CSCE 5160 Parallel Processing

Exam 2: Wednesday April 3, 2019 in F280

Problem 1

Iteration 1: $V_T = \{a\}$, $d[] = \begin{pmatrix} a & \; b & \; c & \; d & \; e \\ a & 0 & 2 & 5 & 5 & \text{inf} \\ b & 2 & 0 & \text{inf} & \text{inf} & 2 \\ c & \text{inf} & 0 & \text{inf} & 1 \\ d & 5 & \text{inf} & \text{inf} & 0 & 2 \\ e & \text{inf} & 2 & 1 & 2 & 0 \end{pmatrix}$

Iteration 2: $V_T = \{a, b\}$, $d[] = \begin{pmatrix} a & \; b & \; c & \; d & \; e \\ a & 0 & 2 & 5 & 5 & \text{inf} \\ b & 2 & 0 & \text{inf} & \text{inf} & 2 \\ c & \text{inf} & 0 & \text{inf} & 1 \\ d & 5 & \text{inf} & \text{inf} & 0 & 2 \\ e & \text{inf} & 2 & 1 & 2 & 0 \end{pmatrix}$

Iteration 3: $V_T = \{a, b, e\}$, $d[] = \begin{pmatrix} a & \; b & \; c & \; d & \; e \\ a & 0 & 2 & 5 & 5 & \text{inf} \\ b & 2 & 0 & \text{inf} & \text{inf} & 2 \\ c & \text{inf} & 0 & \text{inf} & 1 \\ d & 5 & \text{inf} & \text{inf} & 0 & 2 \\ e & \text{inf} & 2 & 1 & 2 & 0 \end{pmatrix}$

Iteration 4: $V_T = \{a, b, e, c\}$, $d[] = \begin{pmatrix} a & \; b & \; c & \; d & \; e \\ a & 0 & 2 & 5 & 5 & \text{inf} \\ b & 2 & 0 & \text{inf} & \text{inf} & 2 \\ c & \text{inf} & 0 & \text{inf} & 1 \\ d & 5 & \text{inf} & \text{inf} & 0 & 2 \\ e & \text{inf} & 2 & 1 & 2 & 0 \end{pmatrix}$

Iteration 5: $V_T = \{a, b, e, c, d\}$, $d[] = \begin{pmatrix} a & \; b & \; c & \; d & \; e \\ a & 0 & 2 & 5 & 5 & \text{inf} \\ b & 2 & 0 & \text{inf} & \text{inf} & 2 \\ c & \text{inf} & 0 & \text{inf} & 1 \\ d & 5 & \text{inf} & \text{inf} & 0 & 2 \\ e & \text{inf} & 2 & 1 & 2 & 0 \end{pmatrix}$
Problem 2

Need to construct Bitonic sequence first.

\{12, 2, 11, 4, 9, 1, 10, 15, 5, 7, 14, 3, 8, 13, 6, 16\} =
\{2, 12, 11, 4, 1, 9, 15, 10, 5, 7, 14, 3, 8, 13, 6, 16\} =
\{2, 4, 11, 12, 15, 10, 9, 1, 3, 5, 7, 14, 16, 13, 8, 6\} =
\{1, 2, 4, 9, 10, 11, 12, 15, 10, 14, 13, 8, 7, 6, 5, 3\}

Now to create a sorted list
\{1, 2, 4, 9, 10, 11, 12, 15, 16, 14, 13, 8, 7, 6, 5, 3\} =

\[
\begin{array}{cccc}
1 & 1 & 1 & 1 \\
2 & 2 & 2 & 2 \\
4 & 4 & 4 & 3 \\
9 & 8 & 3 & 4 \\
10 & 7 & 7 & 5 \\
11 & 6 & 6 & 6 \\
12 & 5 & 5 & 7 \\
13 & 3 & 8 & 8 \\
14 & 16 & 10 & 9 \\
15 & 14 & 11 & 10 \\
16 & 13 & 12 & 11 \\
1 & 9 & 9 & 11 \\
2 & 10 & 16 & 13 \\
3 & 11 & 14 & 14 \\
4 & 12 & 13 & 16 \\
5 & 13 & 15 & 16 \\
\end{array}
\]

