Partitioning

- *Partition the problem into parts such that each part can be solved separately.*

Partitioning

- *Partition the problem into parts such that each part can be solved separately.*
- At the end the solution to the overall problem may just be the collection of the solutions to the parts or we may need to combine the solutions of the parts.

Partitioning

- *Partition the problem into parts such that each part can be solved separately.*
- At the end the solution to the overall problem may just be the collection of the solutions to the parts or we may need to combine the solutions of the parts.
- The effort here often lies in partitioning in such a way that the combining part at the end is not needed or can be simplified.

Partitioning Examples

- Parallel prefix sums
- Bottom-up parallel mergesort
- Parallel bucketsort

Parallel Prefix Sums (a.k.a Parallel Prefix)

The sequential algorithm for computing prefix sums.

```
prefix_sums(X, Sum)
//input Array: X[0...n-1]
//output Array: Sum[0...n-1]
//Sum[i] = Σ_{k=1}^{i} X[k]
1.  Sum[0] ← X[0]
2.  for (i=1; i<n; i++)
3.      Sum[i] ← Sum[i-1] + X[i]
```

Parallel Prefix

Parallel Prefix an useful building block for various parallel algorithms.
In our discussion, we will assume that the n numbers are distributed across p processes, with each process having n/p numbers.

Step I. Each process computes the prefix sums of its share of numbers using the sequential algorithm.

Parallel Prefix

Parallel Prefix an useful building block for various parallel algorithms.
In our discussion, we will assume that the n numbers are distributed across p processes, with each process having n/p numbers.

- Step I. Each process computes the prefix sums of its share of numbers using the sequential algorithm.
- Step II. Process i, $1 \leq i \leq p-1$, sends its last sum ($X[\frac{n}{p}-1]$) to Process 0.

Parallel Prefix

Parallel Prefix an useful building block for various parallel algorithms.
In our discussion, we will assume that the n numbers are distributed across p processes, with each process having n/p numbers.

- **Step I.** Each process computes the prefix sums of its share of numbers using the sequential algorithm.
- **Step II.** Process i, $1 \leq i \leq p-1$, sends its last sum ($X[\frac{n}{p}-1]$) to Process 0.
- **Step III.** Process 0 computes prefix sums of the p partial sums it receives from other processes.

Parallel Prefix

Parallel Prefix an useful building block for various parallel algorithms.
In our discussion, we will assume that the n numbers are distributed across p processes, with each process having n/p numbers.

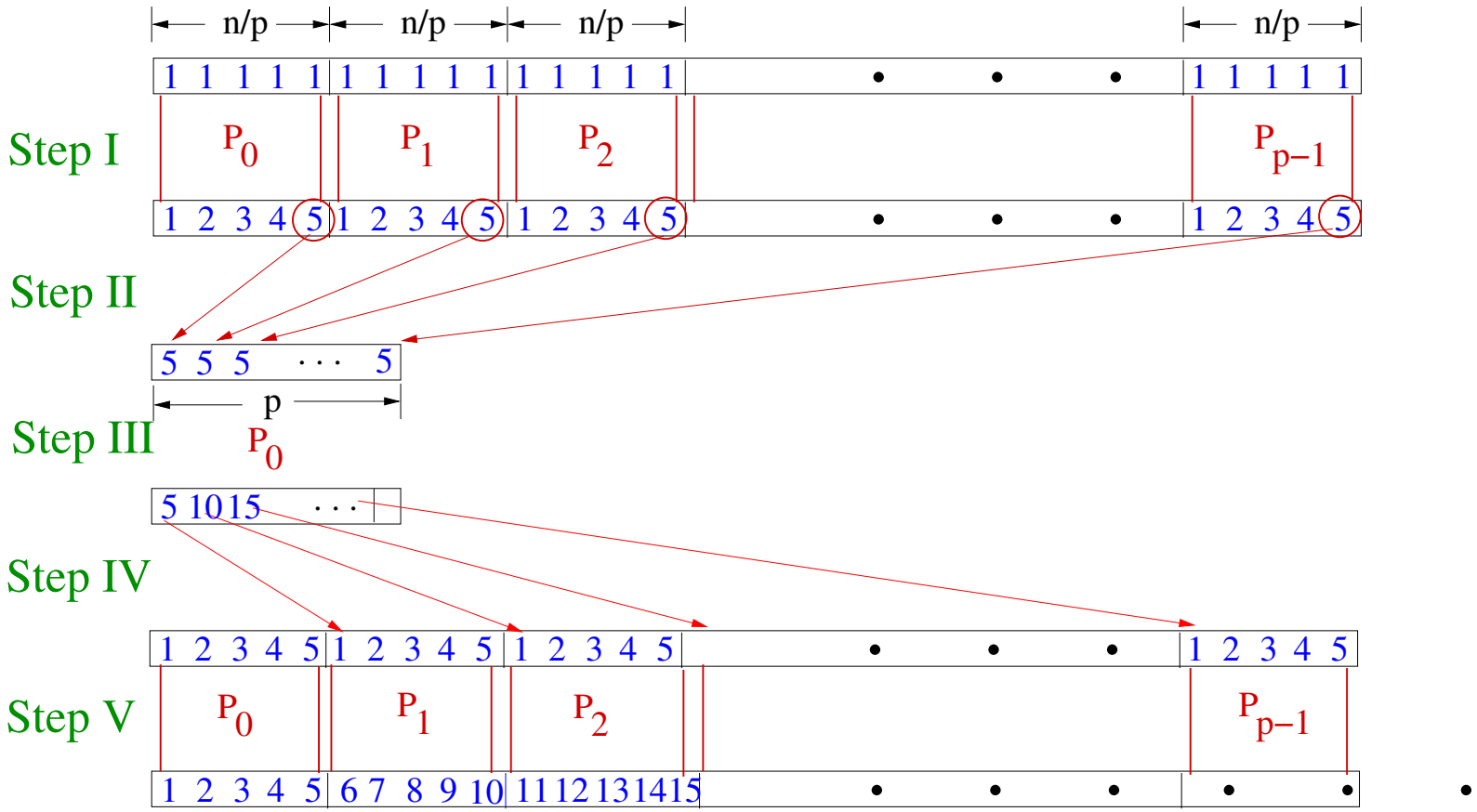
- **Step I.** Each process computes the prefix sums of its share of numbers using the sequential algorithm.
- **Step II.** Process i, $1 \leq i \leq p-1$, sends its last sum ($X[\frac{n}{p}-1]$) to Process 0.
- **Step III.** Process 0 computes prefix sums of the p partial sums it receives from other processes.
- **Step IV.** Process 0 ships the ith value from the array it just computed to Process $i+1$, $0 \leq i \leq p-2$.

