TACC Technical Report TR-11-03

A Global Model for Parallel Programming

Victor Eijkhout*, Barry Smith†, Ritu Arora‡

November 12, 2011

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
† Computation Institute, University of Chicago
‡ Texas Advanced Computing Center, The University of Texas at Austin
Abstract

Current day HPC is acutely aware of the importance of data movement during program execution. Strangely enough, very few programming tools have ways of explicitly manipulating communication as first-class objects. For instance, in MPI communication is implicitly defined by the sequence in which MPI calls in the code are executed. By contrast, we argue for the model used in the PETSc library, where the traffic pattern of an algorithm is an explicit first-class object, and the actual communication is the result of applying this object to input and output arrays.

We extend this model to include callbacks, and both data-parallel and data-pipelined communication.

Furthermore, we introduce a formal description of parallel algorithms, from which the communication objects can be derived by formal reasoning. Thus we argue for the prospect of a parallel language and compiler (or more modestly, a code transformation tool) that is more convenient for expressing parallel algorithms than MPI, while still delivering performance on the order of MPI code.
1 Introduction

It is a curious fact that the High Performance Computing (HPC) community has long been aware of the importance of data movement in computations, but that this awareness is not at all evident in the development of programming tools. Rather, researchers addressing the matter of programmer productivity often work implicitly with simplifying assumptions that are not borne out in practice.

The problems of data movement appear in several guises. For instance, there is talk of running into a ‘memory wall’: moving data from memory to the processor is becoming the limiting factor for performance, rather than what happens with that data in the processor. In parallel programming, there are ‘latency bound’ computations, meaning that moving data over the network between processors is, again, more important than what happens on the processor.

In the HPC community we see this problem addressed mostly by laborious programming and algorithm recoding to circumvent memory and network latency, while in the CS community little real support is offered. Although there is research in parallel programming languages aimed at increasing programmer productivity, little is done to address these problems of data movement explicitly. Typical approaches such as UPC or Chapel define data distributions, but leave any data traffic implicitly defined, at least on the user level. In fact, it is ‘[a] stated goal of the Chapel distribution language to avoid explicit management of communication in a program’ [6]. We think this is completely backwards.

The upshot of this is that such languages give very poor performance when used in a naive way compared to a parallel code using the MPI library [10]. This is not surprising: MPI allows the user to reflect a great deal of knowledge about the nature of the application in the code, so that code can be very efficient. Needless to say, this is not a high-productivity approach: distributed parallel programming is of a more difficult type than regular sequential programming, with many opportunities for performance loss. However, high-productivity tools and languages leave it up to lower software levels to achieve this efficiency. Typically, this is a failing approach, unless the programmer again goes to great lengths to optimize the code, effectively annulling the advantage of such tools.

Leaving communication defined implicitly or handled by lower software layers is suboptimal for at least the following reasons:

- If several independent messages are sent between two processors, they can be aggregated. This is hard to recognize by a compiler or a software service layer.
- In order to aggregate messages, a service layer could postpone communication until data are absolutely needed. However, that takes away the possibility of
overlapping communication and computation by starting the communication at the earliest possible time.

- The same communication pattern is often used several times in a row. This means that any preprocessing for optimizing the communication schedule can be amortized; languages that do not explicitly manage communication leave this possibility by the wayside.

We aim to address this situation by having an explicit acknowledgement of the importance of data movement in parallel programs, by incorporating communication patterns as first-class objects. Thus, we update the title of Wirth’s seminal book ‘Algorithms + Data structures = Programs’ as ‘Parallel Program = Algorithm + Data + Traffic’.

One of the few tools in general use that explicitly acknowledges the existence and importance of data traffic is the PETSc library, in its VecScatter object which can be seen as an implementation of the ‘inspector-executor’ model. (In another domain, the Erlang language has communication between processes as first-class objects.) Applying the VecScatter object to two vectors moves data between the vectors by mapping a set of indices in the one vector to another set in the other vector. If the input vector is distributed and the output local, this can be seen to effect the gathering of the ‘halo’ or ‘ghost region’ data for the sparse matrix-vector product.

Constructing this VecScatter object can be very expensive, potentially involving all-to-all communication, but its application can be efficient, involving only nearest-neighbour processes; see the example following. It also addresses the remaining two points above: it constructs a single message between any two processors, so communication is optimally aggregated; it uses nonblocking communication so traffic can be initiated at the earliest possible time.

An obvious example of the importance of VecScatter objects is their use in the distributed sparse matrix vector product, which occurs in each iteration of an iterative linear system. While communication follows no predetermined pattern, it is limited in execution to processes that model neighbouring parts of the computational domain. Thus, we can achieve very efficient code by

- having explicit objects describing data traffic,
- separating out the construction of these objects from their application, and
- offering powerful tools to the user for defining these objects.

---

1. A first-class object is one that can be assigned to a variable, passed as function argument and result, and, crucially, can be constructed at run-time. For instance, functions are not first-class objects in C/C++ since they cannot be constructed dynamically. They satisfy all other requirements.
2. This operation is explained in many HPC textbooks; for instance [8].
In this paper we will lay out a proposal for the definition of such a tool. Our ideas can be implemented as a library, in fact have been to an extent in the PETSc library, but we will also indicate how code transformation and development tools can be constructed to assist users in parallelizing existing sequential codes, or even to write in a new parallel programming language.

1.1 Related work

The standard communication library is MPI [17]. While it is powerful enough to express almost any parallel algorithm, it lacks support for higher level formulations. Thus it does not prevent programmers from writing inefficient code, and leads to relatively low programmer productivity.

We mention some related and contrasting approaches.

- MPI datatypes can describe arbitrary locations in distributed data structures.
- One-sided communication was implemented efficiently on the Cray T3E, and is making a resurgence on Cray Gemini and SGI UV. The big problem here is bundling.
- Parallel languages such as UPC, X10, Chapell, Co-array Fortran make it easier to write parallel code. However, they are very susceptible to the ‘many small messages’ problem. In fact, any optimized code in these languages looks like an MPI code, including all the buffering, but with send and receive calls replaced by copy operations.
- Global Arrays [18] is a package that supports one-sided put/get operations on globally known arrays. It has a hard time dealing with irregular data, and it can lead to many small messages since it has no abstraction mechanism.
- Charm++ [14] offers one-sided communications through active messages.
- CO2P3S [2] (Correct Object-Oriented Parallel Pattern-based Programming System) is based on parametrized design patterns for parallel programming. We feel this is insufficiently flexible; moreover, it misses capturing communication as first-class object, and is general seems geared towards shared memory threaded programming. Speedup is modest on small shared memory systems.

