )</td>
<td>( B_{76} )</td>
</tr>
<tr>
<td>( A_{77} )</td>
<td>( B_{77} )</td>
</tr>
</tbody>
</table>
## Distribution of Matrix $A$

<table>
<thead>
<tr>
<th></th>
<th>$A_{00}$</th>
<th>$A_{01}$</th>
<th>$A_{02}$</th>
<th>$A_{03}$</th>
<th>$A_{04}$</th>
<th>$A_{05}$</th>
<th>$A_{06}$</th>
<th>$A_{07}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$A_{11}$</td>
<td>$A_{12}$</td>
<td>$A_{13}$</td>
<td>$A_{14}$</td>
<td>$A_{15}$</td>
<td>$A_{16}$</td>
<td>$A_{17}$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$A_{22}$</td>
<td>$A_{23}$</td>
<td>$A_{24}$</td>
<td>$A_{25}$</td>
<td>$A_{26}$</td>
<td>$A_{27}$</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$A_{33}$</td>
<td>$A_{34}$</td>
<td>$A_{35}$</td>
<td>$A_{36}$</td>
<td>$A_{37}$</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$A_{44}$</td>
<td>$A_{45}$</td>
<td>$A_{46}$</td>
<td>$A_{47}$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$A_{55}$</td>
<td>$A_{56}$</td>
<td>$A_{57}$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$A_{66}$</td>
<td>$A_{67}$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$A_{77}$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Distribution of Matrix B
### Distribution of Matrix B

<table>
<thead>
<tr>
<th>$B_{00}$</th>
<th>$B_{11}$</th>
<th>$B_{22}$</th>
<th>$B_{33}$</th>
<th>$B_{44}$</th>
<th>$B_{55}$</th>
<th>$B_{66}$</th>
<th>$B_{77}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$B_{10}$</td>
<td>$B_{21}$</td>
<td>$B_{32}$</td>
<td>$B_{43}$</td>
<td>$B_{54}$</td>
<td>$B_{65}$</td>
<td>$B_{76}$</td>
<td>$B_{07}$</td>
</tr>
<tr>
<td>$B_{20}$</td>
<td>$B_{31}$</td>
<td>$B_{42}$</td>
<td>$B_{53}$</td>
<td>$B_{64}$</td>
<td>$B_{75}$</td>
<td>$B_{06}$</td>
<td>$B_{17}$</td>
</tr>
<tr>
<td>$B_{30}$</td>
<td>$B_{41}$</td>
<td>$B_{52}$</td>
<td>$B_{63}$</td>
<td>$B_{74}$</td>
<td>$B_{05}$</td>
<td>$B_{16}$</td>
<td>$B_{27}$</td>
</tr>
<tr>
<td>$B_{40}$</td>
<td>$B_{51}$</td>
<td>$B_{62}$</td>
<td>$B_{73}$</td>
<td>$B_{04}$</td>
<td>$B_{15}$</td>
<td>$B_{26}$</td>
<td>$B_{37}$</td>
</tr>
<tr>
<td>$B_{50}$</td>
<td>$B_{61}$</td>
<td>$B_{72}$</td>
<td>$B_{03}$</td>
<td>$B_{14}$</td>
<td>$B_{25}$</td>
<td>$B_{36}$</td>
<td>$B_{47}$</td>
</tr>
<tr>
<td>$B_{60}$</td>
<td>$B_{71}$</td>
<td>$B_{02}$</td>
<td>$B_{13}$</td>
<td>$B_{24}$</td>
<td>$B_{35}$</td>
<td>$B_{46}$</td>
<td>$B_{57}$</td>
</tr>
<tr>
<td>$B_{70}$</td>
<td>$B_{01}$</td>
<td>$B_{12}$</td>
<td>$B_{23}$</td>
<td>$B_{34}$</td>
<td>$B_{45}$</td>
<td>$B_{56}$</td>
<td>$B_{67}$</td>
</tr>
</tbody>
</table>
## Distribution of Matrix C

