Task Graph Transformations for Latency Tolerance

Victor Eijkhout*

November 12, 2018

Abstract

The Integrative Model for Parallelism (IMP) derives a task graph from a higher level description of parallel algorithms. In this note we show how task graph transformations can be used to achieve latency tolerance in the program execution. We give a formal derivation of the graph transformation, and show through simulation how latency tolerant algorithms can be faster than the naive execution in a strong scaling scenario.

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].

In this report we will show that latency hiding, and general latency tolerance, can be achieved by task graph transformations. For this we use the formalism of the Integrative Model for Parallelism (IMP), previously defined in [8].

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

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

- 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 was 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 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.

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 1 shows a simple example. The first processor computes

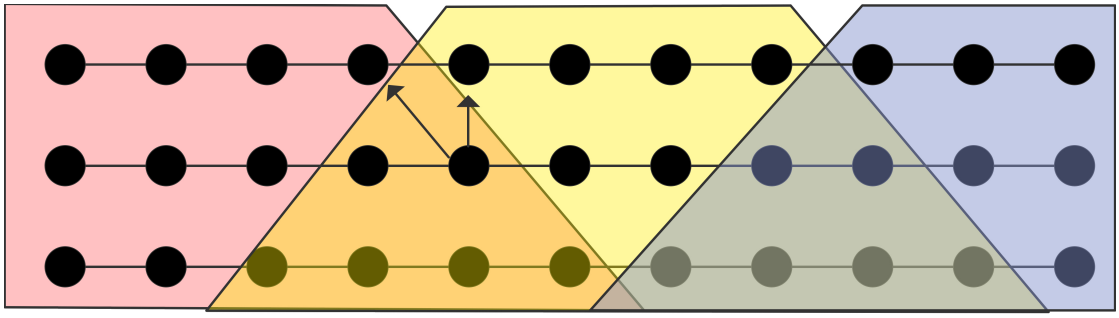

Figure 1: Computation of blocks of grid points over multiple iterations

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.
- 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.

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.

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 [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.

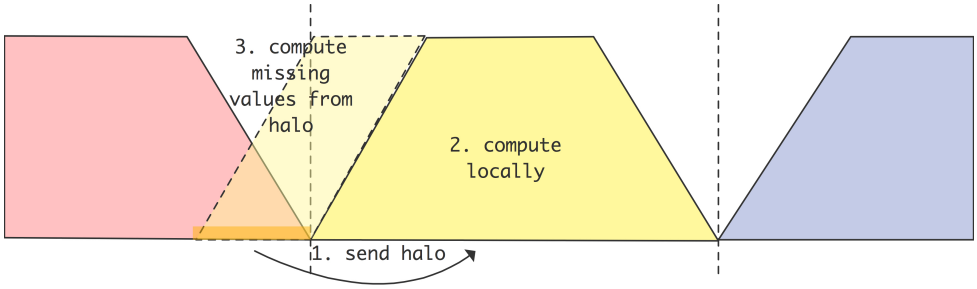

Figure 2: Computation of blocks of grid points over multiple iterations

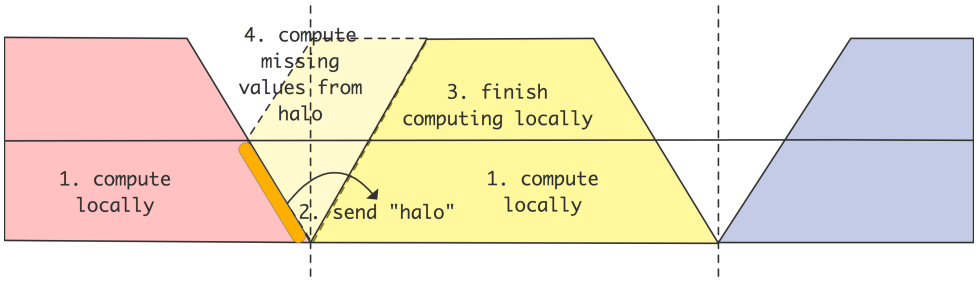

Figure 3: Computation of blocks of grid points over multiple iterations

3 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.

Above, 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 some synchronization synchronization points 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].

We start with a distributed task graph $\{L_p\}_p$ with a predecessor relation

$$t' \in pred(t) \equiv \{\text{task } t' \text{ computes direct input data for task } t\}$$

which holds on the global graph.