There are a few historical precedents for the notion of having explicit objects for traffic patterns. The ‘inspector-executor’ model was first proposed as a runtime code transformation technique [15]: a loop is inspected to determine the communication needed, which is then executed efficiently by bundling and scheduling individual communications optimally. The ‘PARTI primitives’ [19] formalized this model by turning the communication schema into an explicit object.

Finally, we indicate some topics that will explicitly stay out of our purview.
• Topology-aware mapping describes techniques for distributing data in such a way that as little contention as possible occurs. Our research assumes that mapping decisions have been made, and describes how to specify traffic resulting from it, not its optimization.

• Adaptive mesh refinement and dynamic load balancing are indispensible for dynamic applications. We do not target these techniques themselves, but we provide infrastructure for updating traffic patterns after incremental changes to the distributed data.

2 Traffic patterns

To motivate our formal model, in this section we present two basic types of traffic patterns:

• Parallel data communication, and
• Data pipelining.

In parallel data exchange we have a form of data parallelism, but in communication: all sends and receives are independent and can be executed simultaneously.

In data pipelining each processor accepts data, performs local communication, and sends parts of its data, whether old or newly computed, to other processors.

The difference between parallel data exchange and data pipelining is exemplified by the Jacobi and Gauss-Seidel (GS) methods for solving a linear system. Both feature the same computation, but in the Jacobi case all data traffic happens at the start of an iteration, while the GS method has in each iteration a pattern of receive-compute-send in each processor.

2.1 Scatters

Let us consider as a paradigmatic example the solution of a tridiagonal linear system by the Jacobi iteration; we will consider the GS method below. Sequentially, the Jacobi method would be given as

```c
double x[N+1], y[N+1], rhs[N];

// initialization....

for (iter=0; ; iter++) {
    for (i=1; i<=N; i++)
        y[i] = 5.* ( rhs[i-1] + x[i-1] + x[i+1] );
```
for (i=1; i<=N; i++)
    x[i] = y[i];
if (stopping test) exit
}

In the idealized cases where each processor has exactly one array element, parallel code would look like

double xglobal,xlocal[3],rhs;

for (iter=0; ; iter++) {
    xlocal[1] = // x[i-1], obtained through communication
    xlocal[2] = // x[i+1], obtained through communication
    xglobal = .5 * ( rhs + xlocal[1] + xlocal[2] );
}

The local processor code has clearly separated communication and computation steps, in each iteration of the solver process.

We can now express this traffic pattern in terms of a VecScatter object. The syntax is:

VecScatterCreate( in_vector, in_indices, out_vector, out_indices, &scatter_object );
VecScatterApply( in_vector, out_vector, scatter_object );

where in this example the in_vector is a distributed vector, and the out_vector is local. In terms of this, the code then becomes:

// let i be the index of this processor
VecScatterCreate( xglobal, {i,i-1,i+1}, xlocal, {0,1,2}, &scatter );
for (iter=0; ; iter++) {
    VecScatterApply( xglobal, xlocal, scatter );
}

2.2 Pipelines

The VecScatter object described above can handle traffic patterns where there is no dependency between one message and another. In this section we introduce the

---

3. The implementation in PETSc uses matched begin/end calls. This allows for overlapping communication and computation, which is a point we will address later. At the moment we leave out this detail for simplicity.
VecPipeline object (first used in the, no longer supported, ParPre library [5]) which can describe a data flow that interleaves with computation.

The sequential Gauss-Seidel (GS) code resembles the Jacobi code of section 2.1 to a great degree:

double x[N+1], y[N+1];

// initialization....

for (iter=0; ; iter++) {
    for (i=1; i<=N; i++) {
        y[i] = .5 * ( rhs[i-1] + x[i-1] + x[i+1] );
        x[i] = y[i];
    }
    if (stopping test) exit
}

or with the loop body written as compactly as possible by eliminating the y vector:

    x[i] = .5 * ( rhs[i-1] + x[i-1] + x[i+1] );

The difference is that now a newly computed $x[i]$ value is immediately used for index $i+1$ in the same (outer) iteration, rather than the next, as in the Jacobi method.

We develop the local processor code for the distributed memory parallel case in two steps, first considering the case where only one iteration is performed. First we allocate data: in the idealized case where each processor holds one array element, a single scalar holds the local part of the global vector; we also need a three-element array for the local part plus two ‘ghost’ elements corresponding to the data of the neighbouring processors:

double xglobal, xlocal[3];

The code then gathers local and neighbouring data into the xlocal array, and operates on it, storing the result in the xglobal element.

xlocal[0] = xglobal;

xlocal[1] = // x[i-1], obtained through communication
xlocal[2] = // x[i+1], obtained through communication
// send xglobal to processor to the right

We see that there is now communication both before and after the local computation.

Performing more than one iteration, it becomes necessary to update the xlocal[2] element. The following code implements this
for (iter=0; ; iter++) {
    xlocal[0] = xglobal;
    if ( /* not the first processor */ ) {
        // send xglobal to the left
        xlocal[1] = // x[i-1], received from the left,
                    // or boundary value
    }
    if ( /* this is not the last processor */ )
        xlocal[2] = // x[i+1], received from the right,
                    // or boundary value
    xglobal = .5 * (rhs + xlocal[1] + xlocal[2] );
    if ( /* this is not the last processor */ )
        // send xglobal to processor to the right
}

We can phrase this in terms of a proposed VecPipeline object, defined by:

    // create call:
    VecPipelineCreate( global_vec, before_indices, after_indices,
                     local_vec, before_indices, after_indices,
                     pipeline_object );
    // communication before the local processing:
    VecPipelineApplyLeft( object, global_vec, local_vec );
    // communication after the local processing:
    VecPipelineApplyRight( object, global_vec, local_vec );

making the code:

    // let i be the index of this processor
    VecPipelineCreate( xglobal, {i,i-1},{i,i+1}, xlocal, {0,1}, {0,2},
                      &pipeline );
    for (iter=0; ; iter++) {
        VecPipelineApplyLeft( pipeline, xglobal , xlocal,
                      FIRST= (iter==0) );
        xglobal = .5 * (rhs + xlocal[1] + xlocal[2] );
        VecPipelineApplyRight( pipeline, xglobal , xlocal,
                      LAST= (iter==MAXIT) );
    }

The flags for first and last iteration will often be omitted for brevity’s sake in our examples.
2.3 Efficiency of pipeline operations

The one-dimensional GS example as given above gives sequential execution, no matter the number of processors. Two remarks need to be made about this.

