TACC Technical Report IMP-06

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

Unpublished preliminary report. Do not disseminate.

Victor Eijkhout*

May 1, 2017

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-00 The IMP Elevator Pitch
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 (under construction)
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 (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)
IMP-20 A mathematical formalization of data parallel operations
IMP-21 Adaptive mesh refinement (under construction)
IMP-22 Implementing LULESH in IMP (under construction)
IMP-23 Distributed computing theory in IMP (under construction)
IMP-24 IMP as a vehicle for software/hardware co-design, with John McCalpin (under construction)
IMP-25 Dense linear algebra in IMP (under construction)
IMP-26 Load balancing in IMP (under construction)
IMP-27 Data analytics in IMP (under construction)
1 Motivation

On clusters the cost of communication can be high relatively to the cost of computation. Hence, the overlapping computation and communication (also known as latency hiding) has long been a goal of parallel programming. On shared memory processors with explicitly managed scratchpad memory there is an equivalent phenomenon: if data can be pushed to the scratchpad well in advance of it being needed, we now hide the memory, rather than network, latency.

Various techniques for latency hiding have been used. For instance, the PETSc library [10] splits the matrix-vector product in local and non-local parts, so that the former can overlap the communication of the latter. Related, redundant computation in order to avoid communication is an old idea [12].

There is considerable work in the context of iterative methods for linear system to mitigate the influence of communication.

- Reformulation of CG-like methods to reduce the number of inner products [1, 2, 13, 11].
- Multi-step methods that combine inner products, and can have better locality properties [1].
- Overlapping either the preconditioner application or the matrix-vector product with a collective [3].

We will give a new variant, based on [9], that overlaps both.

Recently, the notion of redundant computation we revisited by Demmel et al. [4], in so-called ‘communication avoiding’ methods. We will show how such methods naturally arise in the IMP framework. This will be the main result of this note.
2 Algorithmic latency hiding

2.1 Orchestrating data transfer

In figure 1 we depict the logical structure of data transfer: kernel $k_1$ produces data (specifically, in processor $q$), which is needed in kernel $k_2$ (specifically, in processor $q$). (While the picture makes the kernel tasks look synchronized, that is, all tasks in $k_1$ happening before $k_1$, this need not be the case in practice.)

Actually realizing this conceptual picture in practice is not trivial. We know that kernel $k_2$ takes a certain object as input, so the easiest implementation posts both the sends and receives for that object as a first step in $k_2$. This is depicted in figure 2. However, we would like to post the send/receive operations earlier, so that we can take advantage of possible offloaded communication.

The optimized solution is then that the descriptors for the send/receive operations are moved from $k_2$ which creates them, to $k_1$ where can can earliest be executed. This means that the transfer can overlap with any intervening kernel $k_3$. It also means that the space available for $k_3$ is diminished by buffer space for the $k_1 \rightarrow k_2$ transfer. See figure 3.

2.2 Overlap

We analyze the task graph to find operations that can potentially overlap. For instance, in pipelined Conjugate Gradients [7] the matrix-vector product and one inner product are seen to be causally unrelated, hence overlappable; figure 4.

Such an analysis is NP-complete in the general case [14], but becomes very efficient in the parallelism model of IMP.
We explain the basic idea of the ‘communication avoiding’ scheme. This was originally proposed for iterative methods, such as \( s \)-step CG; in the next section we will show that the IMP framework can realize this in general.

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 \( y \leftrightarrow Ax \) on a processor, that processor needs to obtain the \( x \)-values of its \textit{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 \( h \rightarrow 0 \).

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 \textit{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

\[
\forall i: x^{(n+1)}_i = f(x^{(n)}_i, x^{(n)}_{i-1}, x^{(n)}_{i+1})
\]  

(1)

IMP-06
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:

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, \( x_2^{(n)} \).

Now suppose that we are not interested in the intermediate results, but only the final iteration. Figure 5 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.

Figure 5: Computation of blocks of grid points over multiple iterations
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.

3.1 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.

3.2 Communication and work minimizing strategy

We can make this algorithm more efficient by overlapping the communication and computation. As illustrated in figure 6, 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.

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 7 illustrates that it is possible to use a halo region that uses different points from different time steps. This algorithm (see [4]) 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.
4 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** We define a task \( t \in T \) to be a synchronization point if it has an immediate predecessor on another processor:

\[
t \in C_p \land \exists t' \in C_{p'}: t' \in \text{pred}(t), \quad \text{where } p \neq p'.
\]

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

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

If \( \{L_{k,p}\}_{k,p} \) is a two-parameter covering of \( L \), we similarly define \( B_{k,p} \):

\[
B_{k,p} = \{ t \in L_{k,p}: \text{pred}(t) \not\subseteq L_{k,p} \}.
\]
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, again, not necessarily disjoint:
   \[
   C_p = \bigcup_k L_{k,p}.
   \]
2. the synchronization points synchronize only with previous levels:
   \[
   \text{pred}(B_{k,p}) - C_p \subset \bigcup_{\ell < k} L_{\ell}
   \] (2)

We illustrate this in figure 8. 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.

5  A ‘communication avoiding’ framework

We now show how the above scheme, proposed for iterative methods, can be applied to general task graphs. This means that we can have a ‘communication avoiding compiler’, that turns an arbitrary computation into a communication avoiding one.

In the third subfigure of figure 8 we showed the traditional strategy of communication a larger halo than would be strictly necessary [5, 6, 12]. 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 [4].