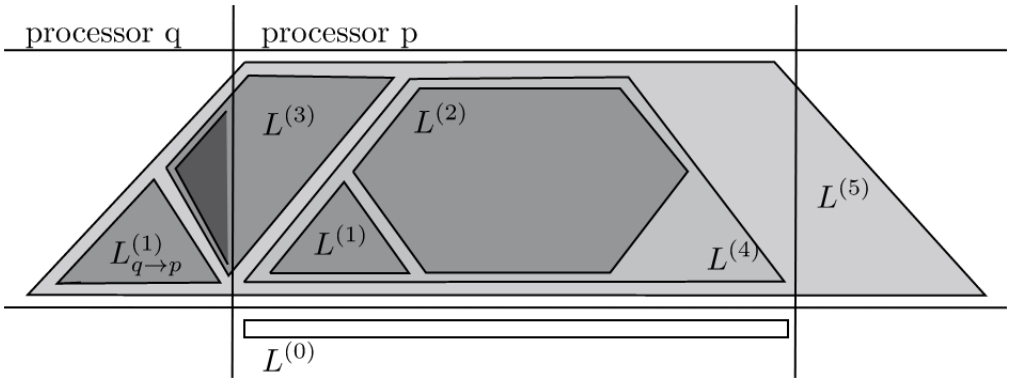

Figure 4: Subdivision of a local computation for minimizing communication and redundant computation.

We now derive subsets $L_p^{(1)}, L_p^{(2)}, L_p^{(3)}$ based on a formal latency-avoiding argument: we will have

$$L_p \subsetneq L_p^{(1)} \cup L_p^{(2)} \cup L_p^{(3)}$$

and any latency will be hidden by the computation of $L_p^{(2)}$, dependent of course on the size of the original task graph.

We now derive a collection of subsets, not neceessarily disjoint, that defines our latency tolerant computation. These concepts are illustrated in figure 4.

Subset 0: inherited from previous level We define $L_p^{(0)}$ as the data that is available on process p before any computation takes place:

$L_p^{(0)}$ contains initial conditions

This is either a true initial condition, or the final result of a previous block step.

Subset 4: all locally computed tasks We define an auxiliary set $L_p^{(4)}$ as the tasks in L_p that can be computed without needed data from processors $q \neq p$.

$$L_p^{(4)} \equiv \left\{ t : \mathit{pred}(t) \in \{L_p^{(0)} \cup L_p^{(4)}\} \right\}$$

This is a subset of all the L_p local tasks.

Subset 5: all predecessors of local tasks Next we define $L_p^{(5)}$ as all tasks that are computed anywhere to construct the local result L_p:

$$L_p^{(5)} \equiv L_p \cup pred(L_p)$$

This is a superset of the local tasks L_p; it includes non-local tasks (that is in L_q for $q \neq p$) that would be communicated in a naive computation.

Subset 1: locally computed tasks, to be used remotely We now define $L_p^{(1)}$ as the locally computed tasks on p that are needed for a $q \neq p$:

$$L_p^{(1)} \equiv L_p^{(4)} \cup \bigcup_{q \neq p} L_q^{(5)} - L_p^{(0)}$$

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 $L_p^{(1)}$ are being sent, we can do a local computation of the remainder of $L_p^{(4)}$:

$$L_p^{(2)} \equiv L_p^{(4)} - L_p^{(1)}$$

These tasks use results from L_1, but are otherwise entirely local, since they are part of the local set L_4.

Subset 3: halo elements and their successors The final part of the computation on p consist of those tasks that, recursively, need results from other processors.

Having received remote elements $L_{q \to p}^{(1)}$ from neighbouring processors $q \neq p$, we can construct the remaining elements of $L_p^{(5)}$ that are needed for L_p:

$$L_p^{(3)} \equiv L_p^{(5)} - L_p^{(4)} - \bigcup_{q \neq p} L_q^{(1)}$$

Theorem 1 *The splitting $L^{(1)}, L^{(2)}, L^{(3)}$ is well-formed and has overlap of communication $L^{(1)} \to L^{(3)}$ with the computation of $L^{(2)}$. Neither $L^{(1)}$ nor $L^{(2)}$ have synchronization points, so the whole algorithm has overlap.*

However, note that $L^{(1)} \cup L^{(2)} \cup L^{(3)}$ is most likely larger than L_k, corresponding to redundant calculation.

In figure 5 we indicate in red the part of $L^{(0)}$ that is sent, and the part of $L^{(3)}$ that is received.