Realistic problems are two or three dimensional, and have relatively sparse graphs. In that case the dependencies induce ‘wavefronts’, which are sequentially dependent, but inside of which the operations are parallel. Expressing the GS iteration with a \texttt{VecPipeline} will automatically lead to this wavefront behaviour.

Even the one-dimensional GS iteration exhibits a form of wavefront parallelism, consisting of points in different outer iterations. This parallelism is not caught by the \texttt{VecPipeline} construct, and may not be expressible unless we talk about all outer iterations as a totality. Also, it is destroyed by including a stopping test.

2.4 The CS connection

The above concepts of \texttt{VecScatter} and \texttt{VecPipeline} are not new in Computer Science. Apart from the inspiration from the inspector-executor model already mentioned, they are at least similar to the Bulk Synchronous Parallel (BSP) [4] and Communication Sequential Processes (CSP) [13] models respectively.

The data-parallel nature of \texttt{VecScatters} relates them to the BSP model, where a code alternates between sections of (global) communication and local processing. However, we note that \texttt{VecScatters} do not incorporate a following barrier, so there is no global synchronization implied. Indeed, by composing \texttt{VecScatters} (as will happen in some of the examples later on in this paper) we arrive at a model that is more like \textit{data flow}: a processor task is waiting for input and the task ‘fires’ as soon as all inputs are available. These tasks may be SPMD in nature, but this does not imply temporal correlation.

Similarly, there is a relation between the \texttt{VecPipeline} idea of processes operating on data and pipelining it to further processes and the classical CSP model. However, note that CSP has no concept of data affinity; also, CSP is about arbitrary processes, while we are concerned with an SPMD model, pipelining data through the same kernel on a sequence of processors.

2.5 Distributed stopping tests

The above examples of Jacobi and GS iteration are typically performed until a global stopping test is satisfied. In case of the Jacobi iteration, the scatters induce a barrier-like behaviour, so we can perform a global reduction after the local computation without any major performance implications.
The GS iteration can not accommodate global reductions since they clash with the dataflow of the pipeline. Performing a reduction after the last processor has updated would induce long waits in the earlier processors. Instead, we observe that we can embed stopping information in the natural flow of data in the pipeline. Each processor would

- calculate the local part of its stopping test (for instance the local difference between two iterates), and send this to the subsequent processors in the pipeline, which would combine this with their local stopping information;
- the final processor in the pipeline then has the global stopping criterium, and it sends a confirmation message back through the pipeline, after which it stops computing; it will still execute a number of redundant receives;
- the confirmation information will gradually propagate back through the pipeline (recall from the GS example that newly computed results also need to be sent back through the pipe);
- with an overhead of at most $2P$ extra iterations the processors have now performed a global stopping test without collective operations.

### 3 Callbacks

In our description of pipelines and scatters, the communication object strictly specified what was to be sent, and from and to where. However, in the context of an algorithm a message typically has an action associated with it. We can formalize this by attaching a handle to a procedure on the recipient to the messages.

One example of how this can be implemented is the Charm++ package [14], which uses a mechanism reminiscent of active messages or remote procedure calls. Each processor object has a method for dealing with incoming data:

```python
class processor():
    def process_incoming(self, data):
        # do something local with incoming data
```

and this method can be called by a different processor, passing its local data as argument:

```python
processors[p].process_incoming(send_buffer)
```

The communication has now become one-sided, driven by the sender, and the execution is triggered when the data is available.

- This mechanism introduces asynchrony, because a method on the receiver can be called at any time. This asynchrony can improve efficiency, since data is processed at the earliest time. No polling or explicit scheduling is necessary.
Programming in this manner is slightly counterintuitive: in general it is more
natural to program in a ‘pull’ model (collect all data, then operate on it) than a
‘push’ model (send all data and have it be processed immediately).

Callbacks can be incorporated into our VecScatter and VecPipeline constructs. Each
processor has a method

```python
def handle_incoming( data, sending_proc ):
    # process and store in global data
    return
```

For now, we let these methods be void procedures, but there may be applications for
return results. This procedure is passed to the VecScatter and Pipeline apply calls:

```python
VecScatterApply( scatter, global_data, local_data,
                 handle_incoming )
```

We will now consider an example of the use of callbacks.

### 3.1 Matrix-vector product formulation with callbacks

The distributed sparse matrix vector product $y \leftarrow Ax$ for a matrix that is distributed by
block rows is typically coded as follows. Each processor $p$ has two arrays

- $x^{\text{loc}}$ containing the local part of the input vector, and
- $x^{\text{off}}$ for the elements of the input that have to be collected from other processors,
and correspondingly matrices $A^{\text{loc}}, A^{\text{off}}$ that together comprise the locally owned block row.

The algorithm then proceeds as follows:

1. Communicate to gather each processor’s $x^{\text{off}}$ array;
2. Evaluate on each processor $y \leftarrow A^{\text{loc}}x^{\text{loc}} + A^{\text{off}}x^{\text{off}}$.

In practice, one would use non-blocking communication:

1. Initiate non-blocking sends and receives to gather $x^{\text{off}}$ on each processor;
2. Perform the $A^{\text{loc}}x^{\text{loc}}$ product while communication is taking place;
3. Wait for all communication to finish;
4. Evaluate the $A^{\text{off}}x^{\text{off}}$ product and add up the partial results.

This algorithm can lead unnecessary wait time if the communications are unbalanced,
or issued staggered because of a work imbalance earlier in the code. In that case, one
can pull steps 3 and 4 apart and write:

$$A^{\text{off}}x^{\text{off}} = \sum_{q \neq p} A^{\text{off}}, q x^{\text{off}}, q$$
where \( p \) is the local processor number. The implementation of steps 3,4 would then be to have a `MPI_Wait_any` loop, and for each incoming message from a processor \( q \) to evaluate the product \( A^{\text{off},q}x^{\text{off},q} \).

