TACC Technical Report IMP-06

Definition of a 'communication avoiding' compiler in the Integrative Model

Unpublished preliminary report. Do not disseminate.

Victor Eijkhout*

November 5, 2017

* 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 messsaging 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-19 Report on NSF EAGER 1451204.
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 [11] 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 [13].

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, 14, 12].
- 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 [10], 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.)

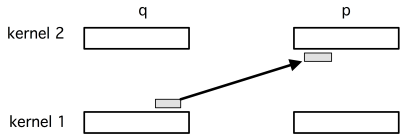


Figure 1: Depiction of task (k_1, q) sending data to (k_2, p)

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.

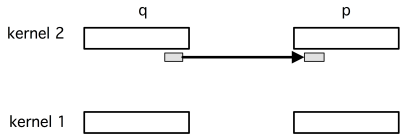


Figure 2: Data transfer as part of k_2

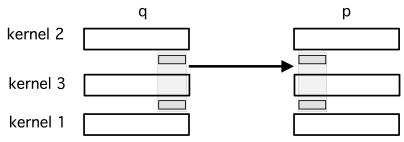


Figure 3: Buffer space detracts from available space for intervening kernel k_3

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 [15], but becomes very efficient in the parallelism model of IMP.

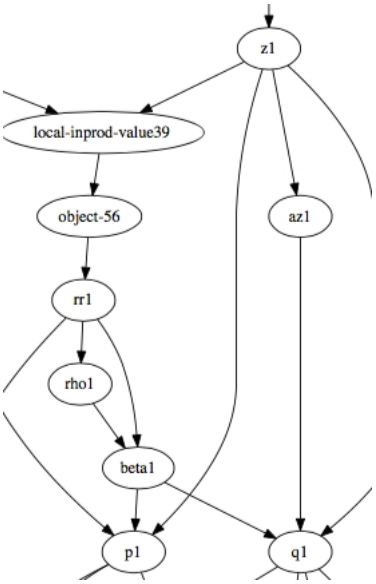


Figure 4: Matrix-vector and vector-vector inner product in the kernel structure of pipelined CG

3 Communication avoiding

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.

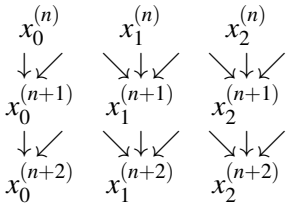
In Partial Differential Equation (PDE) methods, a repeated sequence of sparse matrix-vector products is a regular occurrence. Typically, the sparse matrix can best be viewed as an operator on a grid of unknowns, where a new value is some combination of values of neighbouring unknowns. In a parallel context this means that in order to evaluate the matrix-vector product $y \leftarrow Ax$ 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

$$\forall_i \colon x_i^{(n+1)} = f\left(x_i^{(n)}, x_{i-1}^{(n)}, x_{i+1}^{(n)}\right) \tag{1}$$

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:



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

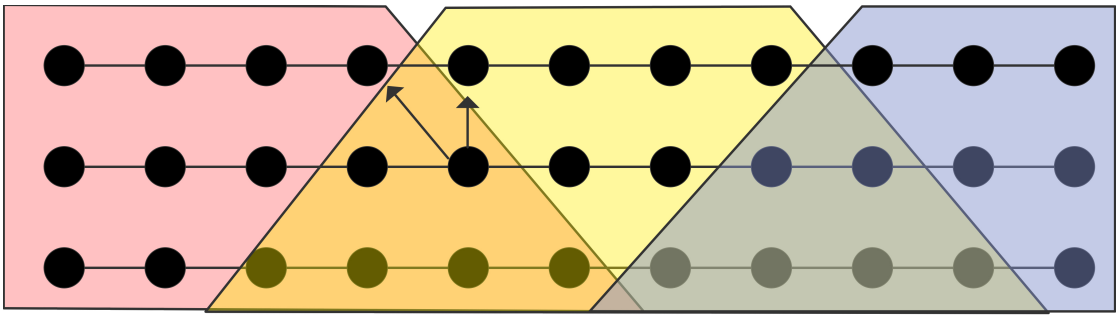


Figure 5: Computation of blocks of grid points over multiple iterations

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.