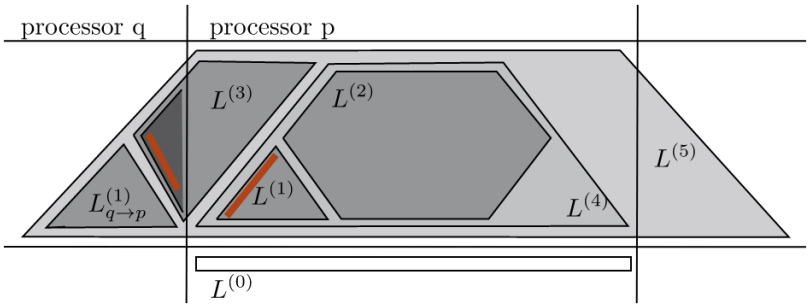

Figure 5: Communicated sets in the communication avoiding scheme

4 Simulation

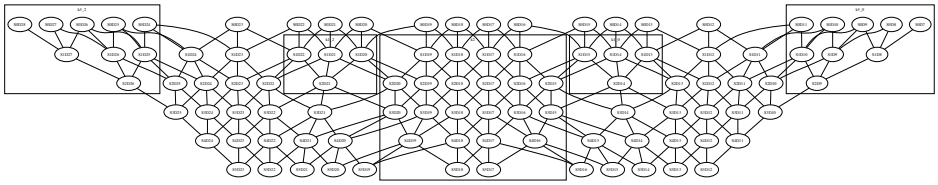

Figure 6: Depiction of k_1, k_2, k_3 sets for a processor doing a 1D heat equation.

We have written a simple simulator[1] that takes a task graph and identifies the k_1, k_2, k_3 sets. In figure 6 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:

- 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

[1] The code can be found in the code repository [7] under `pocs/avoid`.

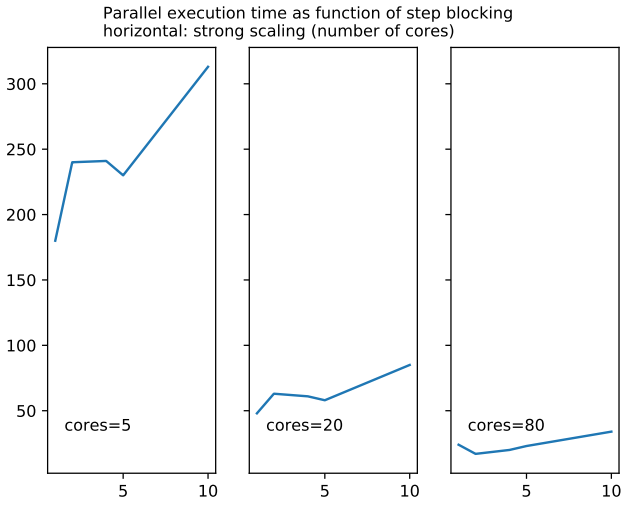

Figure 7: Runtime as a function of core count for moderate latency

will be more pronounced as the core count increases, since this reduces the no-node computation time.

First, in figure 7 we use a low latency; we see that only for very high thread count is there any gain. In figure 8 we use a higher latency, and we see that even for moderate thread counts blocking effects latency hiding.

5 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

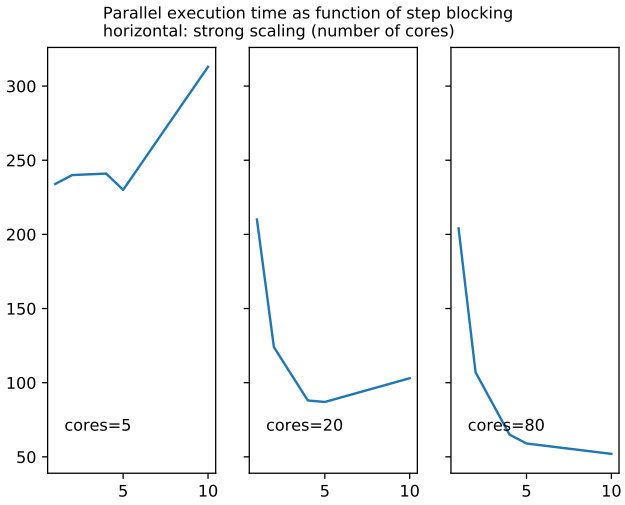

Figure 8: Runtime as a function of core count for high latency

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. IMP code repository, 2014-7. https://bitbucket.org/VictorEijkhout/imp-demo.

[8] Victor Eijkhout. A mathematical formalization of data parallel operations. *CoRR*, abs/1602.02409, 2016. http://arxiv.org/abs/1602.02409.

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

[10] 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.

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

[12] 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.

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