Problem 3 a

a). We have two alternatives

Alternative 1:
Step 1: Each processor receives \(n/p\) elements. Places its inputs into \(p\) buckets.
Step 2: Each processor sorts one bucket

Pseudo code:
- Scatter \(n/p\) elements
- Each processor locally places \(n/p\) elements into \(p\) buckets
- Each processor gathers one bucket
- We need use GatherV
- Sort a bucket
- Gathers at root

Alternative 2: All processors receive \(n\) elements
Each processor selects inputs belonging to one bucket
Sort each bucket

Pseudo code:
- Broadcast \(n\) elements to all processors
- Each processor selects numbers in one bucket
- Sort
CSCE 5160 Parallel Processing

Problem 3 b MPI code for Alt 1

Alt 1:

…. {standard MPI INIT and variable declaration etc
Sendbuffer contains n input values
Rcvbuffer contains data received
Send_count = receive_count = n/p)

MPI_Scatter (&sendbuffer, send_count, MPI_INT, &rcvbuffer,
receive_count, MPI_INT, 0, MPI_COMM_WORLD)

Sort_inputs_into_p_buckets;

/* since each processor may have different number of elements in buckets
/* processor p needs to collect how many each processor has and create two arrays
/* needed for GatherV

for (i=0; i<e; i++){ for (i=0; i<p; i++) {
MPI_Gather(....); } /* gather how many values each processor
Create_arrays_for_gather; /* processor i creates its two arrays
for (i=0; i<e; i++) {
MPI_Gatherv(....); } /* now processor i will gather bucket i
/* final gather at root

/* need to find out how many elements there are in each bucket
/* create two arrays at root
/* use GatherV

MPI_Gather(....); /* gather how many values there are in each bucket
Create_arrays_for_gather;

MPI_Gatherv(....);
Problem 3 b MPI code for Alt 2

```c
MPI_Bcast(....); // all n values to all p processors
Select_values_for_bucket_i; /* processor i will select values for bucket */
Sort_bucket_i; /* final gather at root */
/* need to find out how many elements there are in each bucket */
/* create two arrays at root */
/* use Gather V */
MPI_Gather(....); // gather how many values there are in each bucket
Create_arrays_for_gather;
MPI_Gatherv(......);
```

Problem 3 c Complexity analysis for Alt 1

Step 1: Scatter n/p values

\[ t_c \left( \frac{n}{p} \right) + t_s (\log p) + t_w (n/p)(p-1) \]

Step 2: Each processor places n/p elements in p buckets

\[ t_c (n/p) \]

Step 3: We repeat this step of collecting how many elements in each bucket

\[ p \left( t_s \log p + t_w \left( \frac{n}{p} \right)(p-1) \right) \] -- only one value sent by each processor

Step 4: We repeat this step also p times

MPI_GatherV --worst case each processor may have n/p in a bucket

\[ p \left( t_c \log p + t_w \left( \frac{n}{p} \right)(p-1) \right) \]

Step 5: Each processor sorts its bucket, each bucket may have all n values

\[ t_c (n \log n) \]

Step 6: Need to gather how many from each processor

\[ t_c \log p + t_w (p-1) \]

Step 7: GatherV

\[ t_s \log p + t_w (n/p)(p-1) \]

Total execution time: Computation cost = \[ t_c \left( \frac{n}{p} \right) + t_c (n \log n) \]

Communication cost: \[ = t_s \log p + t_w \left( \frac{n}{p} \right)(p-1) + p^2(t_s \log p + t_w (p-1)) + p(t_c \log p + t_w \left( \frac{n}{p} \right)(p-1)) + t_c \log p + t_w \left( \frac{n}{p} \right)(p-1) \]
CSCE 5160 Parallel Processing

Problem 3 c Complexity analysis for Alt 2

Step 1: one-to-all-broadcast
\[ t_i + t_w(n)(\log p) \]

Step 2: Each processor selects items in its bucket
\[ t_i(n) \]

Step 3: Each processor sorts its bucket and each bucket may have all n elements
\[ t_i(n \log n) \]

