Summary of *LogP* by Culler et al

Nemanja Isailovic
nemanja@cs.berkeley.edu

February 27, 2002

1 Introduction

The goal of this paper is to design a single model that applies to a wide range of realistic parallel processors.

Here is a list of guesses/assumptions made by the authors about the future of parallel computing, not all of which are essential to the LogP model:

- Parallel machines are a collection of complete comuputers. That is, each node contains a uniprocessor, including microprocessor, cache(s) and some memory.
- They use off-the-shelf microprocessors rather than custom made ones.
- They have thousands of nodes at most (so no million node machines).
- Parallel algorithms tailored to specific network topologies are unlikely to be useful (due to the prevalence of faults and adaptive routing). So we would like to design network topology-independent algorithms.

2 Previous Models

2.1 Parallel Random Access Memory (PRAM)

Assumptions:

- synchronous operation of processors
- a single shared memory with unit access time for each processor
- infinite network bandwidth, zero latency, zero overhead
- no memory contention

As a result of these assumptions, the PRAM model does not discourage excessive communication in algorithms. Many other versions of PRAM exist, but each one solves only one or two minor problems with the basic model.

2.2 Bulk Synchronous Parallel Model (BSP)

This model describes a parallel processor as a set of processor/memory modules, a network and a synchronizer (meaning that some mechanism is in place which keeps the processors operating synchronously). A computation consists of a series of *supersteps*. Within each superstep, each processor is allowed to perform local computations which depend exclusively on locally available data. Additionally, each processor may send up to h messages and receive up to h messages (where h is some predetermined constant) within each superstep. However, the data within the received messages is not available to the processor until the next superstep.

There are problems with this perspective:

- A superstep must be large enough to accomodate the worst *2h* message transfers possible. This likely results in very bad running time estimates.
- Messages sent at the beginning of one superstep cannot be used until the next superstep, even though they may arrive much earlier.
- Synchronization hardware is necessary, and that may not be available.

3 LogP Model

This model is based on four paramemters:

- L: transfer latency; an upper bound on the latency incurred in communicating a message (of a few words at most) from source to target (this is the worst case latency)
- o: communication overhead; this is the length of time a processor is engaged in transmission or reception of a message (we assume that no other operations can be performed in parallel)
- g: communication bandwidth; minimum time interval between consecutive message transmissions or receptions at a processor (this is equal to the message size divided by the per-processor bisection bandwidth)
- P: number of processor/memory modules

Further assumptions of the LogP Model:

- Small messages are used exclusively. This allows a tighter bound on L.
- The network has finite capacity. At most $\lceil (L/g) \rceil$ messages can be in transit from or to any processor at any time. A processor will block if sending another messages would violate this rule.
- Each local operation takes unit time.
- Processors are allowed to work asynchronously. Any synchronization barriers must be explicit.
- Per-message latency is unpredictable, but bounded by L.

This model has certain benefits over previously offered models:

- encourages locality of data, since communication is costly

- encourages overlapping of computation and communication to mask limits of the network (that is, if g is larger than $2o$, then there is idle time between successive message sends; this model allows this time to be used for computation)
- encourages balanced communication patterns

4 Examples

4.1 Broadcast

We want to broadcast from one processor to $P - 1$ others. The idea is that each processor that receives a message will transmit it to all neighbor processors as fast as possible, while making sure not to send any repeats. The source begins transmitting at time 0. The datum enters the network at time o, takes L cycles in transit and is fully received at the first target processor at time $L + 2o$. Meanwhile, the source initiates transmissions to other processors at times g, $2g$, $3g$, ... (assuming $g \geq o$). Figure 3 in the paper gives a good example of a possible run of this algorithm.

4.2 Summation

We want to sum n inputs, so we basically use the broadcast tree from the previous example. Each processor will sum its own subset of the n elements with the results from its children and then send the result to its parent.

Suppose it takes T time to finish the algorithm. The last summation at the root (the root's own partial sum with the last result from one of its children) must be performed at time $T - 1$ (remember that local operations take unit time). The child must have sent this value at time $T - 1 - (L + 2o)$ in order for it to arrive on time, and the root's local partial sum must be ready at time $T - 1 - o$ (since the root must devote all its resources to receiving the message during the overhead period). Since the root can receive a message every g cycles, the children must be done at times $T - 1 - (L + 2o)$, $T - 1 - (L + 2o + g)$, $T - 1 - (L + 2o + 2g)$, ... The root performs $g - o - 1$ local additions between messages. Additionally, each processor invests o cycles receiving a message from a child, so each partial sum must represent at least o additions. Using all this information and the broadcast tree from above, we can determine an appropriate computation schedule.

4.3 Fast Fourier Transform

A standard butterfly computation on an n-input FFT contains $n * (\log n + 1)$ individual computations (as per Figure 5 in the paper). These need to be spread out over P processors where $P \ll n$.

4.3.1 Some Solutions

One possibility is to assign the first row of computations (in the butterfly) and every P^{th} row after that to the first processor, the second row and every P^{th} row after that to the second processor, etc. The result of this is that the first $\log(n/P)$ columns (excluding the first one, which is not a compuation column) require only local data, while the last $\log P$ columns require a remote access for each compuation.

Another possibility is to assign the first n/P rows to the first processor, the next n/P rows to the next processor, etc. The result is that the first $\log P$ columns require a remote access for each computation, while the last $\log(n/P)$ columns require only local data. This results in exactly the same number of communications as the idea above.

In either case, each processor has n/P computations per column over $\log n$ columns, for a total compuation time of $(n/P)\log n$, assuming unit time for each computation. Additionally, each processor must send n/P messages per column (one for each of its computations) for a communication time of $g(n/P)+L$ per column. This must be done for each of $\log P$ columns, for a total communication time of $(g((n/P)+L)\log P$ (assuming $g \geq 2o$).

4.3.2 A Better Solution

The first solution above uses only local data for the first $\log(n/P)$ columns, while the second solution uses only local data for the last $\log(n/P)$ columns. Combine these two, using the first layout at the beginning, the second one at the end, and having a single remap step in which every processor must communicate with every other processor in order to rearrange the data. This hybrid layout results in entirely local communication in all columns but one, where there's massive communication necessary.

The computation time is still $(n/P)\log n$ per processor as before. The communication now occurs in a single step. There are n rows. n/P of these belong to a given processor. $1/P$ of these rows must go to another given processor. Thus, each processor must send n/P^2 messages to each other processor. So each processor must send $(n/P^2)*(P-1)$ messages, for a total communication time of $g*(n/P-n/P^2)+L$. This is clearly much better than the time derived for the previous algorithms.

5 Issues to Consider

1. This model doesn't take caches into account. Is this a major problem? Will algorithms with large working sets fail to be simulated properly by this model?

2. This model doesn't take context switching into account. If the number of processes is $\gg$ than the number of processors, will this model still be accurate?

3. This paper claims that parallel processors need to be robust over all algorithms. Is this true? Should algorithms be designed for efficiency (with the network topology in mind) or for portability (topology-free)?

4. The authors claim that the average distance travelled by a message varies only slightly across topologies (see the table on page 15 of the paper), and that the transfer latency is dwarfed by the overhead anyway. But this conclusion depends on minimal contention in the network, which may not be uniform across topologies. Is the topology-free approach flawed from this perspective?