Parallel Prefix

Parallel Prefix an useful building block for various parallel algorithms.
In our discussion, we will assume that the n numbers are distributed across p processes, with each process having n/p numbers.

- **Step I.** Each process computes the prefix sums of its share of numbers using the sequential algorithm.
- **Step II.** Process i, $1 \leq i \leq p-1$, sends its last sum ($X[\frac{n}{p}-1]$) to Process 0.
- **Step III.** Process 0 computes prefix sums of the p partial sums it receives from other processes.
- **Step IV.** Process 0 ships the ith value from the array it just computed to Process $i+1$, $0 \leq i \leq p-2$.
- **Step V.** Each process adds the partial value received to each element of its array of prefix sums.

Prefix Sums (a.k.a Parallel Prefix)



Analysis of Parallel Prefix

$$t_{\text{comm}} = (p-1)(t_{\text{startup}} + t_{\text{data}}) + (p-1)(t_{\text{startup}} + t_{\text{data}}) = \Theta(p)$$

Analysis of Parallel Prefix

$$
\begin{aligned}
t_{\text{comm}} &= (p-1)(t_{\text{startup}} + t_{\text{data}}) + (p-1)(t_{\text{startup}} + t_{\text{data}}) = \Theta(p) \\
t_{\text{comp}} &= \frac{n}{p} + (p-1) + \frac{n}{p} = \Theta(\frac{n}{p} + p)
\end{aligned}
$$

Analysis of Parallel Prefix

$$
\begin{aligned}
t_{\text{comm}} &= (p-1)(t_{\text{startup}} + t_{\text{data}}) + (p-1)(t_{\text{startup}} + t_{\text{data}}) = \Theta(p) \\
t_{\text{comp}} &= \frac{n}{p} + (p-1) + \frac{n}{p} = \Theta(\frac{n}{p} + p) \\
T_p(n) &= t_{\text{comm}} + t_{\text{comp}} = \Theta(\frac{n}{p} + p) \\
T^*(n) &= \Theta(n)
\end{aligned}
$$

Analysis of Parallel Prefix

$$
\begin{aligned}
t_{\text{comm}} &= (p-1)(t_{\text{startup}} + t_{\text{data}}) + (p-1)(t_{\text{startup}} + t_{\text{data}}) = \Theta(p) \\
t_{\text{comp}} &= \frac{n}{p} + (p-1) + \frac{n}{p} = \Theta(\frac{n}{p} + p) \\
T_p(n) &= t_{\text{comm}} + t_{\text{comp}} = \Theta(\frac{n}{p} + p) \\
T^*(n) &= \Theta(n) \\
\text{Speedup } S_p(n) &= \Theta\left(\frac{n}{\frac{n}{p} + p}\right) = \Theta\left(\frac{p}{\frac{p^2}{n} + 1}\right)
\end{aligned}
$$

$$
\lim_{n \to \infty} \frac{p}{\frac{p^2}{n} + 1} = p
$$

Bottom-up Mergesort

- Mergesort is normally written as a top-down recursive algorithm. However, it can also be expressed as a bottom-up iterative algorithm. The picture on the next slide demonstrates how that works.