Programming this is tedious, and callbacks offer a more elegant solution. Let \( p \) again be the processor that is assembling the local part of the matrix-vector product, and let \( q \) be a processor that needs to send data to \( p \). Using the callback mechanism, processor \( q \) would then

- form the array \( x^{\text{send},p} \) with the elements that will become \( x^{\text{off},q} \) on processor \( p \);
- and
- issue a remote call

\[
P[p].\text{off}_\text{prod}(q,x^{\text{off},p})
\]

where \( P[p] \) stands for the environment of processor \( p \) (see above).

Together, this active message has the effect of sending \( x^{\text{send},p} \) from \( q \) to \( p \) where it is received as \( x^{\text{off},q} \), and making processor \( p \) evaluate the product \( A^{\text{off},q}x^{\text{off},q} \).

### 4 A formal model of parallel computation

We define a computation (that is: a kernel, not a whole code) as a directed bipartite graph, that is, a tuple comprising an input data set, an output data set, and a set of elementary computations that take input items and map them to output items:

\[
A = \langle \text{In}, \text{Out}, E \rangle
\]

where

\[
\text{In}, \text{Out} \text{ are data structures}
\]

\((\alpha, \beta)\) is an elementary computation if \( \alpha \in \text{In}, \beta \in \text{Out} \)

and

\[
E \text{ is a set of } (\alpha, \beta) \text{ pairs.}
\]

Occasionally we switch from graph language to a functional notation: if \( e = (\alpha, \beta) \) we also write \( \alpha \ast e \rightarrow \beta \), where we use a left-to-right evaluation order \( \alpha \ast e \equiv e(\alpha) \).

---

4. We also note a certain resemblance between this notation and Hoare triples [12]. We will not explore matters of program correctness in this paper.
To parallelize a computation over \( P \) processors, we define

\[
A = \langle A_1, \ldots, A_P \rangle
\]

where

\[
A_p = \langle \text{In}_p, \text{Out}_p, E_p \rangle
\]

describes the parts of the input and output data set and work that are assigned to processor \( p \). The only restrictions on these distributions are

\[
\text{In} = \bigcup_p \text{In}_p, \quad \text{Out} = \bigcup_p \text{Out}_p, \quad E = \bigcup_p E_p;
\]

none of these distributions are required to be disjoint. To foreshadow the rest of the discussion in this section, we remark that elementary computations in \( E_p \) (meaning that they are executed on processor \( p \)) need not have their input data in \( \text{In}_p \), nor their output in \( \text{Out}_p \). Formalizing parallel computation will be seen to consist of indicating the relations in processor locality between input/output data sets and elementary computations.

Based on the fact that the computations in \( E_p \) are executed on processor \( p \) we can now define the input and output data for these computations:

\[
\text{In}(E_p) = \bigcup_{(\alpha, \beta) \in E_p} \alpha, \quad \text{Out}(E_p) = \bigcup_{(\alpha, \beta) \in E_p} \beta.
\]

These correspond to the input elements that are needed for the computations on processor \( p \), and the output elements that are produced by those computations. These sets are often contained in \( \text{In}_p, \text{Out}_p \), but need not: in fact we can now characterize the communication involved in an algorithm as

\[
\begin{cases}
\text{In}(E_p) - \text{In}_p & \text{data to be communicated to } p \text{ before computation} \\
\text{Out}(E_p) - \text{Out}_p & \text{data computed on } p, \text{ to be communicated out afterwards}
\end{cases}
\]

We see that some simple cases are covered by our model: the common ‘owner computes’ case corresponds to

\[
\text{Out}(E_p) = \text{Out}_p,
\]

that is, each processor computes the elements of its part of the output data structure. If additionally \( \text{In}(E_p) = \text{In}_p \), we have an embarrassingly parallel computation.

For greater generality we can let \( E \) be a hypergraph, that is, \( \alpha, \beta \) are subsets of \( \text{In}, \text{Out} \), rather than elements. This however, requires us to normalize each \( E_p \) to be such that
for any \((\alpha, \beta) \in E_p\) both \(\alpha\) is either completely contained in \(\text{In}_p\) or in \(\text{In}(E_p) - \text{In}_p\), and similarly that \(\beta\) is completely contained either in \(\text{Out}_p\) or \(\text{Out}(E_p) - \text{Out}_p\). We do this by splitting any \((\alpha, \beta) \in E_p\) in at most four parts that satisfy this requirement.

We now characterize the computations in \(E_p\) based on the localization of their data. We split

\[
E_p = L_p \cup R_p \cup S_p \cup X_p
\]

where

\[
L_p = \{(\alpha, \beta) \in E_p : \alpha \in \text{In}_p, \beta \in \text{Out}_p\}
\]

are the operations that take local input data, and touch only local elements of the output data structure;

\[
R_p = \{(\alpha, \beta) \in E_p : \alpha \in \text{In}(E_p) - \text{In}_p, \beta \in \text{Out}_p\}
\]

are the operations that use data not present on \(p\); the set \(\text{In}(E_p) - \text{In}_p\) is often called the ‘ghost region’;

\[
S_p = \{(\alpha, \beta) \in E_p : \alpha \in \text{In}_p, \beta \in \text{Out}(E_p) - \text{Out}_p\}
\]

is a less common set of operations that take local input data but produce data for other processors. The set \(\text{Out}(E_p) - \text{Out}_p\) is not commonly encountered in the literature, an exception being \([9]\), but we will see it appear in an example in section \([5.2]\) we will call this set the ‘halo’. Finally,

\[
X_p = \{(\alpha, \beta) \in E_p : \alpha \in \text{In}(E_p) - \text{In}_p, \beta \in \text{Out}(E_p) - \text{Out}_p\}
\]

are operations that take non-local input and produce non-local output. Such operations occur if dense matrix operations are mapped to a 2D processor grid, but we will not show any examples in this paper.

The sets \(L_p, R_p, S_p, X_p\) are illustrated in figure \([\text{fig:fig1}]\).

