Event-Driven Pattern

I. Problem

The problem consists of performing a series of transformation on a domain where data distribution is highly irregular or sparse. The decomposed partitions are dependent on each other's update in order to advance in the transformation sequence, but not bound to a particular pattern or direction. In other words, the dependency relationship between the partitions is not fixed to the problem, but is dynamically determined by the content of the data structure. Furthermore, the workloads of partitions in many cases are not even because of irregularity in data layout. Because of such reasons, each units of execution does not know when and where exactly to expect its update, solely "reacting" to a series of asynchronous events that other units "fire" until completion requirements have been met.

II. Driving Forces

1. Any unit of execution must not sit idle waiting for events or due to lack of computation.
2. Workload between partitions should be evenly balanced.
3. Loosely synchronous framework should be relaxed, but must be preserved for readability and maintainability.
4. Sending event must be done asynchronously, and receiving end must keep a data structure to efficiently manage events.
5. Computation step should be as coarse as possible and communication should be aggregated in order to reduce overhead cost and to embody loosely synchronous style.
6. Exploiting memory hierarchy (reordering, blocks) should be taken into consideration when referencing memory.

III. Solution

1. Form a primitive loosely synchronous program.

As described above, any program can be expressed as loosely synchronous program without loss of efficiency. Identifying computation and communication phases for the problem is the starting point of constructing a loosely synchronous program. Note this stage only sets up the primitive structure for the program, oblivious to any performance issues. Performance will be tuned in the later stage by relaxations and optimizations. Nonetheless, optimization opportunities depend on clarity of primitive phase identification process.

It is effective to see the program in global view when identifying computational and communication phases. Rather than having a segregated viewpoint that only focuses on individual partition exchanging data, formulate the problem into a series of nested loops where whole flow of the program can be seen sequentially. At the inner core values are loaded, computed, and stored

back to the appropriate continuous place, and the outer loop shifts the location that operations are being applied to. Obvious observation here is that each iteration of the outermost loop is a common candidate for marking as one "phase". An iteration of outer loop means exactly that a set of recurring computation has just been completed and therefore ready to be applied again to another set of data, which is the exact definition of phases. However, a loosely synchronous phase requires that there is no communication operation in until global communication phase. Continue observing from outer loop towards inner loops until a phase has been identified without any remote data dependency.

Coarse-grained phases are crucial in maintaining structure of loosely synchronous program. If the phases are too narrow or fine-grained, recurring overhead is not the only problem. The readability and maintainability of the program will significantly reduce as the narrower phases gets entangled in the frequent communication calls, forming a "spaghetti code". Be sure to make the phases as coarse as possible by delaying all the communication operation until it absolutely has to come.

2. Eliminate global synchronization

Now comes the relaxation stage where the program is "enhanced" in adaptation to asynchronous nature of the problem. Since the problem has imbalanced workloads and irregular communication pattern, it will be disadvantageous if all units of execution wait in barrier between each phase. Conversely, the performance will be increased if each unit of execution is not bound to wait until the global synchronization for communication operation; just send the update to appropriate destination as soon as it is ready.

Apply the relaxation by eliminating the global synchronization stage. Let each unit of execution start communicating after their updates are ready, and continue on to the next computational stage. Note that relaxation does not remove the loosely synchronous structure because computational phases will be automatically blocked if needed update has not arrived. Though not clear cut as the primitive structure, this block will act as an unseen barrier for the unit of execution.

3. One-sided communication / nonblocking send recv

Relaxation causes communication to occur right away when data is ready, but the receiving end might not be ready doing some significant local computation of its own share, causing the sender to block. If the receiving end is not busy, it will block and sit idle until the update arrives, and this reduces processor efficiency. Tightly coupled communication scheme is no longer an effective measure for signaling "events' or updates in the relaxation of loosely synchronous program.

As a consequence, usage of relaxation techniques must be followed by one-sided communication. These are put/get methods where only one side is knowledgeable about the transfer. In the case of this pattern, an unit of execution will "put" the data to the destination when its ready, with the receiver not being aware of the transfer taking place. It is important to make a

bookkeeping data structure that will efficiently record and retrieve the events that have come in. Although the receiver is not aware of events during its occurrence, it can check from time to time (i.e. between local computation) for arrival of events.

Another convenient approach(in cases where one-sided communication is too expensive or not provided) is to use non-blocking send and receive. In such methods, initiation of send/recv will be done first and control is returned to the program. The program can continue on to its computation right after this initiation process, and the communication will happen in the background. The buffered data is loaded when wait() call is issued afterwards. Overlapping communication in between the initiation and wait() is repeated until there is no more computational phases to overlap, in which that case the unit has no choice but for waiting for trigger event. Between phases, received data will be taken into account, possibly generating new computation. In this pattern, it is important to use wildcard irecv(*) because the events arrival time and source is not known ahead of time.

4. If applicable, have elimination tree and aggregate by subtrees

The irregularity in data bewilders the programmer because it is directly correlated to the irregularity in communication. It is always a main concern for a programmer to reduce communication as much as possible between units of execution. In the case of regular neighbor-to-neighbor communication pattern, partition strategy might be straightforward. But among such irregularities in communication observed in event-driven pattern, is there any guideline at all?

Although not applicable to all domains of event-driven pattern, a techinique called “symbolic computation” can be used to predict the outcome of the computation. In many cases, it is possible to obtain the internal structure of the output without knowing the actual numerical values of the solution. Symbolic computation usually takes an order of magnitude lower than the real computation and thus can be used to obtain tree-like dependency graph for the partitions, also known as elimination tree.

Graph partitioning is known to be NP-hard, but there are many approximation algorithms that achieve optimal solution within a good factor. Partitioning the domain according to the elimination tree significantly reduces the communication frequencies across the partitions. Namely, independent subtree and weakly connected nodes are a good candidates for partitioning borders.

5. Tune load imbalance by mapping heuristics

Due to irregularities in data, load imbalance is unavoidable in event-driven pattern. Applying different heuristic information to assign task mappings to units of execution might mediate the problem. Heuristics can vary depending on the type of the problem. Commonly used heuristics include sorting task with the elimination tree depth or workload that each partition has.

6. Cache optimization locally.

Many techniques. Ordering, Chunking, Temporal, Spatial….