title: Communication Computation Overlap

layout: default

Consider the local MPI computation below

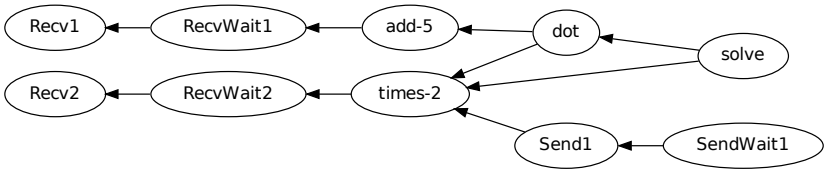

Figure 1: A computation

Each node in this graph is some task that must be performed. The graph contains computations `add-5`, `times-2`, `dot`, `solve` and communication `Recv1`, `RecvWait1`, etc.... The edges specify dependencies between tasks. A job may not be run until all jobs to which it points are completed. Some jobs are communication jobs. `Recv` and `Send` are asynchronous and finish immediately, their corresponding `Wait` jobs block until the transmission is complete.

On a sequential machine we need to run these jobs in some order. A valid order can be found by performing a topological sort on the graph. Note that there are many valid orders. Two are produced below.

```
Order 1: R1 RW1 R2 RW2 *2 +5 dot solve S1 SW1

Order 2: R2 R1 RW2 *2 S1 RW1 +5 dot solve SW1
```

In a purely sequential environment the order is inconsequential as long as it satisfies the dependences of the DAG. Because we have asynchronous operations `S1, R1, R2` the order does matter. In order 1 notice that `Send1` (`S1`) is called just before `SendWait1` (`SW1`). We start and then wait on the transfer. In order 2 `S1` and `SW1` are separated by the operations `RW1, +5, dot, solve`. In this ordering the communication happens in the background while these computational tasks execute. Due to this overlap the runtime of the wait node `SW1` will be shorter, reducing the total runtime of the computation.

Order 1 is the naive ordering. Order 2 attempts to maximize communication computation overlap. How can we obtain orderings like 2 given a DAG?

Satisfying Multiple DAGs

To maximize communication-computation overlap we want to order our nodes so that we do all of our asynchronous Send/Recv starts, then all of our computation, then finally all of our waits. In some sense we want to enforce a DAG that looks like this.

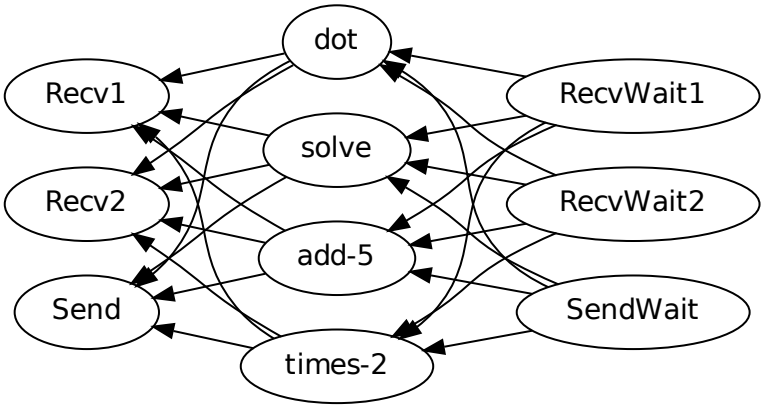

Figure 2: Communication-Computation overlap DAG

while still satisfying our original data dependence DAG

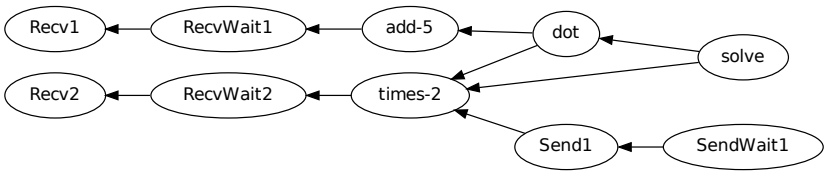

Figure 3: Data dependence DAG

In general this is impossible. For example notice that in the communication overlap DAG `times-2` depends on `S1` but that in the data dependence DAG `S1` depends on `times-2`. The desire that sends happen before computation must defer to the need to maintain data dependence (`Send` is sending the results of `times-2` so it must come afterwards). In general the communication overlap

DAG defers to the data dependence DAG.

We may choose to add more DAGs representing different desires. For example we may choose to execute MPI Sends and Recvs with lower tags first. If all compute nodes adhere to this convention then we remove potential deadlock situations.

The set of DAGs is ordered into a sequence by priority. The data dependence DAG comes first because it is essential. The other DAGs are less important but may be ordered among themselves. We try include as many edges as possible, assuming that these edges do not conflict with higher-ranked DAGs.

Given a sequence of DAGs ordered by importance we construct a consensus DAG which contains as many edges as possible from all of the DAGs, preferring those earlier in the sequence to those later in the sequence. We perform a topological sort on the final conglomerate DAG to obtain an ordering. This DAG has more edges/dependencies and so is less ambiguous. There are relatively fewer correct orderings for the conglomerate DAG.

Partial Orders and Comparator functions

How do we input DAGs from scientific users?

A DAG is equivalent to a partial order. A partial order can be completely described by a comparator function. Comparator functions are widely known in programming communities and serve as a good interface between scientific programmers and scheduling algorithms.

```
def dependence(a, b):
    """ A cmp function for nodes in a graph - does a depend on b?

    Returns positive number if a depends on b
    Returns negative number if b depends on a
    Returns 0 otherwise
    """
    if depends((a, b)): return  1
    if depends((b, a)): return -1
    return 0
```

The communication/computation overlap dag shown above is highly symmetric. It consists of three groups. Each element within a group is ordered equivalently to the others in the group and has a clear ordering relative to the others. This pattern is indicative of a *total order*. Total orders are equivalent to sort-key functions. The sort-key function for the communication-computation overlap is defined below.

```
def mpi_send_wait_key(a):
    """ Wait as long as possible on Waits, Start Send/Recvs early """
    if isinstance(a.op, (MPIRecvWait, MPISendWait)):
        return 1
    if isinstance(a.op, (MPIRecv, MPISend)):
        return -1
    return 0
```

and the key for "lower tags first" is here

```
def mpi_tag_key(a):
    """ Break MPI ties by using the variable tag - prefer lower tags first """
    if isinstance(a.op, (MPISend, MPIRecv, MPIRecvWait, MPISendWait)):
        return a.op.tag
    else:
        return 0
```

Once we have a general system to convert a sequence of comparator functions into an ordering the asynchronous communication problem was easy. We have reduced the problem of ordering computations to maximize communication/computation overlap and eliminate deadlocks to these three small functions, each of which is accessible to a programmer who is not familiar with DAG scheduling.

Conclusion

We posed the communication-computation overlap problem and discussed why it is important. We then saw how this problem could be generalized to the problem of merging a sequence of inconsistent DAGs. We finished by describing the overlap and deadlock problems with three simple comparator functions.

This approach is valuable because it separates the desired policy from the algorithm of how that policy is enacted. This gives the applications programmer the ability to declaratively describe ordering policies without bothering with graph traversals. This allows for rapid prototyping and safer development.

While we started with an example in asynchronous communication this same concept can be used to describe any policy which can be fully described with a DAG.