It is clear that data involved in the \(R_p, S_p, X_p\) instructions needs to be communicated. To make source and target of communications more explicit we define

\[
R_{q \rightarrow p} = \{ e \in R_p : e = (\alpha, \beta) \land \alpha \in \text{In}_q \},
\]

\[
S_{p \rightarrow q} = \{ e \in S_p : e = (\alpha, \beta) \land \beta \in \text{Out}_q \},
\]

that is, \(R_{q \rightarrow p}\) contains those \(R_p\) instructions with input on processor \(q\), and \(S_{p \rightarrow q}\) contains those \(S_p\) instructions for which the output belongs on \(q\).
4.1 Limit cases

Our model covers some limit cases of parallel computing. For instance, if all processors have access to the full input and output, that is,

\[ \text{In}_p \equiv \text{In}, \quad \text{Out}_p \equiv \text{Out} \]

we have a shared memory computation. Typically, \( E = \bigcup_i E_i \) will be a disjoint partitioning, but redundant work can be modeled by having nonzero intersection between some of the \( E_i \) sets.

Task parallelism is also easily modeled by giving each \( A_i \) task its own input and output, meaning that no communication between them is needed:

\[
\begin{align*}
A &= \langle A^{(i)} \rangle_i, \\
\text{In} &= \bigcup_i \text{In}^{(i)}, \quad \text{Out} = \bigcup_i \text{Out}^{(i)}, \quad E = \bigcup_i E^{(i)} \\
(\alpha, \beta) &\in E_p \Rightarrow \alpha \in \text{In}^{(i)}, \quad \beta \in \text{Out}^{(i)}
\end{align*}
\]

Here the \( A^{(i)} \) model independent tasks.

4.2 Scatters

The \textit{VecScatter} introduced above, including callback procedure, can now be modeled as

\[ \text{ScatR} = \{ R_{q\rightarrow p} \}_{p,q}. \]
The scatter with callforward procedure is modeled as

\[ ScatS = \{S_{p \rightarrow q}\}_{p,q} \]

Commonly, we ignore the second type, and write \( Scat \) instead of \( ScatR \). For completeness we define

\[ Thru = \{X_{p \rightarrow q}\}_{p,q} \]

These objects encapsulate the full communication happening in an algorithm. Having constructed these scatters, we can then instantiate the communication by performing (in functional notation; see earlier in this section)

\[ \text{In} \ast \text{ScatR} \rightarrow \text{Out} \]

et cetera.

Note that we have now derived the communication objects from the distribution of the input and output data and executable statements. Given a language that allows for the expression of such distributions, communication objects can be constructed by a combination of compiler and runtime system. The resulting code will be similar to code using the PETSc library, which is comparable in efficiency to the best hand-written code.

### 4.3 Collectives

The above concepts allow us to formulate individual messages, for instance as part of \( R_{q \rightarrow p} \) or \( S_{p \rightarrow q} \). A collective operation is an important work savings with respect to the individual messages that it could be implemented with. Thus, we want our model to incorporate collective operations. While we will not explicitly introduce concepts corresponding to collectives, we can easily infer them in any given algorithm. For instance, if for a given \( p \) the sets \( R_{q \rightarrow p} \) are nonzero for all \( q \), we recognize this as a reduction operation to \( p \).

Below we will see in examples how collectives can be inferred formally.

### 4.4 Discussion

We have defined parallel algorithms using a set-theoretic approach applied to distributions of the input, output, and work. Contrasting the above model of parallel computations with most parallel languages and programming systems, we see that our model is more detailed in that it allows separate partitioning of the data and the instructions.
Normally the partitioning of the work is implicitly induced by the partitioning of the data. While this often makes sense, we will show in section 5 with several examples that this more precise model allows us to formulate some algorithms that can otherwise not be expressed.

By taking a functional view of our model, considering the edge set $E$ as an operator mapping input to output, we arrive at a notion of distributed programming that has much in common with data flow. In the examples below we will see that we often need a decomposition of complicated communication patterns into a number of composited kernels. This functional model holds great promise. For instance, by investigating which operators commute, a compiler can effect the overlap of communication and computation.

5 Applications of our model

We discuss the applicability of our programming model to a number of examples that we consider important or at least illustrative in scientific computing. Other sets of problems to consider are

- The Cowichan set [20].
- The Perfect Club Benchmarks [3].
- The Berkeley Motifs.

5.1 Matrix-vector product

Let’s consider the shared and distributed matrix-vector product $y \leftarrow Ax$, or

$$y_i = \sum_j a_{ij} x_j$$

where $A$ is square of size $n \times n$, and we have $P$ processors. We will consider both the dense and the sparse form.

5.1.1 Shared memory implementation

The shared memory operation is easy to describe since all processors can see all input and output. Thus,

$$\begin{cases} 
\text{In}_p = \text{In}(E_p) = x \\
\text{Out}_p = \text{Out}(E_p) = y \\
E_p = \{ \eta_i : i \in I_p \}
\end{cases}$$
where the $I_p$ are a disjoint partitioning of the index set of the vectors. In other words, only the instructions are distributed, and no communication is needed. However, see section ?? for a discussion of multicore scheduling that marries shared memory to a concept of affinity.

5.1.2 Distributed memory implementation, informal presentation

The natural way to look at the distributed matrix-vector product is from the point of view of the Out$_p$ sets:

$$\forall i \in \text{Out}_p : y_i = \sum_j a_{ij} x_j.$$ We split this as before into

$$\forall i \in \text{Out}_p : y_i = \sum_j a_{ij} x_j = \sum_{j \in \text{In}_p} a_{ij} x_j + \sum_{j \notin \text{In}_p} a_{ij} x_j = \sum_{j \in \text{In}_p} a_{ij} x_j + \sum_{q \neq p} \sum_{j \in \text{In}_q} a_{ij} x_j = \sum_{j \in \text{In}_p} t_{ij} + \sum_{q \neq p} \sum_{j \in \text{In}_q} t_{ij}, \quad t_{ij} = a_{ij} x_j.$$ Now we note that in the computation above the first sum term describes operations on data that is already local, which puts it in $L_p$; for the second sum we distinguish three cases for the $t_{ij}$ computation based on where $a_{ij}$ is stored. We will give a rigorous proof of the following statements below.