Step 4: Need to gather how many from each processor
\[ t_i \log p + t_w(n)(p-1) \]

Step 5: GatherV
\[ (t_i \log p + t_w(n/p)(p-1)) \]

Total Execution time:
Computation time = \[ t_i(n) + t_i(n \log n) \]
Communication time = \[ t_i + t_w(n)(\log p) + t_i \log p + t_w(n)(p-1) + (t_i \log p + t_w(n/p)(p-1)) \]

Note Alternative 2 has much less communication cost

CSCE 5160 Parallel Processing

Review: Graph Algorithms

Shortest path
- Dijkstra’s algorithm
- Floyd’s algorithm

DFS and BFS trees

DFS: row striping of adjacency matrix

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>6</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>7</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>8</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>9</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

CSCE 5160 Parallel Processing
After finding individual depth first trees, we have dfs forests
Now we need to merge these into a single tree
Another important algorithm

Maximal independent set

Somewhat related to the famous 4-Color theorem

MIS is a set of vertices of a graph such that no edges among these nodes exist
The nodes in this set are not adjacent to each other
Maximal mean, if you add one more node to this set, it will no longer be independent

Consider an example Figure 10.15 on page 452

Figure 10.15  Examples of independent and maximal independent sets.
CSCE 5160 Parallel Processing

MIS are not unique
How do we find MIS (sequential algorithm first)

V is the set of all vertices
MIS is the set of independent vertices

Initialization: MIS = null
Repeat

pick a node \( u \) from V
add \( u \) to MIS and remove it from V
remove all nodes \( v \) from V which are adjacent to \( u \)

Until V is empty

Complexity = O(n)

How to parallelize?
If we add one node at a time, we do not get much parallelism

Consider a different approach so that we can add more than one node to MIS at a time
We will assign random values to each node
Pick nodes with lowest value compared to their neighbors
add these nodes to MIS and remove their neighbors from V

CSCE5160 April 8, 2019
CSCE 5160 Parallel Processing

If a graph has very few edges we will use adjacency lists (instead of adjacency matrix) for graph algorithms

- if connected, a graph must have at least n-1 edges (minimum number of edges)
- if strongly connected will have n*(n-1)/2 edges (maximum)

Parallelization of adjacency lists-based algorithms is very difficult

How do we balance load?

If we distribute vertices (and corresponding lists) to processors, some processors may have more edges than others

If we try to distribute equal number of edges, involves more communication

If we assume a special structure for graph, say that on average all nodes have the same number of edges -- grid graphs (see for example Fig 10.14 on page 452)

We will see similar grid graphs in several algorithms that use sparse matrices and sparse graphs

CSCE5160 April 8, 2019
CSCE 5160 Parallel Processing

Remember Dijkstra’s algorithm and see how to parallelize it if we are using adjacency lists.

Note that we have two key operations:
1. Find the next vertex with minimum length to be added:
   find a vertex u belonging to (V - VT) → a node outside the three such that l[u] = min { l[v] | v belongs to (V - VT)}
2. Update lengths of remaining vertices:
   for all v belonging to (V - VT) do
   \[ l[v] = \min\{ l[v], l[u] + w(u,v) \} \]
   /* update lengths, u is a node in tree
   end for

How do we modify these if we use adjacency lists?

If we assume that we always maintain a sorted list of vertices based on lengths l[v]:
   Actually we do not need a fully sorted list. We just keep a Priority Queue with min \[ l[v] \] at the top of the queue.
   this operation takes one time unit -- O (log n)
   the second operation depends on the number of edges from each vertex --- O(|E|)
   Total sequential complexity = O(|E| * log n)

Johnson’s Algorithm for shortest path:

Now consider the actual algorithm shown on page 455 (called Johnson’s algorithm):

\[
\begin{align*}
Q &= V; \\
&\text{for all } v \text{ in } Q \\
l[v] &= \text{infinity}; \\
l[s] &= 0; \quad \#\text{source node}
\end{align*}
\]
while Q is not empty
{
\begin{align*}
&\text{u = extract_smallest_from_Q;} \quad \#\text{find a node } v \text{ such } l[v] \text{ is min} \\
&\text{for } v \text{ in } \text{Adjacency_list of } u \\
&\{ \\
&\quad \text{if } v \text{ is in } Q \text{ and } l[u] + w(u,v) < l[v] \text{ then} \\
&\quad \quad l[v] = l[u] + w(u,v); \\
&\} \\
\}
\]
We keep a priority queue Q such that a where l[a] is the smallest is at the head of the queue.
Consider the following example