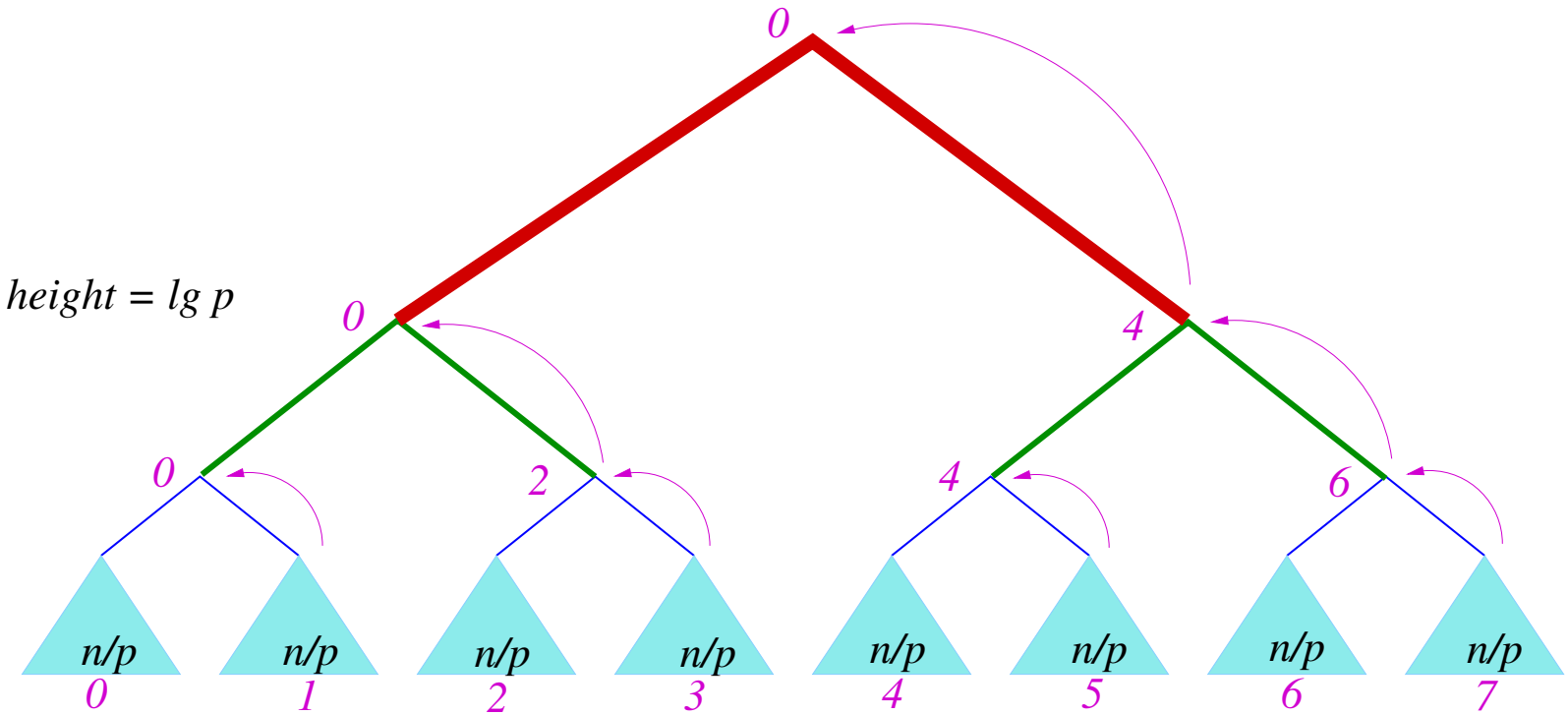
Bottom-up Mergesort

- Mergesort is normally written as a top-down recursive algorithm. However, it can also be expressed as a bottom-up iterative algorithm. The picture on the next slide demonstrates how that works.
- A bottom-up mergesort is more amenable to parallelization via partitioning. Each process starts with roughly n/p elements, where n is the number of elements to sort and p is the number of processors. Each process then sorts them independently. From there the processes merge the sections until Process 0 has the sorted whole.

Bottom-up Mergesort

- Mergesort is normally written as a top-down recursive algorithm. However, it can also be expressed as a bottom-up iterative algorithm. The picture on the next slide demonstrates how that works.
- A bottom-up mergesort is more amenable to parallelization via partitioning. Each process starts with roughly n/p elements, where n is the number of elements to sort and p is the number of processors. Each process then sorts them independently. From there the processes merge the sections until Process 0 has the sorted whole.
- *Parallel Bottom-Up Mergesort illustrates an iterative tree-like coding technique, which is a common design pattern in parallel programs*.

Bottom-up Parallel Mergesort (continued)



–The purple arrows shows the direction of data transfer
–At each level the amount of data sent from one processor doubles
–At each level only half the number of processes remain active

Parallel Mergesort Pseudocode

parallel_mergesort(A, low, high, pid)
//Processes numbered, $0 \leq pid \leq p-1$
//Assume that $p = 2^k$
//Each process has n/p numbers at the start.

1. sort n/p elements locally
2. **for** (h=1; h $\leq$ lg p; h++)
3. **if** (pid mod $2^h == 0$)
4. recv $\frac{n}{p}2^{h-1}$elements from process $pid + 2^{h-1}$
5. **else**
6. send $\frac{n}{p}2^{h-1}$ elements to process $pid - 2^{h-1}$
7. break //this process is done
8. merge $\frac{n}{p}2^{h-1}$ elements with $\frac{n}{p}2^{h-1}$ local elements

Parallel Mergesort Analysis

- At each stage a total of about n/2 elements are transferred. There are a total of $\lg p$ stages. Hence the communication time is as shown below.

$$T_{\text{comm}}(n) = \Theta(n \lg p)$$

Parallel Mergesort Analysis

- At each stage a total of about n/2 elements are transferred. There are a total of $\lg p$ stages. Hence the communication time is as shown below.

$$T_{\text{comm}}(n) = \Theta(n \lg p)$$

- The total computation time includes time to initially sort n/p elements. The at each stage, half the processes merge elements. The merging time is linear to the number of elements. So this time doubles at each stage. The total computation time is shown below.

Parallel Mergesort Analysis

- At each stage a total of about n/2 elements are transferred. There are a total of $\lg p$ stages. Hence the communication time is as shown below.

$$T_{\text{comm}}(n) = \Theta(n \lg p)$$

- The total computation time includes time to initially sort n/p elements. The at each stage, half the processes merge elements. The merging time is linear to the number of elements. So this time doubles at each stage. The total computation time is shown below.

$$\begin{aligned}
T_{\text{comp}}(n) &= \Theta(\frac{n}{p}\lg\frac{n}{p} + (\frac{n}{p} + 2\frac{n}{p} + 4\frac{n}{p} + \ldots + p\frac{n}{p})) \\
&= \Theta(\frac{n}{p}\lg\frac{n}{p} + \frac{n}{p}(1 + 2 + 4 + \ldots + p/4 + p/2 + p)) \\
&= \Theta(\frac{n}{p}\lg\frac{n}{p} + \frac{n(2p-1)}{p}) \\
&= \Theta(\frac{n}{p}\lg\frac{n}{p} + n)
\end{aligned}$$

Parallel Mergesort Analysis

- At each stage a total of about n/2 elements are transferred. There are a total of $\lg p$ stages. Hence the communication time is as shown below.

$$T_{\text{comm}}(n) = \Theta(n \lg p)$$

- The total computation time includes time to initially sort n/p elements. The at each stage, half the processes merge elements. The merging time is linear to the number of elements. So this time doubles at each stage. The total computation time is shown below.

$$\begin{aligned} T_{\text{comp}}(n) &= \Theta(\frac{n}{p}\lg\frac{n}{p} + (\frac{n}{p} + 2\frac{n}{p} + 4\frac{n}{p} + \ldots + p\frac{n}{p})) \\ &= \Theta(\frac{n}{p}\lg\frac{n}{p} + \frac{n}{p}(1+2+4+\ldots+p/4+p/2+p)) \\ &= \Theta(\frac{n}{p}\lg\frac{n}{p} + \frac{n(2p-1)}{p}) \\ &= \Theta(\frac{n}{p}\lg\frac{n}{p} + n) \end{aligned}$$

- **Notes**. Moderate to poor speedup because of large communication and because a large number of processes idle in the second phase. More sophisticated versions that use all the idle processes to make the merge operations faster in each stage give better speedups, especially on shared memory machines.