- In the most common case where the matrix is distributed by block rows, so that $a_{ij}$ is stored on processor $p$, we have

  $$\begin{cases} i \in \text{Out}_p, \\
  j \notin \text{In}_p : j \in \text{In}_q \Rightarrow t_{ij} \text{ computed in } R_{q \rightarrow p} \end{cases}$$

  and $x_j$ needs to be communicated before $a_{ij} x_j$ can be computed.

- If the matrix is stored by columns, $a_{ij}$ is stored on processor $q$, where $q$ is such that $j \in \text{In}_q$. Hence we have

  $$\begin{cases} i \notin \text{Out}_p : i \in \text{Out}_q \\
  j \in \text{In}_p \Rightarrow t_{ij} \text{ computed in } S_{p \rightarrow q} \end{cases}$$

  (this is illustrated in figure ??) and we compute $t_{ij} \leftarrow a_{ij} x_j$ on processor $q$ and communicate it to processor $p$.

- Finally, in a two-dimensional decomposition of the matrix the computation of $t_{ij}$ is in $X_r$ for some processor $r$, where $r$ is such that

  $$i \in \text{Out}_r, \quad i \notin \text{Out}_p, \quad j \in \text{In}_r, \quad j \notin \text{In}_p$$

  We do not consider this case further, although we note that for parallel scalability it is actually the most important case; see [8, section 6.2].
5.1.3 Distributed implementation; formal derivation

For the formal derivation of the distributed implementation we consider separately the instructions for computing the temporary $t_{ij} = a_{ij}x_j$ quantities, and the reduction $y_i = \sum_j t_{ij}$. This makes the matrix vector product $y = Ax$ (where $x, y$ are of size $N$) a triplet $\langle x, y, E \rangle$ where

$$E = E^\eta \cup E^\tau, \quad E^\eta = \{\eta_i: i < N\}, \quad E^\tau = \{\tau_{ij}: i, j < N\}$$

with the elementary statements

$$\eta_i := 'y_i \leftarrow \sum_j t_{ij}', \quad \tau_{ij} := 't_{ij} \leftarrow a_{ij}x_j'. $$

We will consider both one-dimensional and two-dimensional distributions of the matrix. In both cases, the discussion is based on sets $I_p$ that form a disjoint partitioning of the index set $0 \ldots N - 1$, the simplest example being

$$I_p = \left[ p \times \frac{N}{P}, \ldots, (p + 1) \times \frac{N}{P} - 1 \right].$$

Formally, the sets $I_p$ are defined by the distribution of $x, y$:

$$\text{In}_p = x(I_p), \quad \text{Out}_p = y(I_p)$$

where $x(I_p)$ is shorthand for those elements of $x$ with indices in $I_p$.

The $\eta_i$ and $\tau_{ij}$ operations fit together as follows:

$$x(I_p) = \text{In}_p^\tau \xrightarrow{E_p^\tau} \text{Out}_p^\tau = \text{In}_p^\eta \xrightarrow{E_p^\eta} \text{Out}_p^\eta = y(I_p)$$
For the reduction part we have
\[ E^\eta_p = \{ \eta_i : i \in I_p \}, \quad \text{Out}^\eta_p = y(I_p) \]
so \( \text{Out}^\eta_p = \text{Out}(E^\eta_p) \), and no communication needs to be done after the reduction. What remains to be decided is the distribution of \( \text{In}^\eta_p \). Assuming no communication happens during the reduction, we have
\[ \text{In}(E^\eta_p) = \text{In}^\eta_p = t(I_p, *) \]
and the remaining part follows from \( E^\eta_p \): we can derive that \( \text{In}(E^\eta_p) = t(I_p, *) \), so
\[ x(I_p) = \text{In}^\tau_p \xrightarrow{E^\tau_p} \text{Out}^\tau_p = t(I_p, *) \]
the question is how much of this is local and how much needs to be communicated. The precise implementation of the communication follows from the choice to distribute the matrix by rows or columns, as we will now show.

**Distribution by rows** Distributing \( A \) by rows, that is, processor \( p \) stored \( A(I_p, *) \), means that
\[ E^\tau_p = \{ \tau_{ij} : i \in I_p \}. \]
Since \( \text{In}^\tau_p = x(I_p) \) and \( \text{In}(E^\tau_p) = x(*) \), all elements \( x(I_q) \) with \( q \neq p \) need to be communicated:
\[ R^\tau_{q \rightarrow p} = \{ \tau_{ij} : i \in I_p, j \in I_q \}. \]
The only local computation is
\[ L^\tau_p = \{ \tau_{ij} : i, j \in I_p \}. \]

**Distribution by columns** If \( A \) is distributed by columns,
\[ E^\tau_p = \{ \tau_{ij} : j \in I_p \}. \]
In this case,
\[ \text{In}^\tau_p = \text{In}(E^\tau_p) = x(I_p), \]
so no communication is needed prior to \( t_{ij} \) computation. On the other hand,
\[ \text{Out}^\tau_p = \text{In}^\eta_p = t(I_p, *) \]
but
\[ \text{Out}(E_p^\tau) = t(\ast, I_p), \]
so the \( t_{ij} \) have to be communicated after their computation:
\[ S_{p \rightarrow q}^\tau = \{ \tau_{ij} : i \in I_q, j \in I_p \} \]
for \( q \neq p \). Again there is a local component of
\[ L_p^\tau = \{ \tau_{ij} : i, j \in I_p \}. \]

**Two-dimensional distribution** For a two-dimensional distribution we assume correspondingly a two-dimensional processor indexing scheme for the processors (leaving entirely aside whether the underlying architecture is grid-oriented), giving
\[ E_{pq} = \{ \tau_{ij} : i \in I_p, j \in I_q \}. \]
It follows that
\[ \text{In}(E_{pq}) = x(I_q), \quad \text{Out}(E_{pq}) = y(I_q). \]
The analysis of where to perform the \( \tau_{ij} \) computations is similar to the cases discussed above; what makes this case interesting is that we can make various assumptions on the distributions of \( x, y \):