![Graph Image]

Initialization: $l[u]$ contains only infinite values

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
<th>e</th>
<th>f</th>
</tr>
</thead>
<tbody>
<tr>
<td>infinity</td>
<td>infinity</td>
<td>infinity</td>
<td>infinity</td>
<td>infinity</td>
<td>infinity</td>
</tr>
</tbody>
</table>

Iteration 1: select source node: a
Need to update the $l[u]$ vector

<table>
<thead>
<tr>
<th>b</th>
<th>c</th>
<th>d</th>
<th>e</th>
<th>f</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>infinity</td>
<td>infinity</td>
<td>2</td>
<td></td>
</tr>
</tbody>
</table>

Q: {b, f, c, d, e} sorted by length values

Iteration 2: select node $u$ with min $l[u]$ value (head of Q)
we will select b.
for each $v$ in adjacency list of $b$, update $l[v]$
We need to consider only c, d (since a is already on the tree)

consider c: min { $l[c] = 3; l[b] + w[b,c] = 1+5 = 6$ } no change to $l[c]$
consider d: min { $l[d] = infinity; l[b] + w[b,d] = 1+1 = 2$ } new $l[d] = 2$

Q: {d, f, c e} sorted – actually insert the new value at right place
CSCE 5160 Parallel Processing

Iteration 3: select a new node \( u \) with \( \min l[v] \)

let us select \( d \)

We need to update \( l[v] \) for all vertices \( v \) in adjacency list of \( d \)

We need only consider \( c \) and \( e \); since \( b \) is already on the tree

consider \( c \): \( \min \{ l[c] = 3; l[d] + w[d,e] = 2 + 2 = 4 \} \) no change to \( l[c] = 3 \)

consider \( e \): \( \min \{ l[e] = \infty; l[d] + w[d,e] = 2 + 4 = 6 \} \) change \( l[e] = 6 \)

\[
\begin{array}{ccc}
  c & e & f \\
  3 & 6 & 2 \\
\end{array}
\]

Q: \( \{f, c, e\} \)

Iteration 4: select a node \( u \) with \( \min l[u] \). Select \( f \)

Need to update \( l[v] \) for all \( v \) in the adjacency list of \( f \)

We need to consider \( e \)

consider \( e \): \( \min \{ l[e] = 6; l[f] + w[f,e] = 2 + 7 \} \) no change to \( l[v] \)

Q: \( \{c, e\} \)

CSCE 5160 Parallel Processing

Iteration 5: select a node \( u \) with \( \min l[u] \); select \( c \); we only need to consider updating \( l[e] \)

consider \( e \): \( \min \{ l[e] = 6; l[c] + w(c,e) = 3 + 1 = 4 \} \) the new \( l[e] = 4 \)

Shortest path lengths from node \( a \) to all other nodes are

\[
\begin{array}{cccc}
  b & c & d & e & f \\
  1 & 3 & 2 & 4 & 2 \\
\end{array}
\]

How do we parallelize the algorithm----Single Priority Queue

We have a single queue, and a single processor updates the queue;
The only parallelism is for updating \( l[v] \)

for all \( v \) belonging to \( V - V_T \) do

\[
  l[v] = \min \{ l[v], l[u] + w(u,v) \} \quad /\* \text{update lengths} \*
\]

Depending on the average number of edges, very small parallelism exits

Number of processors busy = \( O(|E| / n) \)
CSCE 5160 Parallel Processing

When implementing using MPI, we can use **gatherv** (since you may have different number of \( l \) values from each processor) to update the queue. But we do not get much speed up.

How about maintaining “distributed priority Q”? Let us consider the implementation using Distributed Q.

