TACC Technical Report IMP-06

Definition of a ‘communication avoiding’ compiler in the Integrative Model

Victor Eijkhout*

September 7, 2015

This technical report is a preprint of a paper intended for publication in a journal or proceedings. Since changes may be made before publication, this preprint is made available with the understanding that anyone wanting to cite or reproduce it ascertains that no published version in journal or proceedings exists.

Permission to copy this report is granted for electronic viewing and single-copy printing. Permissible uses are research and browsing. Specifically prohibited are sales of any copy, whether electronic or hardcopy, for any purpose. Also prohibited is copying, excerpting or extensive quoting of any report in another work without the written permission of one of the report’s authors.

The University of Texas at Austin and the Texas Advanced Computing Center make no warranty, express or implied, nor assume any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed.

* eijkhout@tacc.utexas.edu, Texas Advanced Computing Center, The University of Texas at Austin
Abstract

IMP distributions are defined with respect to abstract processing entities, leading to a concept of tasks, rather than processes. In a past note we defined processors, and describe their interaction as it arises from the task dataflow. In this note we extend that story, showing that certain arrangements of the task graph over processors leads to a communication minimizing and latency hiding behaviour.

The following IMP reports are available or under construction:

**IMP-01** IMP Distribution Theory
**IMP-02** The deep theory of the Integrative Model
**IMP-03** The type system of the Integrative Model
**IMP-04** Task execution in the Integrative Model
**IMP-05** Processors in the Integrative Model
**IMP-06** Definition of a ‘communication avoiding’ compiler in the Integrative Model
**IMP-07** Associative messaging in the Integrative Model (under construction)
**IMP-08** Resilience in the Integrative Model (under construction)
**IMP-09** Tree codes in the Integrative Model
**IMP-10** Thoughts on models for parallelism
**IMP-11** A gentle introduction to the Integrative Model for Parallelism
**IMP-12** K-means clustering in the Integrative Model
**IMP-13** Sparse Operations in the Integrative Model for Parallelism
**IMP-14** 1.5D All-pairs Methods in the Integrative Model for Parallelism (under construction)
**IMP-15** Collectives in the Integrative Model for Parallelism
**IMP-16** Processor-local code generation (under construction)
**IMP-17** The CG method in the Integrative Model for Parallelism (under construction)
**IMP-18** A tutorial introduction to IMP software (under construction)
1 Motivation

In IMP-05 we defined processors as subsets of the dataflow task graph that is basic in the IMP model. (See IMP-01 for the derivation of this task graph.)

2 Latency hiding and communication avoiding

In Partial Differential Equation (PDE) methods, a repeated sequence of sparse matrix-vector products is a regular occurrence. Typically, the sparse matrix can best be viewed as an operator on a grid of unknowns, where a new value is some combination of values of neighbouring unknowns. In a parallel context this means that in order to evaluate the matrix-vector product $y \leftarrow Ax$ on a processor, that processor needs to obtain the $x$-values of its ghost region. Under reasonable assumptions on the partitioning of the domain over the processors, the number of messages involved will be fairly small: in a Finite Element Method (FEM) or Finite Difference Method (FDM) context, the number of messages is $O(1)$ as $h \downarrow 0$.

Since there is little data reuse, and in the sparse case not even spatial locality, it is normally concluded that the sparse product is largely a bandwidth-bound algorithm. Looking at just a single product there is not much we can do about that. However, if a number of such products is performed in a row, for instance as the steps in a time-dependent process, there may be rearrangements of the operations that lessen the bandwidth demands, typically by lessening the latency cost.

Consider as a simple example

$$\forall i: x_i^{(n+1)} = f(x_i^{(n)}, x_{i-1}^{(n)}, x_{i+1}^{(n)})$$

(1)

and let’s assume that the set $\{x_i^{(n)}\}_i$ is too large to fit in cache. This is a model for, for instance, the explicit scheme for the heat equation in one space dimension. Schematically:

$$\begin{array}{c}
\uparrow \quad \quad \uparrow \quad \quad \uparrow \\
X_0^{(n+1)} \quad X_1^{(n+1)} \quad X_2^{(n+1)} \\
\downarrow \quad \quad \downarrow \quad \quad \downarrow \\
X_0^{(n+2)} \quad X_1^{(n+2)} \quad X_2^{(n+2)}
\end{array}$$

In the ordinary computation, where we first compute all $x_i^{(n+1)}$, then all $x_i^{(n+2)}$, the intermediate values at level $n + 1$ will be flushed from the cache after they were generated, and then brought back into cache as input for the level $n + 2$ quantities.