- Processors in the \( j \)-th column (of the processor grid) need \( x(I_j) \), and likewise processors in the \( i \)-th row contribute to \( y(I_i) \), so we can declare \( x, y \) to be accordingly (redundantly) distributed:
  \[ \text{In}_{pq} = x(I_q), \quad \text{Out}_{pq} = y(I_p). \]
- Alternatively we can declare that only one processor in the \( j \)-th column has \( x(I_j) \) at the start of the algorithm, and only one processor in the \( i \)-th row will fully assemble \( y(I_i) \). For convenience we take these to be the processors on the diagonal of the processor grid, and declare
  \[ \begin{align*}
  \text{In}_{pq} &= x(I_q) \\
  \text{In}_{pq} &= 0 \text{ if } p \neq q
  \end{align*} \]
  and correspondingly for \( \text{Out}_{pq} \).
In some cases (iterative methods for linear systems, Markov chains), the output of the matrix-vector product will be the input of a subsequent product operation. In the second distribution scheme this is no problem; in the first scheme we detect a mismatch, and \( y \) needs to be transposed over the processor grid.
5.2 Dense LU solution

There are two algorithms for solving a triangular system. The traditional algorithm for $Ly = b$ is

```c
for (r=0; r<n; r++) {
    t[r] = 0;
    for (c=0; c<r; c++)
        t[r] += L[r,c]*y[c];
    y[r] = L[r,r] \ (b[r]-t[r]);
}
```

where the backslash notation $A \backslash B$ stands for ‘apply the inverse of $A$ to $B$, from the left’. This algorithm accesses $L$ by rows.

On the other hand, a more column-oriented algorithm is

```c
for (r=0; r<n; r++)
    t[r] = 0;
for (r=0; r<n; r++)
    y[r] = L[r,r] \ (b[r]-t[r]);
for (c=r; c<n; c++)
    t[c] += L[c,r]*y[r];
```

5.2.1 Formal derivation of pipeline formulations

In this section we give both parallel code for the above algorithm variants using the `VecPipeline` construct, as well as the formal derivation of this code.

**First variant** Here we assume that each processor owns a row of $L$, and a single element of $y$.

Implementationally, there is a global vector $y$, of which each processor has just one element: $y_{\text{global}}$. In order to form the products $L[r,c]*y[c]$, the processor also needs an array $y_{\text{local}}$ of local $y$ values that are collected from other processors. An auxiliary array $t$ on each processor stores the products. We will write $L[j]$ for the $j$-th element of the locally stored matrix row. With the processor id stored in the variable $p$, this means that the local element $L[j]$ corresponds to global element $L[p,j]$.

The pipeline is then defined as

```c
VecPipelineCreate( y_{\text{global}}, \{0,...,p-1\}, {},
    y_{\text{local}}, \{0,...,p-1\} {}, &pipe_one )
```
where the second sets of indices are empty since we have only one wave of data flowing through the processors. The processor code is in figure 3.

```plaintext
VecPipelineLeft( pipe_one, yglobal, ylocal )
for i<p:
    t[i] = L[i] * ylocal[i]
yglobal = L[p] \ ( bglobal - sum( i<p, t[i] ) )
VecPipelineRight( pipe_one, yglobal, ylocal )
```

Figure 3: Local processor code for the lower triangular solve, first variant

A formal derivation looks much like the derivation of the matrix-vector product in section 5.1; we again consider two sets of instructions:

$$E^\tau = \{ \tau_{ij} \}, \quad \tau_{ij} := 't_{ij} \leftarrow l_{ij}y_j'$$

$$E^\eta = \{ \eta_i \}, \quad \eta_i := 'y_i \leftarrow l_{ii}^{-1}(b_i - \sum_{j<i} t_{ij})'$$

The algorithm now follows by formal reasoning:

1. From the definition of a pipeline, the $\eta_i$ calculations have to be fully local, so $\text{In}^\eta_p = \text{In}(E^\eta_p)$, which gives us

   $$\text{In}^\eta_p = t(I_p, *).$$

2. This implies that also

   $$\text{Out}^\tau_p = t(I_p, *).$$

3. Since we have a row distribution of the matrix,

   $$E^\tau_p = \{ \tau_{ij} : i \in I_p \}$$

   which satisfies point 2.

4. Another conclusion from the row distribution is

   $$\text{In}(E^\tau_p) = \{ y(I_q) : q < p \}.$$  

5. From matching up $\text{In}^\tau_p$ and $\text{In}(E^\tau_p)$ we find that the $\tau$ computations have a component

   $$R^\tau_{q\rightarrow p} = y(I_q)$$

   for all $q < p$. 

TR-11-03 22
We can also express the pipeline in functional notation as

\[
[R^n_{p \rightarrow q}]_{q \in \text{Tar}_p(E^n_p)} \circ L^T_p \circ [R^n_{q \rightarrow p}]_{q \in \text{Src}_p(E^n_p)}
\]

where (from right to left) the gather part of the pipeline is a set of \( R^n_{q \rightarrow p} \) where \( q \in \text{Src}_p(E^n_p) \), the local calculation of \( t_{ij} \) is in \( L^T_p \), and the scatter part of the pipeline is a set of \( R^n_{p \rightarrow q} \) where \( q \in \text{Tar}_p(E^n_p) \).

**Second variant**  This corresponds to the case where the matrix is partitioned over the processors by columns. Here, each processor collects \( t \) quantities from earlier processors, solves its local \( y \) element, computes the \( t \) quantities that depend on its column of \( L \), and sends these quantities to further processors.

Here we communicate \( t \) quantities, which, in the global view, constitute a two-dimensional array. Hence we posit the existence of a global two-dimensional object, of which each processor owns a column \( t_{*,p} \), stored locally as \( \text{tgloba}( p+1 \ldots n ) \). Each processor also has a local array \( t_{local} \) corresponding to \( \text{tgloba}( *,p ) \) that stores both the values that update its element of the right hand side, and the values that it computes.