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.
- α, β, γ 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 α, β, γ 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.

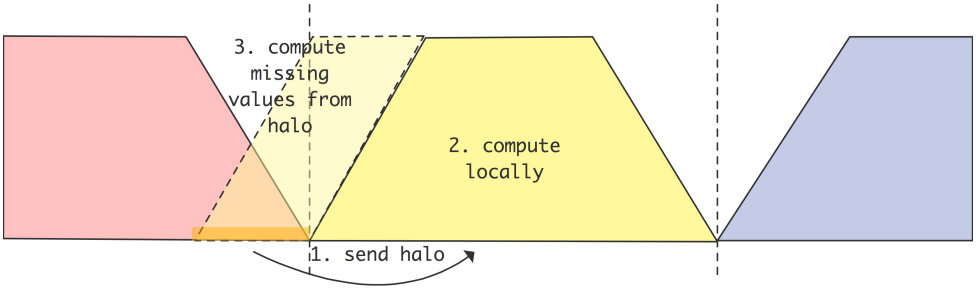


Figure 6: Computation of blocks of grid points over multiple iterations

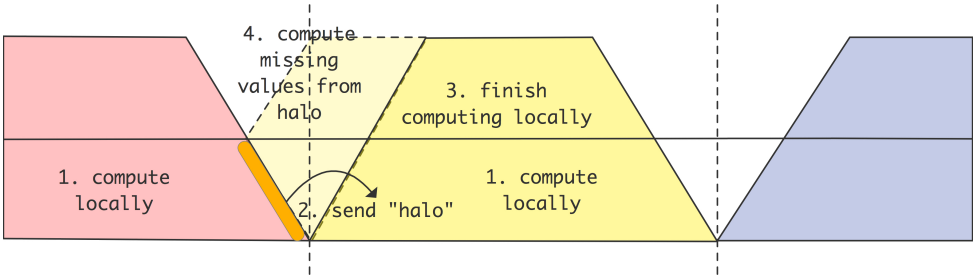


Figure 7: Computation of blocks of grid points over multiple iterations

4 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 **??** we showed the traditional strategy of communication a larger halo than would be strictly necessary [5, 6, 13]. 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 a well-formed collection in the sense of definition **??**. 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.

Subset 0: inherited from previous level In step k we can assume that data from a previous step is available. This can be either a initial condition, or an actually computed step. We define $L^{(0)}_{k,p}$

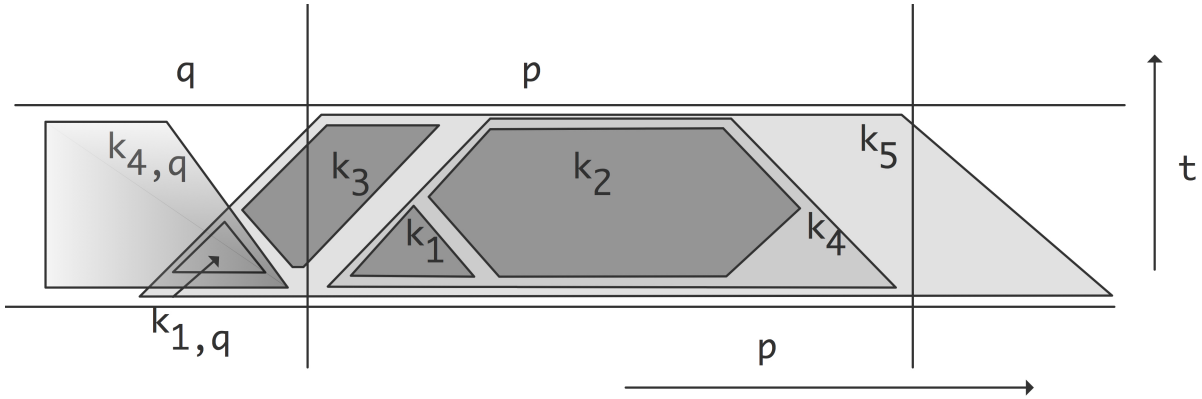


Figure 8: Subdivision of a local computation for minimizing communication and redundant computation.

as the data that is available on process p in step k before any computation takes place in step k:

$$L^{(0)}_{k,p} \equiv L_{k-1,p}$$

Subset 4: all locally computed tasks Starting with $L^{(0)}_{k,p}$, some tasks in $L_{k,p}$ can be computed without needed data from processors $q \neq p$. We term this $L^{(4)}_{k,p}$, but note that this set is an auxiliary in our story; it is not computed as such.

$$L^{(4)}_{k,p} \equiv \left\{t\colon\ pred(t) \in \{L^{(0)}_{k,p} \cup L^{(4)}_{k,p}\}\right\}$$

(This definition is implicit, but well-formed; it is a stand-in for a more complicated explicit one. We hope the reader bears with us.)