Sequential Bucketsort

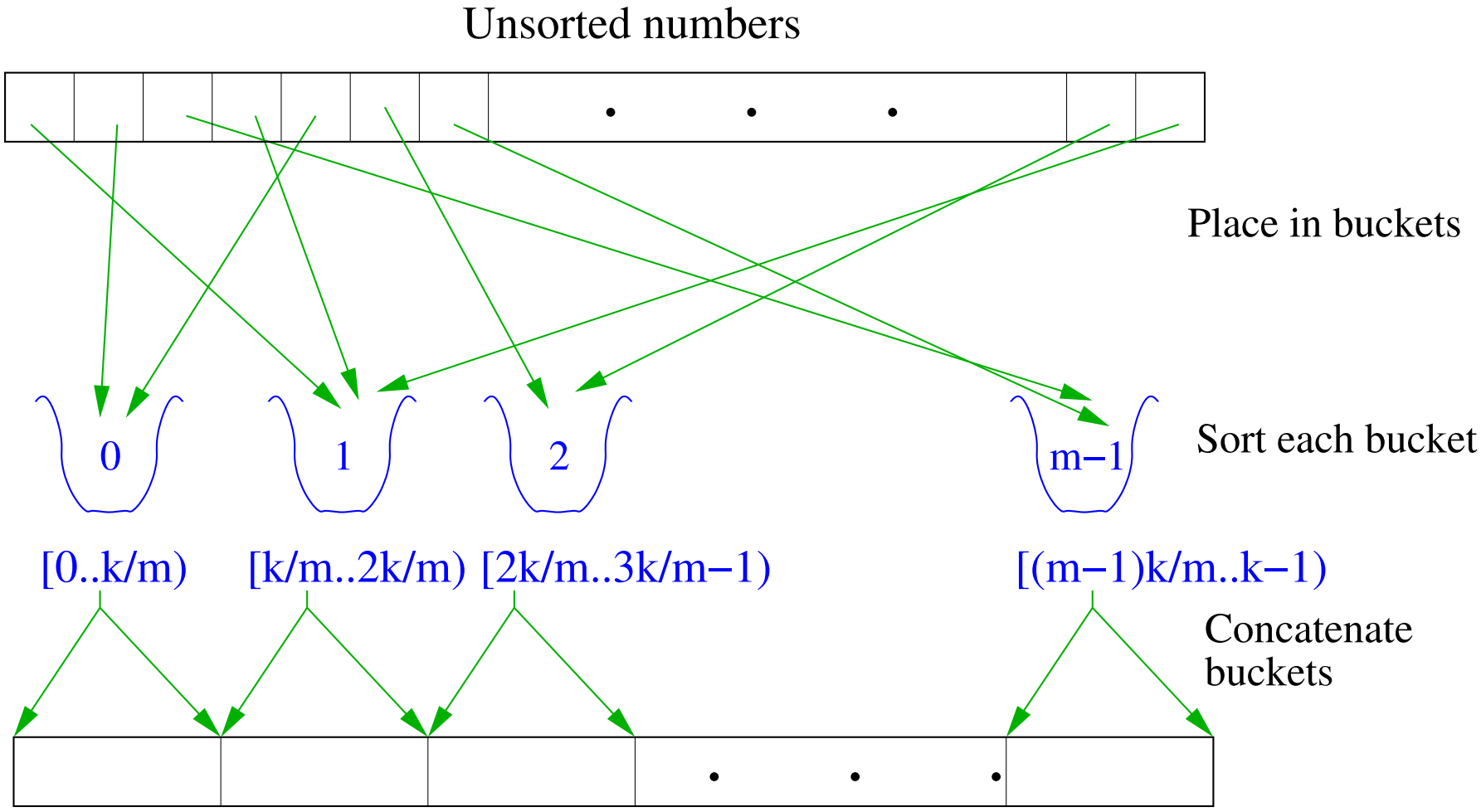
- We have n key values distributed uniformly in the range $[0..k-1]$.

Sequential Bucketsort

- We have n key values distributed uniformly in the range $[0..k-1]$.
- The algorithm uses m *buckets*, with the ith bucket representing key values in the range $[i\frac{k}{m} \ldots (i+1)\frac{k}{m} - 1]$, for $0 \leq i \leq m-1$.

Sequential Bucketsort

- We have n key values distributed uniformly in the range $[0..k-1]$.
- The algorithm uses m *buckets*, with the ith bucket representing key values in the range $[i\frac{k}{m} \ldots (i+1)\frac{k}{m} - 1]$, for $0 \leq i \leq m-1$.
- The sequential bucketsort algorithm is as follows.
  1. Place a key value x in the *bucket* $\lfloor x/(k/m) \rfloor$.
  2. Sort each *bucket* using an optimal sorting algorithm.
  3. Concatenate the results together into one array or list.



Sequential Bucketsort

Analysis of Sequential Bucketsort

- **Step 1**. This step requires $\Theta(n)$ time to walk through the list and place each *key* value into the appropriate *bucket*. At the the end of Step 1, each *bucket* has $\Theta(n/m)$ elements (assuming uniform distribution).