Each processor must maintain a portion of the Q; With \( p \) processors, we will assign \( n/p \) vertices and associated adjacency lists to each processor.

Consider 3 processors, each with 2 nodes. Note \( P_{ab} \) only knows connections originating in \( a \) or \( b \). Likewise \( P_{cd} \) and \( P_{ef} \).

Initialization is as before: all \( l(v) = \infty \).

Step 1: \( a \) is added to the tree

\( P_{ab} \) updates its \( l(v) \)

\[
\begin{array}{ccc}
 b & c & \infty & \infty & 2 \\
 1 & 3 & \infty & \infty & 2 \\
\end{array}
\]

\( P_{ab} \) sends new \( l(v) \) values to \( P_{cd} \) \( P_{ef} \) since we found a shorter path to source to \( c \) and \( f \).

And updates distances for node \( b \) locally –

At \( P_{cd} \) computes values as follows

\[ l[b] = 3 + 5 = 8; \ l[d] = 3 + 2 = 5; \ l[e] = 3 + 1 = 4 \]

And processor \( P_{ef} \) computes

\[ l[e] = 2 + 5 = 7 \]
Iteration 2:
Now $P_{a-b}$ sends the updated values to $P_{c-d}$ (if $d$ changed), while $P_{c-d}$ sends its updated values to $P_{a-b}$ and $P_{e-f}$.

And $P_{e-f}$ sends updates values to $P_{a-b}$ (not to $P_{c-d}$ since no changes to $l[c]$ or $l[d]$).

We proceed in this manner until the $l[v]$ values are no longer updated.

How would distributed Q methods using Shared memory (PRAM)?

We can use the same method as before; use a lock to update the queue and use new values.

How would distributed Q methods using Shared memory (PRAM)?

We can use the same method as before; use a lock to update the queue and use new values.

### Other MPI functions of interest – non blocking send and receive

`MPI_Isend` and `MPI_Irecv` are non blocking send and receive

```c
int MPI_Irecv (void *buffer, int cnt, MPI_Datatype dtype, int src, int tag, MPI_Comm comm, MPI_Request *handle)

int MPI_Isend (void *buffer, int cnt, MPI_Datatype dtype, int dest, int tag, MPI_Comm comm, MPI_Request *handle)
```

Pointer to object that identifies communication operation

The first 6 parameters are the same. We have a new argument called `MPI_Request`.

The handle object will receive the status of the receive (or send).

Receiver cannot immediately access buffer. Must first check the status.
CSCE 5160 Parallel Processing

You need to use MPI_Wait before accessing the receive buffer

\[
\text{int MPI_Wait (MPI_Request *handle, MPI_Status *status)}
\]

Note the process executing MPI_Wait blocks until the message communication is complete.

If receiver executes MPI_Wait, \textit{on return}, the receiver can access the buffer (in MPI_Irecv).

If sender, the sender can reuse the send buffer.

The status provides details about the communication including errors, number of elements received.

One more function

\[
\text{int MPI_Probe (int src, int tag, MPI_Comm comm, MPI_Status *status)}
\]

This function allows you check if a message with “tag” from sender “src” is received (or will be blocked) – \textit{you can use Probe or Wait}.

You can use MPI_ANY_SOURCE and MPI_ANY_TAG

CSCE5160 April 8, 2019

CSCE 5160 Parallel Processing

So far we have seen Data parallel (execute loops in parallel).

There are other models for parallelism – task parallelism.

Let us think of one way to use task parallelism:

Manager – Worker Model – now we can use non blocking send and receive to send work when a worker is idle – and requests more work.