Subset 5: all predecessors of local tasks Next we define $L^{(5)}_{k,p}$ as all tasks in L_k that are computed anywhere to construct the local result $L_{k,p}$:

$$L^{(5)}_{k,p} \equiv L_{k,p} \bigcup \big(pred(L_{k,p}) \cup L_k\big)$$

Loosely, this contains all local tasks and the remote ones in the extended halo. Again, this set is an auxiliary, used to defined the following actually computed steps.

Subset 1: locally computed tasks, to be used remotely With the local tasks $L^{(4)}_{k,\star}$ and the needed tasks $L^{(5)}_{k,\star}$, we now define $L^{(1)}_{k,p}$ as the locally computed tasks on p that are needed for a

$q \neq p$:

$$L^{(1)}_{k,p} \equiv L^{(4)}_{k,p} \cup \bigcup_{q \neq p} L^{(5)}_{k,q} - L^{(0)}_{k,p}$$

These are the tasks computed first. For each $q \neq p$, a subset of these elements will be sent to process q, in a communication step that overlaps the computation of the next subset.

Subset 2: locally computed tasks, only used locally While elements of $L^{(1)}_{k,p}$ are being sent, we can do a local computation of the remainder of $L^{(4)}_{k,p}$:

$$L^{(2)}_{k,p} \equiv L^{(4)}_{k,p} - L^{(1)}_{k,p}$$

Subset 3: halo elements and their successors Having received remote elements $L^{(1)}_{k,q}$ from neighbouring processors $q \neq p$, we can construct the remaining elements of $L^{(5)}_{k,p}$ that are needed for $L_{k,p}$:

$$L^{(3)}_{k,p} \equiv L^{(5)}_{k,p} - L^{(4)}_{k,p} - \bigcup_{q \neq p} L^{(1)}_{k,q}$$

We omit definition of the precise elements of $L^{(1)}_{k,\star}$ that are transported; we merely note that this set is considerably smaller than $L^{(1)}_{k,\star}$ itself.

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 $L_{k_1,p} \cup L_{k_2,p} \cup L_{k_3,p}$ is most likely larger than $L_{k,p}$, corresponding to redundant calculation.

We spell out the synchronization points and bases in figure 9:

- 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.

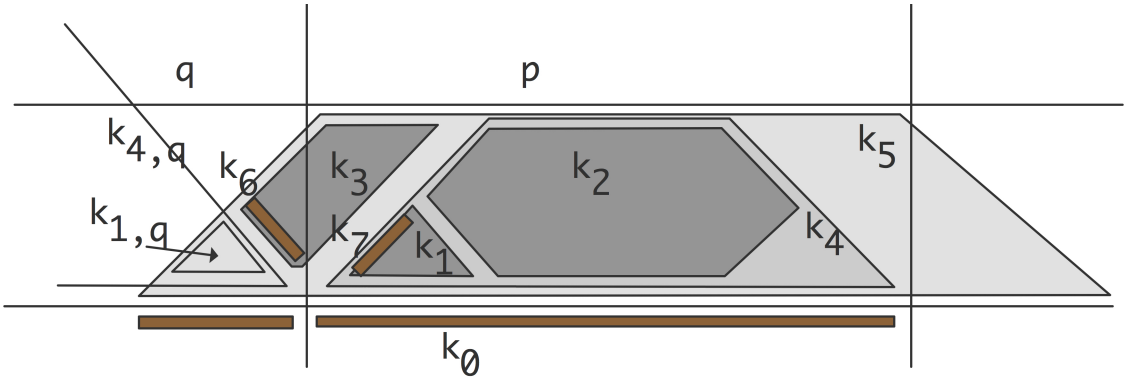
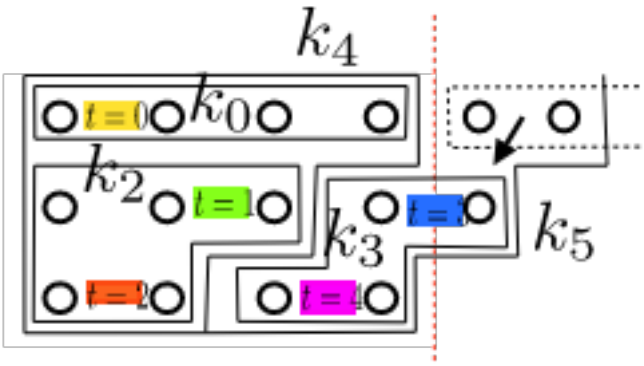