Analysis of Sequential Bucketsort

- **Step 1**. This step requires $\Theta(n)$ time to walk through the list and place each *key* value into the appropriate *bucket*. At the the end of Step 1, each *bucket* has $\Theta(n/m)$ elements (assuming uniform distribution).
- **Step 2**. Using an optimal sort in Step 2 would then take $\Theta(\frac{n}{m} \lg \frac{n}{m})$ time per *bucket*. Since we have m *buckets* to sort, the time for Step 2 is $\Theta(m \frac{n}{m} \lg \frac{n}{m}) = \Theta(n \lg \frac{n}{m})$.

Analysis of Sequential Bucketsort

- **Step 1**. This step requires $\Theta(n)$ time to walk through the list and place each *key* value into the appropriate *bucket*. At the the end of Step 1, each *bucket* has $\Theta(n/m)$ elements (assuming uniform distribution).
- **Step 2**. Using an optimal sort in Step 2 would then take $\Theta(\frac{n}{m}\lg\frac{n}{m})$ time per *bucket*. Since we have m *buckets* to sort, the time for Step 2 is $\Theta(m\frac{n}{m}\lg\frac{n}{m}) = \Theta(n\lg\frac{n}{m})$.
- **Step 3**. This step takes $\Theta(n)$ time.
- The sequential time for bucketsort is: $\boxed{T^*(n) = \Theta(n + n\lg\frac{n}{m})}$

Analysis of Sequential Bucketsort

- **Step 1**. This step requires $\Theta(n)$ time to walk through the list and place each *key* value into the appropriate *bucket*. At the the end of Step 1, each *bucket* has $\Theta(n/m)$ elements (assuming uniform distribution).
- **Step 2**. Using an optimal sort in Step 2 would then take $\Theta(\frac{n}{m}\lg\frac{n}{m})$ time per *bucket*. Since we have m *buckets* to sort, the time for Step 2 is $\Theta(m\frac{n}{m}\lg\frac{n}{m}) = \Theta(n\lg\frac{n}{m})$.
- **Step 3**. This step takes $\Theta(n)$ time.
- The sequential time for bucketsort is: $\boxed{T^*(n) = \Theta(n + n\lg\frac{n}{m})}$
- If we choose the number of *buckets* to be linear to n, say $m = rn$, where r is a constant between 0 and 1, then the sequential run time for bucketsort is

$$\boxed{T^*(n) = \Theta(n\lg\frac{n}{rn}) = \Theta(n)}$$

Non-uniformly Distributed Input

The bucketsort algorithm assumes that the input is uniformly distributed over a certain range. How can we use bucketsort if that is not the case? Here are some ideas.

Non-uniformly Distributed Input

The bucketsort algorithm assumes that the input is uniformly distributed over a certain range. How can we use bucketsort if that is not the case? Here are some ideas.

- If the input fits some standard distribution (like Normal distribution, Zipf distribution etc.) then we can divide the range using the probability density function to make the buckets be approximately balanced.

Non-uniformly Distributed Input

The bucketsort algorithm assumes that the input is uniformly distributed over a certain range. How can we use bucketsort if that is not the case? Here are some ideas.

- If the input fits some standard distribution (like Normal distribution, Zipf distribution etc.) then we can divide the range using the probability density function to make the buckets be approximately balanced.
- Use a universal hash function to randomize the data. An example of when this would be useful is if the key values are Internet URL's.

Parallel Bucketsort

- Assume that the n input elements are distributed across the p processes, with each process initially holding n/p unsorted elements. Further assume that each elements is in the range $[0 \ldots k-1]$.

Parallel Bucketsort

- Assume that the n input elements are distributed across the p processes, with each process initially holding n/p unsorted elements. Further assume that each elements is in the range $[0 \ldots k-1]$.
- *The goal is that at the end of the sorting, process i has elements that are sorted and any element on process i is greater than or equal to any element on process j, where $j < i$ and any element on process i is less than or equal to any element on process k, where $k > i$.*

Parallel Bucketsort (contd.)

Each process now has multiple little buckets (one per process that it may receive data from): we refer to these little buckets as a *box*. Set the number of *boxes* on each process to be $m = p$.

Step 1. Each process places its n/p key values into p *boxes* with key value x going to the *box* number $\lfloor x/(k/p) \rfloor$.

Parallel Bucketsort (contd.)

Each process now has multiple little buckets (one per process that it may receive data from): we refer to these little buckets as a *box*. Set the number of *boxes* on each process to be $m = p$.

Step 1. Each process places its n/p key values into p *boxes* with key value x going to the *box* number $\lfloor x/(k/p) \rfloor$.

Step 2. For each process i, where $0 \leq i \leq p-1$, send the jth *box* to process j, $0 \leq j \leq p-1, j \neq i$.