However, if we compute not one, but two iterations, the intermediate values may stay in cache. Consider $x_0^{(n+2)}$: it requires $x_0^{(n+1)}, x_1^{(n+1)}$, which in turn require $x_0^{(n)}, \ldots, x_2^{(n)}$. 

IMP-06 1
Now suppose that we are not interested in the intermediate results, but only the final iteration. Figure 1 shows a simple example. The first processor computes 4 points on level \(n+2\). For this it needs 5 points from level \(n+1\), and these need to be computed too, from 6 points on level \(n\). We see that a processor apparently needs to collect a ghost region of width two, as opposed to just one for the regular single step update. One of the points computed by the first processor is \(x_3^{(n+2)}\), which needs \(x_4^{(n+1)}\). This point is also needed for the computation of \(x_4^{(n+2)}\), which belongs to the second processor.

The easiest solution is to let this sort of point on the intermediate level redundantly computed, in the computation of both blocks where it is needed, on two different processors.

- First of all, as we motivated above, doing this on a single processor increases locality: if all points in a coloured block (see the figure) fit in cache, we get reuse of the intermediate points.
- Secondly, if we consider this as a scheme for distributed memory computation, it reduces message traffic. Normally, for every update step the processors need to exchange their boundary data. If we accept some redundant duplication of work, we can now eliminate the data exchange for the intermediate levels. The decrease in communication will typically outweigh the increase in work.

### 2.1 Analysis

Let’s analyze the algorithm we have just sketched. As in equation (1) we limit ourselves to a 1D set of points and a function of three points. The parameters describing the problem are these:

- \(N\) is the number of points to be updated, and \(M\) denotes the number of update steps. Thus, we perform \(MN\) function evaluations.
- \(\alpha, \beta, \gamma\) are the usual parameters describing latency, transmission time of a single point, and time for an operation (here taken to be an \(f\) evaluation).
- \(b\) is the number of steps we block together.
Each halo communication consists of $b$ points, and we do this $\sqrt{N}/b$ many times. The work performed consists of the $MN/p$ local updates, plus the redundant work because of the halo. The latter term consists of $b^2/2$ operations, performed both on the left and right side of the processor domain.

Adding all these terms together, we find a cost of

$$\frac{M}{b} \alpha + M \beta + \left( \frac{MN}{p} + Mb \right) \gamma.$$

We observe that the overhead of $\alpha M/b + \gamma Mb$ is independent of $p$. Note that the optimal value of $b$ only depends on the architectural parameters $\alpha, \beta, \gamma$ but not on the problem parameters.

### 2.2 Communication and work minimizing strategy

We can make this algorithm more efficient by overlapping the communication and computation. As illustrated in figure 2, each processor start by communicating its halo, and overlapping this communication with the part of the communication that can be done locally. The values that depend on the halo will then be computed last.

![Figure 2: Computation of blocks of grid points over multiple iterations](image)

If the number of points per processor is large enough, the amount of communication is low relative to the computation, and you could take $b$ fairly large. However, these grid updates are mostly used in iterative methods such as the Conjugate Gradients (CG) method, and in that case considerations of roundoff prevent you from taking $b$ too large[1].

A further refinement of the above algorithm is possible. Figure 3 illustrates that it is possible to use a halo region that uses different points from different time steps. This algorithm (see [2]) cuts down on the amount of redundant computation. However, now the halo values that are communicated first need to be computed, so this requires splitting the local communication into two phases.

### 3 Processor synchronization

We recapitulate the definitions of processors and synchronization from IMP-05.
Let \( \{C_p\}_p \) be a covering of the set of tasks. Then we define:

**Definition 1** For a task \( t \in T \) we define a task \( t' \) as a synchronization point if \( t' \) is an immediate predecessor on another processor:

\[
t \in C_p \land t' < t \land t' \in C_{p'} \land p \neq p'.
\]

**Definition 2** Given a set of tasks \( L \subset T \), we define its base \( B_L \) as

\[
B_L = \{t \in L: \ pred(t) \not\subset L\}.
\]

**Definition 3** We call a two-parameter covering \( \{L_{k,p}\}_{k,p} \) of \( T \) a set of local computations if

1. the \( p \) index corresponds to the division in processors:

\[
C_p = \bigcup L_{k,p}.
\]

