CS 606	**More Sorting**

We will now see what happens when you try to sort N keys using P processors.

First, let us pick up where we left off in the last lecture, sorting N in a mesh of size $\sqrt{N} * \sqrt{N}$. We shall now try to do the same task, but in a smaller mesh, of size $\sqrt{P} * \sqrt{P}$ where $P \leq N$.

One quick way is to simulate the $\sqrt{N} * \sqrt{N}$ network in the $\sqrt{P} * \sqrt{P}$ network as shown in Fig 1. We divide the $\sqrt{N} * \sqrt{N}$ buffer into smaller meshes, each of size $\sqrt{\frac{N}{P}} * \sqrt{\frac{N}{P}}$, assigning each to a processor of the host network. This is a good way because neighbors in the guest network will remain neighbors in host network too.
Hence, each processor of the host mesh ends up doing the work of $\frac{N}{P}$ processors of the guest. Recall that the guest network sorted the N keys in $O(\sqrt{N})$. Hence, the time required to sort N keys in the host is $O(\sqrt{N} * \frac{N}{P})$.

Note that the Speedup lower bound (sequential time / number of processors) is $\frac{N \log N}{P}$, while the diameter (half the diameter of the graph) bound is $2 * \sqrt{P} - 2$. Our algorithm differs form these bounds by a big amount, but it turns out we can get both, a better algorithm as well as better (higher) bounds.

Better Bound:
We define the Bisection Width of a graph as the minimum number of edges necessary to be removed in order to disconnect the graph into two (roughly equal) parts. To develop the concept of the bisection width lower bound, we note that in any graph of processors, we know the nodes where the input shall be taken and where the final output be presented. We can then consider the case where input is provided in the most inconvenient way, and then argue that it must travel to the output nodes.
e.g. In this case, where each row must be sorted from left to right (or vice versa), we can put the largest elements in the left half of the row and smallest in the

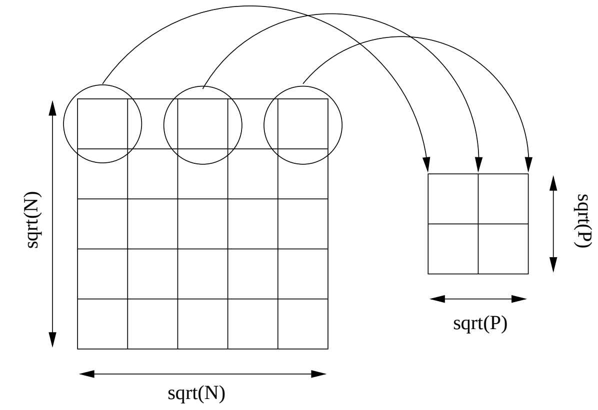

Figure 1: N-key sorting on P-processor mesh

right half. These elements must then get exchanged one-by-one at the center of the row. This will take at least $\frac{N}{2}$ steps.

Hence, consider the bisection width bound for the full graph, (number of data to be exchanged / number of wires), which is $\frac{\frac{N}{2}}{\sqrt{P}}$. $\frac{N}{\sqrt{P}}$ is a tighter bound than either of the above bounds, except if $N >> P$, when $\frac{N \log N}{P}$ is higher.

Faster Algo:

To get a faster algo, we define a Merge-Split (MS) operation, which is the equivalent of the Compare-Exchange operation, in the case that there are many keys per processor. Consider two processors each with $\frac{N}{P}$ keys. An MS-operation results in the following: The two sets of keys are retained in memory, as well as exchanged between the processors. Comparing the new keys with the old ones, one of the processors retains the lower keys, while the second retains the higher half of the keys.

Generalized simulation sorting algorithm: Using the MS operation, we can define the Generalized simulation sorting algorithm, to convert any N-key N-processor algorithm to the corresponding N-key P-processor algorithm:

1. Each processor reads $\frac{N}{P}$ keys and does a local sort of them. Time taken: $O(\frac{N}{P} \log \frac{N}{P})$.

2. Take the N-key N-processor algo and replace each CE operation with an MS operation.

Time complexity:
Time taken is $O(\frac{N}{P} * P\_key\_sorting\_time)$. The $\frac{N}{P}$ factor is due to the exchange of $\frac{N}{P}$ keys that the MS operation entails. In our case of sorting N-keys in a $\sqrt{P} * \sqrt{P}$ mesh, the time taken will be: $\frac{N}{P} \log \frac{N}{P} + \frac{N}{\sqrt{P}} * P$.

- For intermediate values of N ($N = P^k$), this is approximately $O(\frac{N}{\sqrt{P}})$, which is the bisection width lower bound.
- For large N ($N = \exp P$), order is $O(\frac{N}{P} \log \frac{N}{P})$, which is the speedup bound.