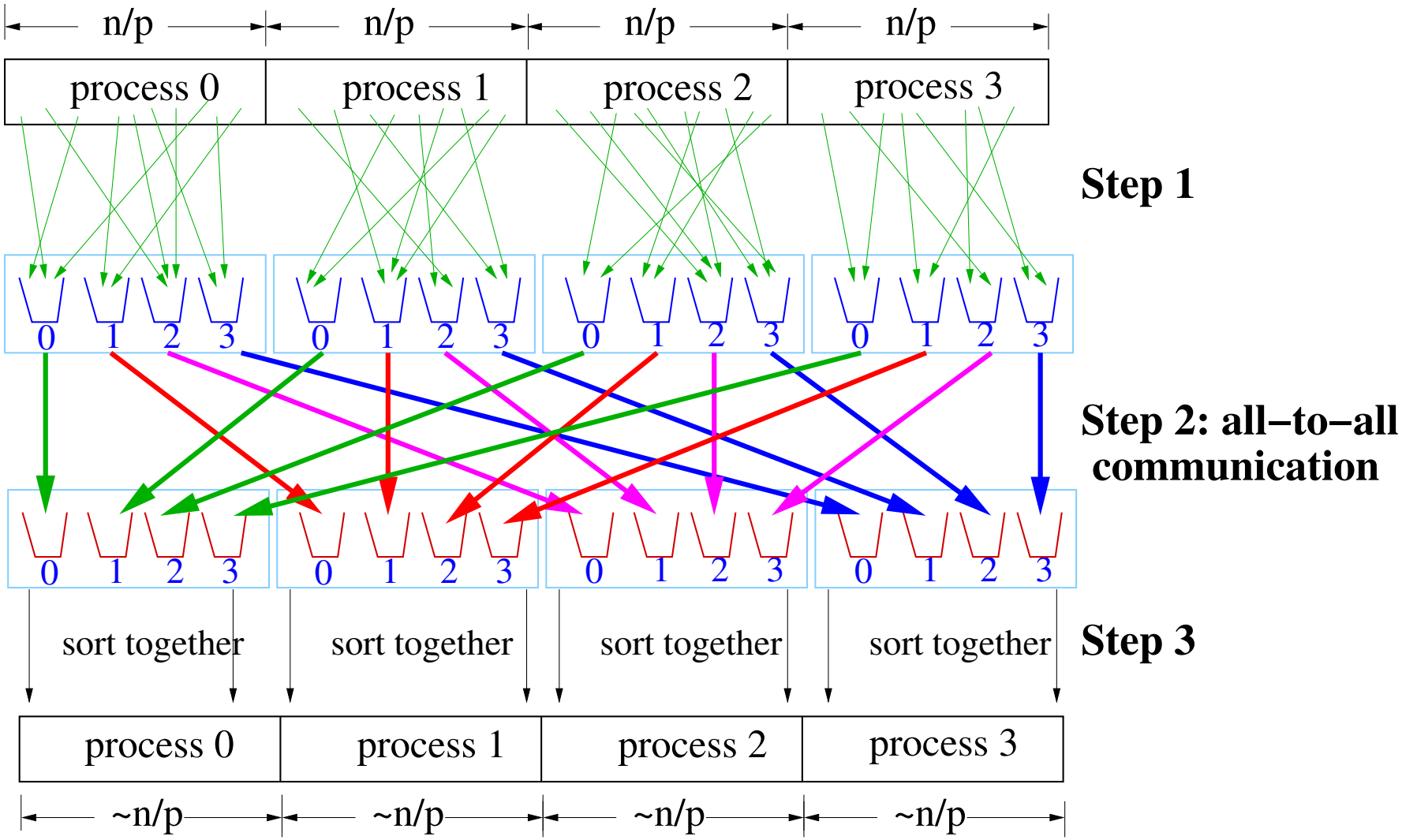
Parallel Bucketsort (contd.)

Each process now has multiple little buckets (one per process that it may receive data from): we refer to these little buckets as a *box*. Set the number of *boxes* on each process to be $m = p$.

Step 1. Each process places its n/p key values into p *boxes* with key value x going to the *box* number $\lfloor x/(k/p) \rfloor$.

Step 2. For each process i, where $0 \leq i \leq p-1$, send the jth *box* to process j, $0 \leq j \leq p-1, j \neq i$.

Step 3. Each process receives $p-1$ *boxes*. It combines them with its *box* and then sorts all of them together using sequential bucketsort to make one list of approximately n/p key values.