Manager

\[\text{Assign job} \rightarrow \text{Worker} \rightarrow \text{Assign job} \rightarrow \text{Worker} \rightarrow \text{Assign job} \rightarrow \text{Manager} \rightarrow \text{Assign job} \]

Assign job

Return completed job

Return completed job

CSCE5160 April 8, 2019
CSCE 5160 Parallel Processing

An example using Document Classification or search engines – a Big Data example

Worker 0 reads the dictionary and shares it with all other workers

Workers send result vectors to manager

CSCE 5160 Parallel Processing

Killing MPI processes

MPI_Abort -- to kill another process

MPI_Abort (process_rank, error_code);

We need this to kill worker processes when there is no more work

We also need to create separate communication group
one for all workers that excludes the manager

This is useful in document classification so that workers may exchange dictionary (or in some cases discoveries or share other results)

We can create communication groups using MPI_Comm_Split

First the manager will exclude itself by passing a MPI_UNDEFINED to split_key

The communication group created will be MPI_COMM_NULL

Then the remaining processes can be in a separate group.
int id;

MPI_Comm worker_comm;
...

if (!id) /* Manager is rank 0 */
    MPI_Comm_split (MPI_COMM_WORLD, MPI_UNDEFINED,
                   id, &worker_comm);
else /* worker processes */
    MPI_Comm_split (MPI_COMM_WORLD, 0, id, &worker_comm);

We also need non blocking communication – and we have seen how to do this
For example the manager may want to receive requests from workers for more work
But we should use non-blocking and also receive messages from anyone

Can we consider this type of a model in OpenMP?
How do we separate tasks for Worker and Manager?
We know how to create identical threads using \texttt{#pragma parallel for}

\textbf{We need task parallelism where different threads execute different codes}

We can define sections of code for Manager and workers
\begin{verbatim}
#pragma omp sections [private....]
{
    #pragma omp section
    {manager}

    #pragma omp section
    {#pragma omp parallel for
        for (i=0; i<number_of_workers; i++)
        {worker(i)}
    }
}
\end{verbatim}
Need Locks

This type of synchronization in shared memory programming is known as "producer" – "consumer" synchronization.

Here we can think of a worker thread as a consumer and manager thread as a producer.

Locks in OpenMP

Need to declare lock variables of type `omp_lock_t`

Initialize the lock

```c
omp_init_lock
```

To lock, use `omp_set_lock`

And to unlock (or release the lock) use `omp_unset_lock`

```
An generic example

omp_lock_t my_lock;
omp_init_lock(&my_lock);

#pragma omp parallel private(tmp, id)
{
  id = omp_get_thread_num();
  tmp = result_of_lots_of_work(id);
  omp_set_lock(&my_lock);
  print("%d %d", id, tmp);
  omp_unset_lock(&my_lock);
}
omp_destroy_lock(&my_lock);
```

For our purpose

Manager:

```
repeat {until no more work }
{ .Lock
  .Check if a request for work exists
  .If not, Unlock
  .Check if work remains
  .Set work
  .If not, "kill worker threads" or flag that no more work
  .Unlock}
```

Worker

```
repeat {no more work}
{
  .Lock
  .set request
  .Unlock
  { .Lock
    .Get work if exists and Unlock
    .If no work, unlock and try again
    { If work, do work
  }
}
```
Yet another model for parallelism
Pipelined model of parallelism
\( P_0 \) performs some computation and sends results to \( P_1 \)
\( P_i \) computes its computation and passes results to \( P_{i+1} \)

We have seen one example for sorting
\( P_0 \) reads next input number, keeps small value
(min or new value and previously stored value)
and sends larger value to \( P_1 \)
\( P_i \) likewise keeps the smaller value and sends larger to \( P_{i+1} \)

Another Example of Pipelined parallelism can be applied to find prime numbers
Let us consider the algorithm known as the Sieve Eratosthenes

1. Create list of unmarked natural numbers 2, 3, …, \( n \)
2. \( k = 2 \) is the first in the list
3. Repeat
   (a) Mark all multiples of \( k \) between \( k^2 \) and \( n \)
   (b) set \( k \) to the smallest unmarked number > \( k \)
   until \( k^2 > n \)
4. The unmarked numbers are primes

pipelined parallelism
Each processor keeps the next prime number
And if a new number is not divisible by the prime at \( P_i \), send it to \( P_{i+1} \)

• Processor \( P_0 \) starts reading numbers, and checks if the number is divisible by 2 (the first prime number).
• If the number is not divisible, it is passed to \( P_1 \)
• \( P_1 \) will have the second prime number. \( P_i \) passes only the numbers that are not divisible to \( P_2 \) and so on.

How would you implement this using OpenMP?