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 \( L_{k_5,p}, L_{k_5,p} \).
1. Let $L_{k_4,p}$ be the tasks in $L_{k,p}$ that can be computed from local data:

$$\text{pred}(L_{k_4,p}) \subset L_{k,p} \cup L_{k-1,p} \cup \cdots.$$ 

2. Let $L_{k_5,p}$ be the tasks in $L_{k,p}$ 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 $L_{k_5,p} \cap L_{k,q} \neq \emptyset$, and define

$$L_{k_1,q} = L_{k_5,p} \cap L_{k_4,q}.$$ 

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 $L_{k_1,p}$ that will contribute to neighbour processors of $p$. Since they are in $k_4$, these tasks can be computed locally; however, determining the sets will take communication between tasks $p,q$.

4. Define

$$L_{k_2,p} = L_{k_4,p} - L_{k_1,p}.$$ 

These are also tasks that can be computed locally, but unlike $L_{k_1}$ have no transitive predecessor relation to any $L_{k,q}$ for $q \neq p$.

5. Finally, define

$$L_{k_3,p} = L_{k_5,p} \cup L_{k_1,q}.$$ 

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

**Theorem 1** The splitting $L_{k_1,p}, L_{k_2,p}, L_{k_3,p}$ is well-formed and has overlap of communication $L_{k_1} \rightarrow L_{k_3}$ with the computation of $L_{k_2}$. Neither $L_{k_1}$ nor $L_{k_2}$ have synchronization points, so the whole algorithm has overlap.
However, note that $L_{k_1,p} \cup L_{k_2,p} \cup L_{k_3,p}$ can be larger than $L_{k,p}$, corresponding to redundant calculation.

![Figure 10: Communicated sets in the communication avoiding scheme](image)

We spell out the synchronization points and bases in figure 10:

- $k_0$ is the result of the previous operation; we use it as basis for computing $k_1, k_2$.
- $k_7$ is the subset of $k_1$ that is communicated.
- $k_6$ is how $k_7$ is received: it contains the synchronization points of $k_3$.

### 6 Example: split computation of matrix-vector product

Let’s revisit the example of a one-dimensional heat equation that we started this story with. (This will also stand for a single sparse matrix-vector product.) In figure 11 we overlay one step of a heat equation on the previous definition of a communication avoiding scheme. However, since we need to block kernels to be able to derive the $k_{1,2,3}$ regions, we do the following:

1. We introduce a first kernel which establishes the beta distribution of the heat update / matrix vector product. In IMP terms this is a copy operation between $\alpha_{spmvp}$ and $\beta_{spmvp}$.
2. We let the actual heat update / matrix-vector product go between $\beta_{spmvp}$ as input distribution and $\gamma_{spmvp}$ as output. This is a local operation!

Now we have two blocked kernels and we can analyze a communication avoiding scheme as described above. We find that:

1. $k_1$ are the points that need to be sent as halo data. Computing these is of course trivial: we already have them.
2. Next we send off our $k_1$ elements, and post a receive for the ones from neighbour processors.
3. Then we compute $k_2$, the local part of the update/product.
4. After receiving the remote $k_1$ sets we can compute the missing parts of the update/product.
Figure 11: Split computation of the sparse matrix-vector product, in our communication avoiding framework.

Of course, this analysis is easy to do in the abstract. In code it would require that each point of the vector-to-be-updated is a single task. Alternatively, we need a code splitting mechanism that is outside of the scope of our work for now. We briefly address this question in [8].

7 DRAM semantics

In section 5 we discussed various approaches to disjoint parallel computing. In all of these, we find a recurring situation where code is executed with some data in local memory (typically L2 cache), and other data is to be streamed from memory.

If the local memory is small enough, as is the case with caches, we can assume that the number of instructions is very finite, and that we can manipulate it as a not-indexed set.

We could approach this as follows:

1. Order the instructions for optimal L1 cache and register use. This can be based on template orderings such as lexicographic traversal, or separating out local and non-local operations.
2. We know which operations are synchronization points, so from the instruction order we find a synchronization order.
3. This determines the DRAM access pattern.

7.1 Prefetch streams

Consider the simple case of evaluating a five-point stencil

$$\forall ij: y_{ij} = f(x_{ij}, x_{i+1,j}, x_{i-1,j}, x_{i,j-1}, x_{i,j+1})$$
on an $m \times n$ grid. Using a lexicographic ordering we can describe the allowable latency on elements in the halo region:

1. bottom $i = 0$: latency $= j$
2. left $j = 0$: latency $= nj$
3. right $j = n$: latency $= n + im$
4. top $i = n$: latency $= (n - 1)m + j$

We posit an $\alpha + \beta t$ model for the allowable latency of element $t$.

- In message passing, $\beta$ is sort of out of our control, since we need all data to be available: there is no streaming model. Thus we only reason about $\alpha$.
- In a shared memory setup, both $\alpha$, $\beta$ are relevant.

Figure 12 depicts how we can apply this delay model to algorithms: given an algorithm with a certain desired structure for when data is needed, we fit $\alpha$, $\beta$ so that the line $y = \alpha + \beta t$ stays under the algorithm curve. Here $\alpha$ is the latency, and $\beta$ is the time per data item, or the inverse of the bandwidth. Note that $\alpha$, $\beta$ are not uniquely determined:

- For a given $\alpha$, lower $\beta$ values are also feasible. This corresponds to keeping latency constant, but increasing the bandwidth.
- If we can lower latency, we can increase the $\beta$ value, that is, lower the bandwidth, and still maintain a best fit.

8 Conclusion

We have discussed the concepts and prior work in latency hiding and communication avoiding. We have shown that the IMP framework can turn an arbitrary computation graph into a communication avoiding one by judicious partitioning of the task graph, and duplicating certain tasks.
References