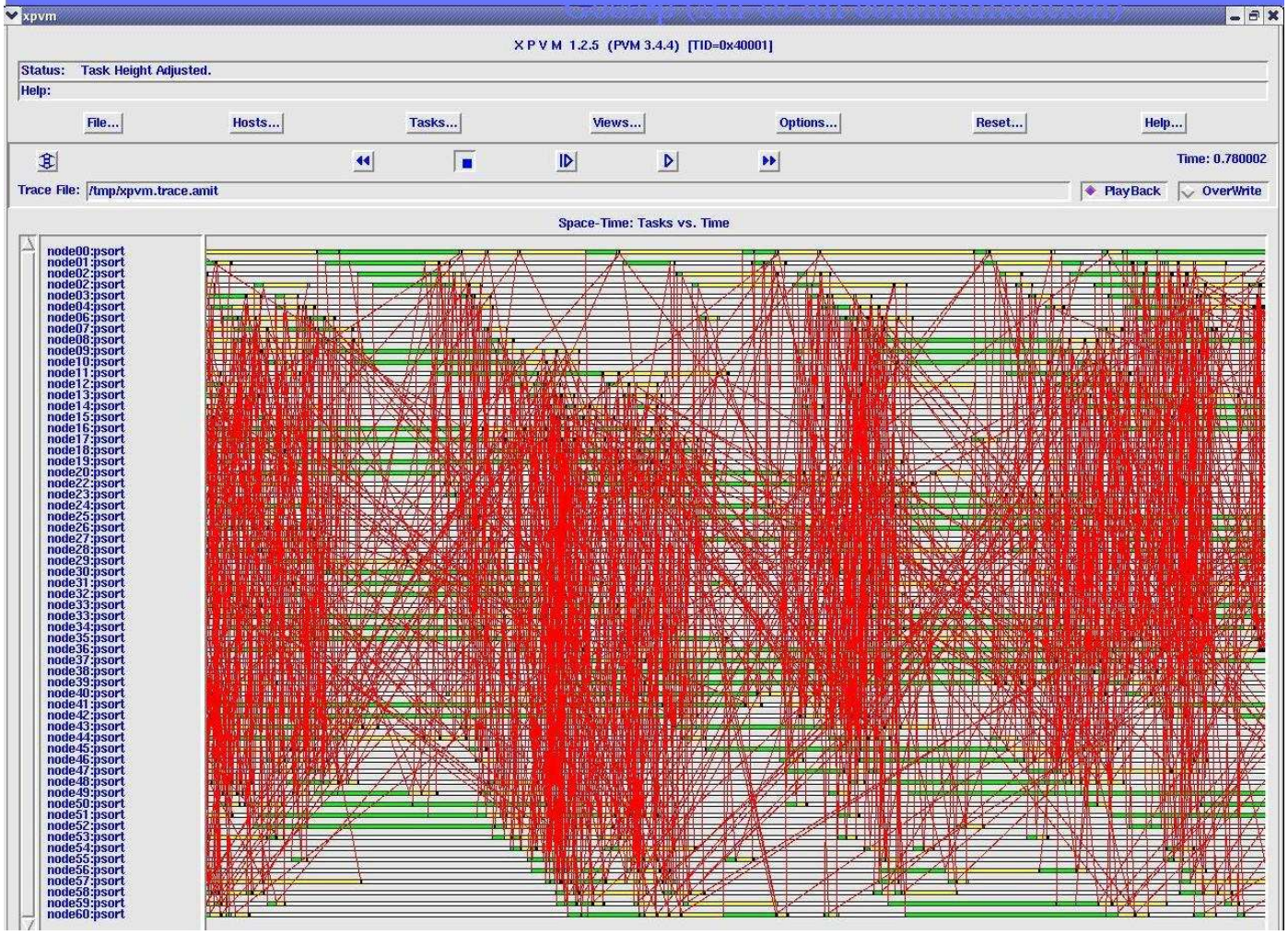


Parallel Bucketsort

Parallel Bucketsort Implementation

- Step 2 involves an **all-to-all communication** pattern (a.k.a gossip).
- The next frame shows a space-time visualization of an all-to-all communication. The example shows a parallel bucketsort program in action on a Beowulf Cluster.

Gossip (All-to-all communication)



Parallel Bucketsort Analysis

Step 1. $t_{\text{comp}} = \Theta(\frac{n}{p})$ $t_{\text{comm}} = \Theta(1)$

Parallel Bucketsort Analysis

Step 1. $t_{\text{comp}} = \Theta(\frac{n}{p})$ $\quad$ $t_{\text{comm}} = \Theta(1)$

Step 2. Assuming uniform distribution. Each box will have about n/p^2 key values. Each process send $p-1$ such boxes.

$$\begin{aligned} t_{\text{comp}} &= \Theta(1) \\ t_{\text{comm}} &= \Theta(p(p-1)(t_{\text{startup}} + \frac{n}{p^2} t_{\text{data}})) \end{aligned}$$

Parallel Bucketsort Analysis

Step 1. $t_{\text{comp}} = \Theta(\frac{n}{p})$ $\quad$ $t_{\text{comm}} = \Theta(1)$

Step 2. Assuming uniform distribution. Each box will have about n/p^2 key values. Each process send $p-1$ such boxes.

$$\begin{aligned} t_{\text{comp}} &= \Theta(1) \\ t_{\text{comm}} &= \Theta(p(p-1)(t_{\text{startup}} + \frac{n}{p^2} t_{\text{data}})) \end{aligned}$$

Step 3.

$$\begin{aligned} t_{\text{comp}} &= \Theta(\frac{n}{p} \lg \frac{n}{p}) \\ t_{\text{comm}} &= \Theta(1) \end{aligned}$$

Parallel Bucketsort Analysis

Step 1. $t_{\text{comp}} = \Theta(\frac{n}{p})$ $\quad$ $t_{\text{comm}} = \Theta(1)$

Step 2. Assuming uniform distribution. Each box will have about n/p^2 key values. Each process send $p-1$ such boxes.

$$\begin{aligned} t_{\text{comp}} &= \Theta(1) \\ t_{\text{comm}} &= \Theta(p(p-1)(t_{\text{startup}} + \frac{n}{p^2} t_{\text{data}})) \end{aligned}$$

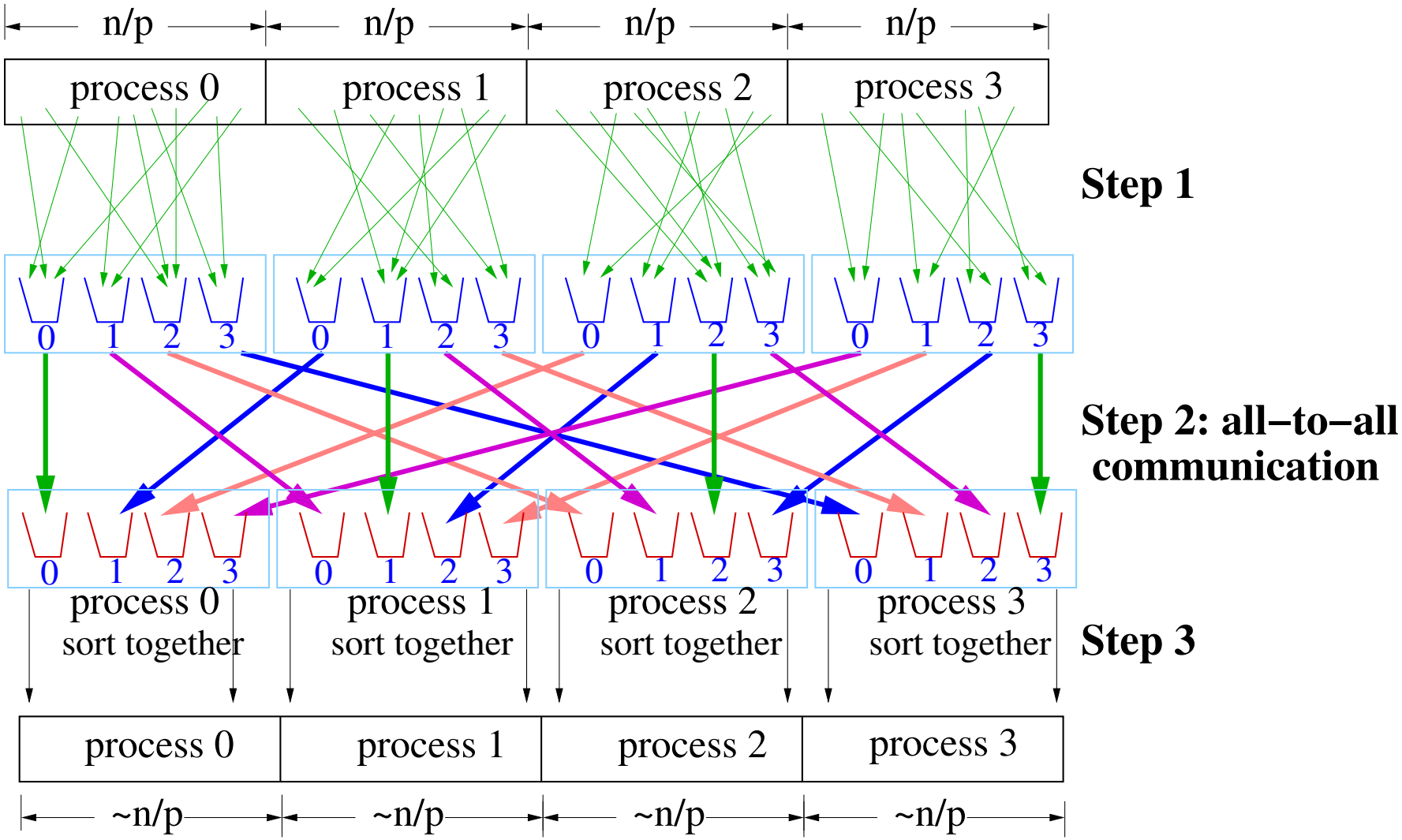
Step 3.