Proof of correctness: We will prove that the above algorithm is correct only in the case where data is only 1's and 0's. (The 0-1 sorting lemma will then prove correctness for all cases.) To prove that the above algorithm works, we first make some definitions:

- Small Problem: The P-key P-processor problem
- Large Problem: The N-key P-processor problem
- t-state: Any state of processors that can be observed after t steps in time.
- Image: An image is a relation between t-states of the small and large problems. If we represent the memory of a processor in the large problem, we see that it must contain $\frac{N}{P}$ bits. For the small problem, however, each processor has only 1 bit. We then examine corresponding processors in the large and small problems and define an image as shown in fig 2. The image of a t-state is the composite of the images of individual processors.

We will first prove two lemmas which will lead to the required proof:

1. *If every image of the large problem is sorted (and if numbers within every processor of the large problem are internally sorted), then the large problem is also sorted*

   Proof By contradiction. Let L be a t-state of the large problem such that it is internally sorted, and every image img(L) of it is sorted, and let L not

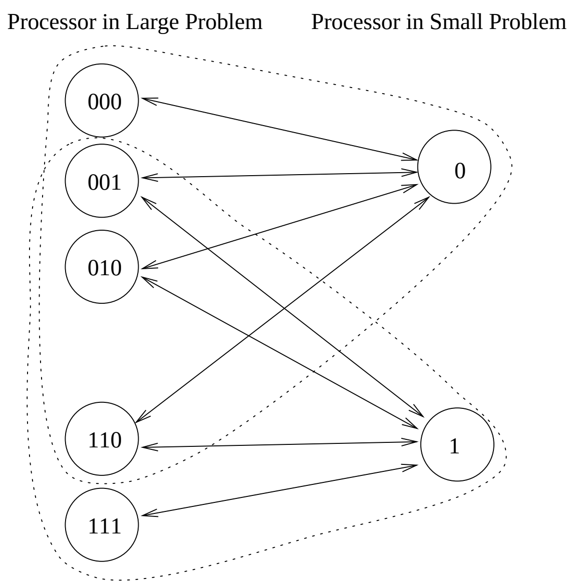

Figure 2: Image os a t-state

be sorted. Since L is not sorted, there is at least one 0 that appears *after* (in the logical sense) at least one 1. Hence, there exist processors i and j of the large problem, such that there is at least one 0 in some Proc i and at least one 1 in some Proc j such that $i > j$ (N.B. Numbering of processors is done in the order in which the sorted numbers appear finally). Note that $i = j$ is not possible because we are assuming there is complete internal sorting. But this means that Proc i can have an image of 0, and Proc j of 1, and thus we have found an image of L (which is the composite image of the individual images) that is NOT sorted. Hence, there cannot be any such t-state L of the large problem.

2. *If C is a t-state for the lage problem, then every img(C) is a t-state for the small problem*

   Proof By induction.
   Base case: At t = 0, all images of any input to the large problem are legal t-states of the small.
   Induction Hypothesis: If C be a t-state of large problem, every img(C) is a t-state of small problem. To prove: Let D be a (t+1)-state derived from C. Then, every img(D) is a (t+1)-state of small problem.

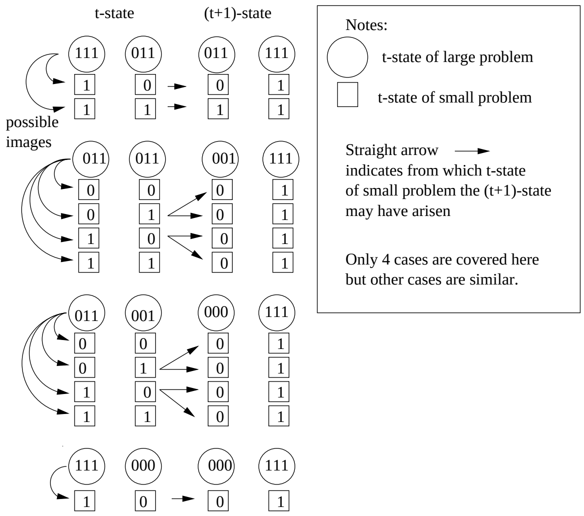

Figure 3: Images of t-states and corresponding (t+1)-states

We consider many cases as shown in Fig 3.

Now, consider what happens after t = $Small\_problem\_sorting\_time$. By Lemma 2, images of all t-states of the large problem are t-states in small problem. And these t-states in small problem are all solutions to small problem, since t = $Small\_problem\_sorting\_time$. Now, by Lemma 1, we conclude that the large problem is also sorted. Hence, we have a faster algorithm than a simple simulation.