<table>
<thead>
<tr>
<th>C_{00}</th>
<th>C_{01}</th>
<th>C_{02}</th>
<th>C_{03}</th>
<th>C_{04}</th>
<th>C_{05}</th>
<th>C_{06}</th>
<th>C_{07}</th>
</tr>
</thead>
<tbody>
<tr>
<td>C_{10}</td>
<td>C_{11}</td>
<td>C_{12}</td>
<td>C_{13}</td>
<td>C_{14}</td>
<td>C_{15}</td>
<td>C_{16}</td>
<td>C_{17}</td>
</tr>
<tr>
<td>C_{20}</td>
<td>C_{21}</td>
<td>C_{22}</td>
<td>C_{23}</td>
<td>C_{24}</td>
<td>C_{25}</td>
<td>C_{26}</td>
<td>C_{27}</td>
</tr>
<tr>
<td>C_{30}</td>
<td>C_{31}</td>
<td>C_{32}</td>
<td>C_{33}</td>
<td>C_{34}</td>
<td>C_{35}</td>
<td>C_{36}</td>
<td>C_{37}</td>
</tr>
<tr>
<td>C_{40}</td>
<td>C_{41}</td>
<td>C_{42}</td>
<td>C_{43}</td>
<td>C_{44}</td>
<td>C_{45}</td>
<td>C_{46}</td>
<td>C_{47}</td>
</tr>
<tr>
<td>C_{50}</td>
<td>C_{51}</td>
<td>C_{52}</td>
<td>C_{53}</td>
<td>C_{54}</td>
<td>C_{55}</td>
<td>C_{56}</td>
<td>C_{57}</td>
</tr>
<tr>
<td>C_{60}</td>
<td>C_{61}</td>
<td>C_{62}</td>
<td>C_{63}</td>
<td>C_{64}</td>
<td>C_{65}</td>
<td>C_{66}</td>
<td>C_{67}</td>
</tr>
<tr>
<td>C_{70}</td>
<td>C_{71}</td>
<td>C_{72}</td>
<td>C_{73}</td>
<td>C_{74}</td>
<td>C_{75}</td>
<td>C_{76}</td>
<td>C_{77}</td>
</tr>
</tbody>
</table>
The Algorithm

For $i = 0$ to $7$ Do

(1) Compute the Product of blocks of $A$ and $B$
(2) Add this product to block of $C$
(3) Pass block of $A$ to the left neighbor in a wraparound fashion
(4) Pass block of $B$ to the upper neighbor in a wraparound fashion
End Do

In step (1) of the $i$th iteration, $PE_{k,l}$ performs the following computation.

$$C_{k,l} = C_{k,l} + A_{k,(i+j) \mod 8} \times B_{(i+r) \mod 8, l}$$

initial column index of $A$  initial row index of $A$
### First Iteration of Multiplication

