Loosely Synchronous Model As Parallel Structure

The advent of high performance computers has created diverse classes of parallel applications in the scientific community. From biomolecular simulations to astronomic predictions, the vast problem size and the time-to-solution requirement has created large demands for the use of high throughput parallel processing power. However, it is often noted in the community that, while the peak performance of these computing machinery is evolving at a great pace, the parallel programming models that embody the problem solutions are still suffering from failure to bring peak performance, easy maintenance, and efficient implementation process.

The sequential programming models leverage from structured programming models where peak performance, maintenance, and efficient implementations can be achieved through the use of loops, indices, and abstractions such as objects. Anchoring these structures as the starting point, various optimization techniques can be applied while preserving conciseness and readability of the program at the same time. Conversely, it has been stated in Dijkstra's argument[2] that without structured style a program's readability and performance will seriously be damaged.

Unfortunately, structured programming style is not precisely how most of parallel scientific codes are written in currently. Widespread usage of message passing model has encouraged many codes to be written in parallel-equivalent of go-to statements, which Dijkstra considers harmful in [2]. Namely, message transfer library calls such as send() and recv() without any restriction can appear anywhere in the program blocking the process waiting for data to be flushed or received. This is harmful for numerous reasons. First, programmers are now being burdened with possibility of deadlock. Second, excessive use of unstructured library calls can make the program cumbersome to read and maintain. Third, the entangled calls hidden in the programs limit the optimization(by human and also compiler) opportunity to fine-tune the program's performance. Identification of a structured programming model that will bridge the continuously growing gap between hardware peak performance and programmability is required for the balanced evolution of the high performance computing community.

Loosely synchronous model is a programming model that is composed of sequence of local and global phases. It has been also noted by Valiant in 1991 as "Bulk Synchronous Model" for shared memory environment. In local phase each process will perform computation with its own data, not concerning at all what other process is doing. By global phase, any necessary synchronization or communication between processor is done; no computation is done in the global phase. By decoupling the computation with communication the programmer

does not have to worry about deadlock situation or trace through library calls to see which data is ready. Each phase gives the programmer complete assumption that data is ready to be computed or communicated. Through chunking of the programming structure this way, progress of a process is much more readable than entanglement of library calls and opportunity of optimization is better exposed through sectioning of the parallel codes.

Valiant has argued that loosely synchronous model provides structure sufficient to pinpoint the progress of a parallel program, and any program can be transformed in such way without of loss of efficiency. But in many cases, structures can become too rigid and can hinder the performance of the program. For example, uneven workloads in local phases of different process can certainly bring overhead to the lightly loaded process. For this reason, loosely synchronous model must be accompanied by relaxation of constraints to expose better opportunity for the optimizations. Proper application of relaxation now enables us to express any parallel algorithm into loosely synchronous model without having to worry about the performance. We will use loosely synchronous model as framework for all algorithmic patterns from this point.

1. Relaxation of Synchronization
   - I don't have to wait to synchronize, I'll just do it now.

2. Relaxation of Communication
   - I don't have to wait to communicate. I'll just do it now.

3. Relaxation of Computation
   - I don't have to wait to compute. I'll just do it now.

Event-Driven Model

The instance deriving from the relaxation of the synchronization is "Event-Driven Model".

1. **Description of Event-Driven Model.**
   - From loosely synchronous local phases, synchronization have been totally removed between stages. Supersteps are no longer in need for synchronization.
   - However it does not mean the phases are EmbarrasinglyParallel or totally independent of each other. They do require "events" or signal from each other in order for their computations to advance to next superstep. The events can be simple a messaging saying the data is ready or the wanted data itself.
   -
2. **Driving Force of Event-Driven Model**
   - To send the data as soon as possible when the data is ready.
   - To hold the data as late as possible when the data is in cache
     - (Cache Driven)
   - To react to the data as soon as possible by doing appropriate computation.
3. Characterization of Event-Driven Model
   - Elimination Tree (Dependency Graph)
   - Elimination of outer loop
   - Event Detection / Book Keeping
   - Where to Send the Ready Data
   - Decomposition / Aggregation / Mapping / Load Balancing /
   - Optimization

4. Applications and Example of using Event-Driven Model
   - Cholesky Factorization