$$\begin{aligned} t_{\text{comp}} &= \Theta(\frac{n}{p} \lg \frac{n}{p}) \\ t_{\text{comm}} &= \Theta(1) \end{aligned}$$

Total: The overall runtime for parallel bucketsort:

$$\begin{aligned} T_p(n) &= \Theta(\frac{n}{p} + p(p-1)(t_{\text{startup}} + \frac{n}{p^2} t_{\text{data}}) + \frac{n}{p} \lg \frac{n}{p}) \\ &= \Theta(\frac{n}{p} \lg \frac{n}{p} + \overbrace{p^2 t_{\text{startup}}}^{\textit{problem...}} + n t_{\text{data}}) \end{aligned}$$

Can we cut down on the startup time?



Parallel Bucketsort

All-to-All Communication

- Note that we can break the all-to-all communication into p rounds as shown below.
  round 0 $P_0 \to P_0$ $P_1 \to P_1$ $P_2 \to P_2$ $P_3 \to P_3$

All-to-All Communication

- Note that we can break the all-to-all communication into p rounds as shown below.

round 0	$P_0 \to P_0$	$P_1 \to P_1$	$P_2 \to P_2$	$P_3 \to P_3$
round 1	$P_0 \to P_1$	$P_1 \to P_2$	$P_2 \to P_3$	$P_3 \to P_0$

All-to-All Communication

- Note that we can break the all-to-all communication into p rounds as shown below.

round 0	$P_0 \to P_0$	$P_1 \to P_1$	$P_2 \to P_2$	$P_3 \to P_3$
round 1	$P_0 \to P_1$	$P_1 \to P_2$	$P_2 \to P_3$	$P_3 \to P_0$
round 2	$P_0 \to P_2$	$P_1 \to P_3$	$P_2 \to P_0$	$P_3 \to P_1$

All-to-All Communication

- Note that we can break the all-to-all communication into p rounds as shown below.

round 0	$P_0 \to P_0$	$P_1 \to P_1$	$P_2 \to P_2$	$P_3 \to P_3$
round 1	$P_0 \to P_1$	$P_1 \to P_2$	$P_2 \to P_3$	$P_3 \to P_0$
round 2	$P_0 \to P_2$	$P_1 \to P_3$	$P_2 \to P_0$	$P_3 \to P_1$
round 3	$P_0 \to P_3$	$P_1 \to P_0$	$P_2 \to P_1$	$P_3 \to P_2$

All-to-All Communication

- Note that we can break the all-to-all communication into p rounds as shown below.

round 0	$P_0 \to P_0$	$P_1 \to P_1$	$P_2 \to P_2$	$P_3 \to P_3$
round 1	$P_0 \to P_1$	$P_1 \to P_2$	$P_2 \to P_3$	$P_3 \to P_0$
round 2	$P_0 \to P_2$	$P_1 \to P_3$	$P_2 \to P_0$	$P_3 \to P_1$
round 3	$P_0 \to P_3$	$P_1 \to P_0$	$P_2 \to P_1$	$P_3 \to P_2$

- Note that in each round each process is sending out one message and receiving one message. If the processes are on nodes connected via a switch (completely connected network), then each round can be done simultaneously. This does assume that the network interface is capable of sending/receiving at the same time (which is true for most network interfaces but not all).

All-to-All Communication

- Note that we can break the all-to-all communication into p rounds as shown below.

round 0	$P_0 \to P_0$	$P_1 \to P_1$	$P_2 \to P_2$	$P_3 \to P_3$
round 1	$P_0 \to P_1$	$P_1 \to P_2$	$P_2 \to P_3$	$P_3 \to P_0$
round 2	$P_0 \to P_2$	$P_1 \to P_3$	$P_2 \to P_0$	$P_3 \to P_1$
round 3	$P_0 \to P_3$	$P_1 \to P_0$	$P_2 \to P_1$	$P_3 \to P_2$

- Note that in each round each process is sending out one message and receiving one message. If the processes are on nodes connected via a switch (completely connected network), then each round can be done simultaneously. This does assume that the network interface is capable of sending/receiving at the same time (which is true for most network interfaces but not all).
- Further improvement can be made by making use of multicasting (if supported in the underlying hardware). With multicasting, we will have $p-1$ messages, each one a multicast message. This would cut down the parallel time to:

$$T_p(n) = \Theta(\frac{n}{p} \lg \frac{n}{p} + pt_{\text{startup}} + nt_{\text{data}})$$