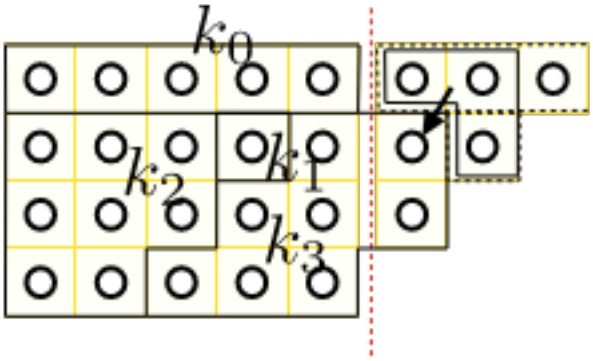
Figure 9: Communicated sets in the communication avoiding scheme

5 Simple examples




```
Graph on node 0, execution of 1 macro steps:
k1 local execution: 0
  parallel time: 0
k3 receive: 2
k2 local execution: 5
  parallel time: 2
k3 local execution: 4
  parallel time: 2
total parallel time : 1*(0+2+2) = 4
overlap analysis:
  2 tasks to recv, 2 k2 parallel time
```

```
Graph on node 1, execution of 1 macro steps:
k1 local execution: 0
  parallel time: 0
k3 receive: 2
k2 local execution: 5
  parallel time: 2
k3 local execution: 4
  parallel time: 2
total parallel time : 1*(0+2+2) = 4
overlap analysis:
  2 tasks to recv, 2 k2 parallel time
```




```
Graph on node 0, execution of 1 macro steps:
k1 local execution: 1
   parallel time: 1
k3 receive: 3
k2 local execution: 8
   parallel time: 3
k3 local execution: 8
   parallel time: 4
 total parallel time : 1*(1+3+4) = 8
 overlap analysis:
   3 tasks to recv, 3 k2 parallel time
```

```
Graph on node 1, execution of 1 macro steps:
k1 local execution: 1
   parallel time: 1
k3 receive: 3
k2 local execution: 8
   parallel time: 3
k3 local execution: 8
   parallel time: 4
 total parallel time : 1*(1+3+4) = 8
 overlap analysis:
   3 tasks to recv, 3 k2 parallel time
```

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 10 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 α_{spmvp} and β_{spmvp}.
2. We let the actual heat update / matrix-vector product go between β_{spmvp} as input distribution and γ_{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.