2. the \( k \) index corresponds to the partial ordering on tasks: the sets \( L_k = \bigcup_p L_{k,p} \) satisfy

\[
t \in L_k \land t' < t \Rightarrow t' \in \bigcup_{\ell \leq k} L_{\ell}
\]

3. the synchronization points synchronize only with previous levels:

\[
\text{pred}(B_{k,p}) - C_p \subset \bigcup_{\ell < k} L_{\ell}
\]

We illustrate this in figure 4. Case (a) is the normal single step grid update, much like our motivating example. Case (b) shows that for a second update we would need a point on the same \( k \)-level, so this is not a well-formed local computation by the above definition. Case (c) shows how this is solved by transferring a larger halo, and computing one point redundantly.
4 A ‘communication avoiding’ compiler

In the third subfigure of figure 4 we showed the traditional strategy of communication a larger halo than would be strictly necessary [3, 4, 5]. With this, and some redundant computation, it is possible to remove a synchronization point from the computation.

However, this is not guaranteed to overlap communication and computation; also, it is possible to avoid some of the redundant work. We will now formalize this ‘communication avoiding’ strategy [2].

Let $L_{k,p}$ be a collection of local computations. We assume that it is not a well-formed collection in the sense of definition 3. We will now split $k$ in three substeps, $k_1, k_2, k_3$, giving us a splitting that is well-formed, and that has overlap of computation and communication.

We also need temporary sets $L_{k_1,p}, L_{k_3,p}$.

1. Let $L_{k_4,p}$ be the tasks in $L_{k,p}$ that can be computed from local data:

$$\text{pred}(L_{k_4,p}) \subset L_{k_4,p} \cup L_{k_5,p} \cup \cdots.$$
2. Let $L_{k, p}$ be the tasks in $L_k$ that have successors in $L_{k+1}$, plus their transitive closure in $L_k$ under predecessorship. This set is what is normally computed redundantly if a transitive augmentation of the halo is communicated.

3. Let $q$ a processor be such that $L_{k, q} \cap L_{k, p} \neq \emptyset$, and define

$$L_{k, q} = L_{k, p} \cap L_{k, q}.$$ 

These tasks can be computed locally on processor $q$ without synchronization, and they contributed to the computation of $L_{k, p}$. Similarly, $p$ constructs the set $L_{k, p}$ that will contribute to neighbour processors of $p$. However, note that determining the sets will take communication between tasks $p, q$.

4. Define

$$L_{k, p} = L_{k, p} - L_{k, q}.$$ 

These are also tasks that can be computed locally, but unlike $L_{k, p}$ have no transitive predecessor relation to any $L_{k, q}$ for $q \neq p$.

5. Finally, define

$$L_{k, p} = L_{k, p} - \bigcup_q L_{k, q}.$$ 

This set contains tasks both in $C_p$ and other processors.

**Theorem 1** The splitting $L_{k, p}, L_{k, p}, L_{k, p}$ is well-formed and has overlap of communication $L_{k, p} \rightarrow L_{k, p}$ with the computation of $L_{k, p}$. Neither $L_{k, p}$ nor $L_{k, p}$ have synchronization points, so the whole algorithm has overlap.

However, note that $L_{k, p} \cup L_{k, p} \cup L_{k, p}$ can be larger than $L_{k, p}$, corresponding to redundant calculation.

**Write out predecessor relations with proof**

5 Implementation

In a simple version of communication minimizing, we compute explicitly the beta distribution of a second kernel:

```c
signature_function *f = new signature_function();
f->add_beta_oper( noop );
f->add_beta_oper( left );
f->add_beta_oper( right );
extend = new mpi_distribution
   ( env, f->derive_beta_structure(block,block) );
```
and use it as gamma distribution of a first.

```c
object *v0,*v1,*v2;
v0 = new mpi_object(block);
v1 = new mpi_object(extend);
v2 = new mpi_object(block);
kernel *k1,*k2;
k1 = new mpi_kernel( v0,v1 );
k1->add_beta_oper( noop );
k1->add_beta_oper( left );
k1->add_beta_oper( right );
k2 = new mpi_kernel( v1,v2 );
k2->add_beta_oper( noop );
k2->add_beta_oper( left);
k2->add_beta_oper( right );
```

More sophisticatedly,

1. Find non-local dependencies
2. Make new distribution with their output locally reproduced
3. This distribution has more tasks locally
4. The predecessors of a task are now no longer unique; look locally first.

**References**


