Simulating a Parallel Random Access Machine

Introduction to Parallel Algorithms and Architectures,

§3.6

Why simulate?

- The PRAM is an excellent framework for studying parallelism.
- However, a global shared memory is not easily implementable on a large scale.
- *Practical approach*: construct a fixed-connection network and simulate the PRAM on it.

Simulation on a butterfly

- Each *PRAM processor* is simulated by a node of the butterfly.
- The *global memory* is distributed among the nodes of the butterfly.
- Memory *access*: send a packet to the appropriate node.
- Memory *read*: said node returns the desired data.

A worst-case scenario

- If #memory cells >> #processors, memory contention may be an issue (even with EREW).
- $M \geq N^2$: all N processors may wish to access memory locations that reside on the same node.
- Combining will not help in practice.

A randomized simulation based on hashing

- To simulate an N-processor PRAM with M memory cells on an N-node butterfly:
  - We randomly distribute the M memory cells among the butterfly's N local memories using a O(logN)-wise independent random hash function $h:[1,M]\rightarrow[1,N]$.
  - The packet routing problem that emerges for a single step of the PRAM computation is an average-case routing problem, solvable in O(logN) steps with high probability.

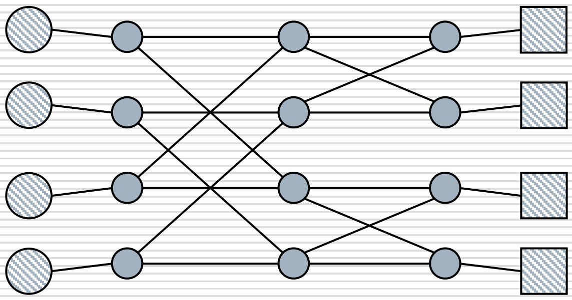
A closer look at the simulation

- Route each packet within its row to level 0. (each row ends up with O(logN) packets)
- Then, route each packet to its correct row. (in O(logN) steps with O(1)-size queues)
- Finally, route each packet to the correct level within its destination row. (with high probability there are O(logN) packets destined for each row, so this takes O(logN) steps)

Methods for improving efficiency

- The simulation we described is optimal, since each processor may wish to access data that is $\Omega(\log N)$ away in the network.
- However, using a $\log N$-dimensional butterfly yields a $\Theta(\log N)$-factor improvement in the efficiency of the simulation.

Simulation using a logN-dimensional butterfly



- Can be used to simulate a NlogN-processor PRAM.
- Each input node simulates logN processors.
- Routing can still be done in O(logN) steps with high probability.

Simulating with data replication

- *Data replication*: make multiple copies of the data stored in the global memory.
- *Idea*: if there is contention for one memory block, we might still gain quick access to another memory block that replicates the data we need.

Data replication overhead

- Storing k copies of each data item takes k times as much total space.
- Keeping track of old copies.
- We need to ensure that any set of N memory locations can be accessed quickly.

A deterministic simulation using replicated data

- Each item is replicated k=logM times.
- Any set of N items can be accessed in O(logM logN loglogN) steps on an N-node butterfly.
- Each copy of an item includes a *timestamp* (PRAM step during which the copy was last updated).
- To complete a memory access, we have to successfully access at least $\left\lceil \frac{k+1}{2} \right\rceil$ copies.

A special hash function for data replication

- The j-th copy of the i-th item will be stored in memory location $h(i,j)$ where $h:[1,M]\times[1,k]\rightarrow[1,N]$ is a special hash function satisfying:
  - any block of memory stores $O(Mk/n)$ copies of items.
  - the copies of any set of s items are spread across at least $3ks/4$ blocks of memory, for $s\leq\varepsilon_0 N/k$.

A phase of the simulation (1)

1. Compute the number of unsatisfied requests, I_t.
2. Identify a set of $s=\min\{I_t, \varepsilon_0 N/k\}$ *active* unsatisfied requests.
3. Relocate the i-th active request to node (i-1)k+1.
4. Make k copies of each request, and store the j-th copy of the i-th request in node (i-1)k+j.

A phase of the simulation (2)

5. Sort the sk resulting requests by destination block, and eliminate all but one for each block. At least 3sk/4 requests survive.
6. Route surviving requests to their destinations, and return successful packets to the node where they originated.
7. Check whether or not (k+1)/2 or more copies of each active request were satisfied.
8. Identify a current copy for each satisfied request.

An upper bound on the number of phases

- An active request is not satisfied $\Leftrightarrow$ at least k/2 of its copies are not satisfied.
- But the number of copies of active requests that are not satisfied is at most ks/4.
- Therefore, at least s/2 active requests are satisfied in each phase.
- It turns out that O(logM) phases suffice.

Running time of the simulation

- Each phase of the simulation can be completed in O(logN loglogN) steps.
- The loglogN-factor is due to sorting.
- The running time can be improved by a logloglogN-factor with a better analysis.

Information dispersal

- Encode each item z into k pieces z_1, z_2, …, z_k such that:
  - $|z_i| \approx 3|z|/k$, for each i.
  - z can be reconstructed from any $k/3$ pieces.
- Each time we need to access z, we are content with accessing $2k/3$ pieces of z.

Using information dispersal to improve performance

- $O(\log M)$ phases of the previous algorithm still suffice.
- However, the operations involve much shorter items and we can expect things to run $k/3=\Theta(\log M)$ times faster.
- Therefore, the running time now becomes $O(\log N \log\log N)$ steps.