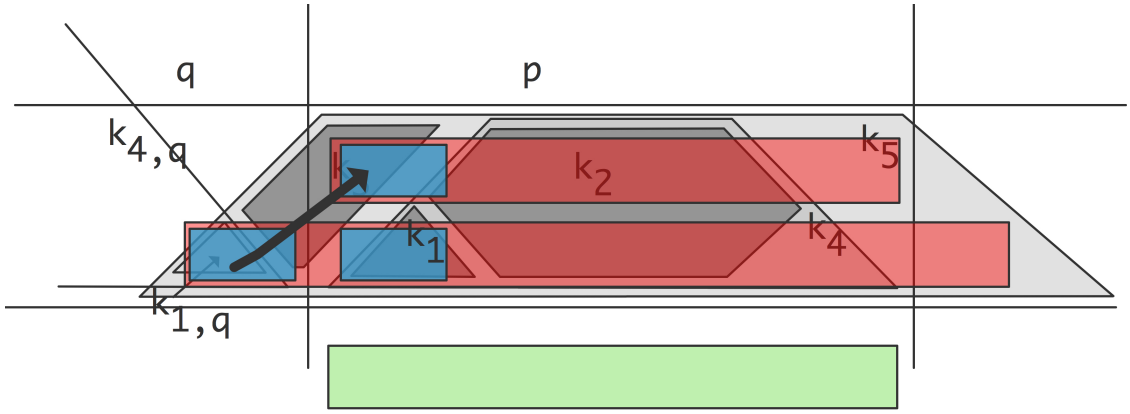


Figure 10: Split computation of the sparse matrix-vector product, in our communication avoiding framework.

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.

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 4 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

$$\forall_{ij}\colon y_{ij} = f(x_{ij}, x_{i+1,j}, x_{i-1,j}, x_{i,j-1}, x_{i,j+1})$$

on an $m \times n$ grid. Using a lexicographic ordering we can 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 messsage passing, β 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 α.
- In a shared memory setup, both α, β are relevant.

Figure 11 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 α, β so that the line $y = \alpha + \beta t$ stays under the algorithm curve. Here α is the latency, and β is the time per data item, or the inverse of the bandwidth. Note that α, β are not uniquely determined:

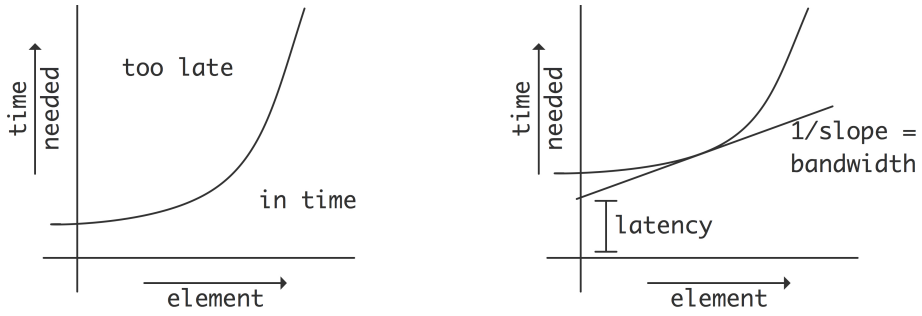


Figure 11: Delay model for prefetch streams

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

8 Simulation

We have written a simple simulator[1] that takes a task graph and identifies the k_1, k_2, k_3 sets. In figure 12 we illustrate these sets for a one-dimensional case, but we note that the analysis works on arbitrary task graphs.

We then simulate parallel runtimes by evaluating runtimes for the following scenario:

1. The code can be found in the code repository [9] under `pocs/avoid`.

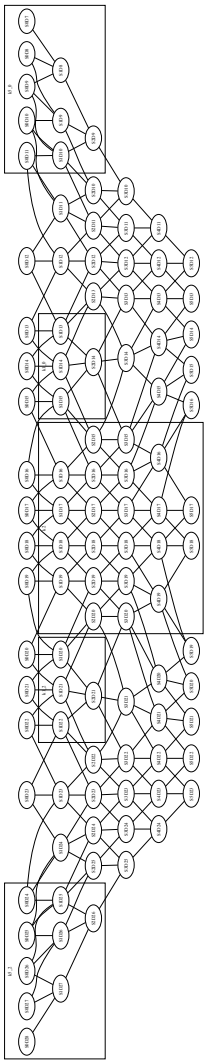


Figure 12: Depiction of k_1, k_2, k_3 sets for a processor doing a 1D heat equation.

- We use a strong scaling scenario where we have a given problem size and partitioning into tasks, as well the number of MPI nodes;
- We set the ratio of message latency to floating point operation as fixed;
- We evaluate the runtime as a function of the number of threads available for the task graph on an MPI node.

The prediction is that, with non-negligible latency, blocking will reduce the running time more than the extra computation time from the extended halo. Also, this effect will be more pronounced as the core count increases, since this reduces the no-node computation time.