<table>
<thead>
<tr>
<th>A_{00} B_{00}</th>
<th>A_{01} B_{11}</th>
<th>A_{02} B_{22}</th>
<th>A_{03} B_{33}</th>
<th>A_{04} B_{44}</th>
<th>A_{05} B_{55}</th>
<th>A_{06} B_{66}</th>
<th>A_{07} B_{77}</th>
</tr>
</thead>
<tbody>
<tr>
<td>A_{11} B_{10}</td>
<td>A_{12} B_{21}</td>
<td>A_{13} B_{32}</td>
<td>A_{14} B_{43}</td>
<td>A_{15} B_{54}</td>
<td>A_{16} B_{65}</td>
<td>A_{17} B_{76}</td>
<td>A_{10} B_{07}</td>
</tr>
<tr>
<td>A_{22} B_{20}</td>
<td>A_{23} B_{31}</td>
<td>A_{24} B_{42}</td>
<td>A_{25} B_{53}</td>
<td>A_{26} B_{64}</td>
<td>A_{27} B_{75}</td>
<td>A_{20} B_{06}</td>
<td>A_{21} B_{17}</td>
</tr>
<tr>
<td>A_{33} B_{30}</td>
<td>A_{34} B_{41}</td>
<td>A_{35} B_{52}</td>
<td>A_{36} B_{63}</td>
<td>A_{37} B_{74}</td>
<td>A_{30} B_{05}</td>
<td>A_{31} B_{16}</td>
<td>A_{32} B_{27}</td>
</tr>
<tr>
<td>A_{44} B_{40}</td>
<td>A_{45} B_{51}</td>
<td>A_{46} B_{62}</td>
<td>A_{47} B_{73}</td>
<td>A_{40} B_{04}</td>
<td>A_{41} B_{15}</td>
<td>A_{42} B_{26}</td>
<td>A_{43} B_{37}</td>
</tr>
<tr>
<td>A_{55} B_{50}</td>
<td>A_{56} B_{61}</td>
<td>A_{57} B_{72}</td>
<td>A_{50} B_{03}</td>
<td>A_{51} B_{14}</td>
<td>A_{52} B_{25}</td>
<td>A_{53} B_{36}</td>
<td>A_{54} B_{47}</td>
</tr>
<tr>
<td>A_{66} B_{60}</td>
<td>A_{67} B_{71}</td>
<td>A_{60} B_{02}</td>
<td>A_{61} B_{13}</td>
<td>A_{62} B_{24}</td>
<td>A_{63} B_{35}</td>
<td>A_{64} B_{46}</td>
<td>A_{65} B_{57}</td>
</tr>
<tr>
<td>A_{77} B_{70}</td>
<td>A_{70} B_{01}</td>
<td>A_{71} B_{12}</td>
<td>A_{72} B_{23}</td>
<td>A_{73} B_{34}</td>
<td>A_{74} B_{45}</td>
<td>A_{75} B_{56}</td>
<td>A_{76} B_{67}</td>
</tr>
</tbody>
</table>
### Second Iteration of the Multiplication

|   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
| A01 | B10 | A02 | B21 | A03 | B32 | A04 | B43 | A05 | B54 | A06 | B65 | A07 | B76 | A00 | B07 |
| A12 | B20 | A13 | B31 | A14 | B42 | A15 | B53 | A16 | B64 | A17 | B75 | A10 | B06 | A11 | B17 |
| A23 | B30 | A24 | B41 | A25 | B52 | A26 | B63 | A27 | B74 | A20 | B05 | A21 | B16 | A22 | B27 |
| A34 | B40 | A35 | B51 | A36 | B62 | A37 | B73 | A30 | B04 | A31 | B15 | A32 | B26 | A33 | B37 |
| A45 | B50 | A46 | B61 | A47 | B72 | A40 | B03 | A41 | B14 | A42 | B25 | A43 | B36 | A44 | B47 |
| A56 | B60 | A57 | B71 | A50 | B02 | A51 | B13 | A52 | B24 | A53 | B35 | A54 | B46 | A55 | B57 |
| A67 | B70 | A60 | B01 | A61 | B12 | A62 | B23 | A63 | B34 | A64 | B45 | A65 | B56 | A66 | B67 |
| A70 | B00 | A71 | B11 | A72 | B22 | A73 | B33 | A74 | B44 | A75 | B55 | A76 | B66 | A77 | B77 |


Analysis

- In each PE, there is an 8 by 8 matrix block.
- In step 1, the multiplication of the blocks in each PE requires: \(8^3\) multiplication and \(8^3\) additions
- In step 2, the shift (left communication) operation requires: \(8^2\) shifts
- In step 3, the shift (upper communication) operation requires: \(8^2\) shifts
- So the total number of operations in each iteration is: 
  \[2 \times 8^3 + 2 \times 8^2 = 1152\]
- The total number of operations in 8 iterations is: \(8 \times 1152 - 128 = 9088\).