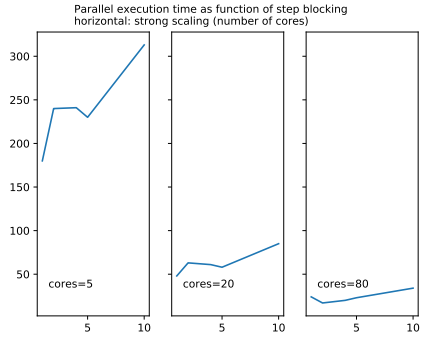


Figure 13: Runtime as a function of core count for moderate latency

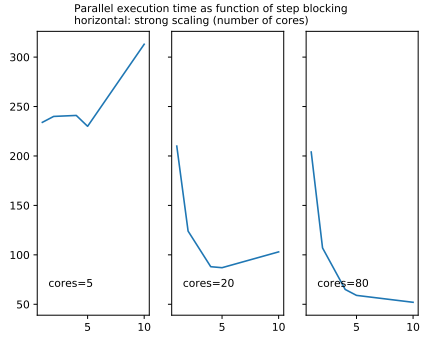


Figure 14: Runtime as a function of core count for high latency

First, in figure 13 we use a low latency; we see that only for very high thread count is there any gain. In figure 14 we use a higher latency, and we see that even for moderate thread counts blocking effects latency hiding.

9 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

[1] A. Chronopoulos and C.W. Gear. *s*-step iterative methods for symmetric linear systems. *Journal of Computational and Applied Mathematics*, 25:153–168, 1989.

[2] E.F. D'Azevedo, V.L. Eijkhout, and C.H. Romine. A matrix framework for conjugate gradient methods and some variants of cg with less synchronization overhead. In *Proceedings of the Sixth SIAM Conference on Parallel Processing for Scientific Computing*, pages 644–646, Philadelphia, 1993. SIAM.

[3] J. Demmel, M. Heath, and H. Van der Vorst. Parallel numerical linear algebra. In *Acta Numerica 1993*. Cambridge University Press, Cambridge, 1993.

[4] James Demmel, Mark Hoemmen, Marghoob Mohiyuddin, and Katherine Yelick. Avoiding communication in sparse matrix computations. In *IEEE International Parallel and Distributed Processing Symposium*, 2008.

[5] C. Douglas. Caching in multigrid algorithms: problems in two dimensions. *International Journal of Parallel, Emergent and Distributed Systems*, 9:195–?204, 1996.

[6] Victor Eijkhout. Polynomial acceleration of optimised multi-grid smoothers; basic theory. Technical Report UT-CS-02-477, Innovative Computing Lab, University of Tennessee Knoxville, August 2002.

[7] Victor Eijkhout. The CG method in the Integrative Model for Parallelism. Technical Report IMP-17, Integrative Programming Lab, Texas Advanced Computing Center, The University of Texas at Austin, 2014. (Report under construction. Included in source code repository.).

[8] Victor Eijkhout. Processor-local code generation (under construction). Technical Report IMP-16, Integrative Programming Lab, Texas Advanced Computing Center, The University of Texas at Austin, 2014.

[9] Victor Eijkhout. IMP code repository, 2014-7. https://bitbucket.org/VictorEijkhout/imp-demo.

[10] Bill Gropp. Update on libraries. http://jointlab-pc.ncsa.illinois.edu/events/workshop3/pdf/presentations/Gropp-Update-on-Libraries.pdf.

[11] W. D. Gropp and B. F. Smith. Scalable, extensible, and portable numerical libraries. In *Proceedings of the Scalable Parallel Libraries Conference, IEEE 1994*, pages 87–93.

[12] Gerard Meurant. Multitasking the conjugate gradient method on the CRAY X-MP/48. *Parallel Computing*, 5:267–280, 1987.

[13] Thomas Oppe and Wayne D. Joubert. Improved SSOR and incomplete Cholesky solution of linear equations on shared memory and distributed memory parallel computers. *Numerical Linear Algebra with Applications*, 1:287–311, 1994.

[14] Yousef Saad. Practical use of some krylov subspace methods for solving indefinite and non-symmetric linear systems. *SIAM J. Sci. Stat. Comput.*, 5:203–228, 1984.

[15] R.N. Taylor. Complexity of analyzing the synchronization structure of concurrent programs. *Acta Inf.*, 19:57?–84, 1983.