The pipeline is defined as

\[
\text{VecPipelineCreate( tglobal, \{ (p,1),\ldots,(p,p-1) \}, \{\}, tlocal, \{ 1\ldots p-1 \}, \{\}, \&pipe\_two )}
\]

and the processor code is

\[
\text{VecPipelineLeft( pipe\_two, tglobal, tlocal )}
\]

\[
yglobal = L[p] \ \backslash \ ( bglobal - \text{sum( i<p, tlocal[i] ) } )
\]

for \( i>p: \)

\[
tlocal[i] = L[i] * yglobal
\]

\[
\text{VecPipelineRight( pipe\_two, tglobal, tlocal )}
\]

Figure 4: Local processor code for the lower triangular solve, second variant

While this pipeline code is easier to write than an MPI-based code, this code is not trivial; in particular the create call of the pipeline object requires careful thinking. On the other hand, the algorithm follows again by formal reasoning:

1. From the definition of a pipeline, the \( \eta_i \) calculations have to be fully local, so \( \text{In}^n_p = \text{In}(E^n_p) \), which gives us

\[
\text{In}^n_p = t(I_p,*).
\]
2. This implies that also
\[ \text{Out}_p^\tau = t(I_p, \ast). \]

3. With a column distribution of the matrix,
\[ E_p^\tau = \{ \tau_{ij} : j \in I_p \} \]
(note the subtle difference with the previous algorithm) which implies
\[ \text{Out}(E_p^\tau) = t(\ast, I_p). \]

4. Matching up \( \text{Out}_p^\tau \) and \( \text{Out}(E_p^\tau) \) we find the \( t \) computations now have a component
\[ S_{p \rightarrow q}^\tau = t(I_q, I_p). \]

5.3 Finite Element matrix construction

The distributed construction of a Finite Element Methods (FEM) matrix needs to reconcile two different partitioning schemes. During construction a partitioning based on elements is the natural choice, since numerical integration has to be done over elements. However, an assembled matrix is usually stored based on a node partitioning. The conflict is now that two elements can belong to different processors during the construction phase, yet both compute part of the same matrix coefficient.

This case can be accommodated by the VecScatter construct, if we use a mapping local (or distributed) vector of nodes in local elements to a distributed vector of nodes in the global ordering.

Formally, we give a simplified model of our computation as \( \langle M, D, E \rangle \) where \( M \) is the mesh, of which each processor has a disjoint part \( M_p \), \( D \) is the diagonal of the matrix, of which each processor has a disjoint part \( D_p \).

The mapping from mesh element and local number to global numbering is
\[ G : M \times L \mapsto D. \]

We define two inverses, first finding the elements that contain a globally numbered node:
\[ G^{-1}(n) = \{ e : \exists \ell : G(e, \ell) = n \}; \]

next finding the local number of a global numbered node on an element:
\[ G^{-1}(n, e) = \text{the unique } \ell \text{ such that } G(e, \ell) = n. \]

TR-11-03 24
With this we can express the instructions \( E \) as a union of

\[
E = E^\delta \cup E^\sigma = \{\delta_{e,\ell}\} \cup \{\sigma_i\},
\]

\[
\begin{align*}
\delta_{e,\ell} & := \text{‘compute } d(e, \ell)\text{’} \\
\sigma_i & := \text{‘sum } d_i = \sum_{e \in G^{-1}(i)} d(e, G^{-1}(e, i))\text{’}
\end{align*}
\]

We assume that the \( \delta_{e,\ell} \) computations can be performed without prior communication:

\[
\text{In}_p^\delta = \text{In}(E_p^\delta) = M_p.
\]

The total output of this operation is \( \text{Out}(E_p^\delta) = d(M_p, \star) \). Without going through the formal derivation we simply state that

\[
\text{Out}_{p\rightarrow q}^\delta = d(M_p \cap G^{-1}(D_q), \star)
\]

where we leave the \( \ell \) argument unspecified.

### 6 Language

The \texttt{VecScatter} and \texttt{VecPipeline} objects we have described above, as well as their create and apply functions, provide library calls that make it possible to write parallel programs in any regular language, and on a higher level than with MPI. The work required of the programmer is then:

- provide the layout of the parallel objects;
- analyze traffic patterns and construct the traffic objects accordingly;
- write the local processor code.

Thus we arrive at a first usage model for our library:

*The \texttt{VecScatter} and \texttt{VecPipeline} objects are high-productivity tools for parallel programming in a traditional language such as C/C++ or Fortran.*

However, the local processor code, as observed above, is often identical to the sequential code of the algorithm, except that it is applied to a local data structure, typically containing ‘ghost points’. Thus, the transformation from sequential code to parallel code should be doable by a source-to-source translation tool, guided by sufficient annotations on the sequential code.

*Transforming a sequential code to a parallel one can not be done fully automatically, but inserting calls to our API can be done by a source-to-source tool, guided interactively by the user or by directives.*

More interestingly, our approach with traffic objects makes it possible to design a parallel programming language, in which a parallel algorithm is given by
• local processor code, formulated as operations on local data;
• the layout of global data, and
• a description of the traffic pattern that maps global to local data.

Note that these are precisely the three ingredients claimed to comprise a parallel program in the title of this paper.

We aim to design at least the essentials of a parallel programming language that allows a programmer to specify
• local code
• task dependencies, as resulting from data traffic dependencies
• data distribution

7 Dynamism

We want to support updates to a traffic map. This can be needed in various circumstances, such as under domain refinement or dynamic load balancing. We assume that such changes to the parallel structure are incremental, and we want to update a traffic map rather than reconstruct it from scratch.

As a typical case, we consider a graph operation such as the distributed sparse matrix-vector product, and we design mechanisms to cope with small changes to the graph, such as adding a new node. If this node is close to the border between two processor partitions, it may require additional communication during construction of the ghost region.

The problem here is that one processor can typically determine which additional data it will need and from what processor, but that second processor is not capable of deducing that it needs to send new data. We solve this mismatch by letting each processor spin on an MPI_Probe instruction, to listen for any incoming requests. In order to decide how long the processor should listen we use a mechanism known as Distributed Termination Detection (DTD) [7, 16].

• A processor sends a request for data;
• all processors repeatedly call MPI_Probe; the processor that receives the message sends an acknowledgement back.
• Sending processors count the incoming acknowledgements;
• A global reduction is used to decided that all data requests have been sent and received; this is then broadcast, which is the signal for all processors to abandon the spin loop.
• The reduction and broadcast can be made more efficient by using a tree structure, making the whole operation cost $O(\log p)$. 

TR-11-03  26
8 Conclusion

In this paper we argued for the need of a parallel programming modality where data traffic is an explicit object, constructable at runtime. This model is directly implementable as an API; part of it has already been implemented in the PETSc library, which is regularly used for distributed codes running on up to hundreds of thousands of cores. The advantages of this API are that it allows for easy expression of efficient communication; by separating the construction and application of the communication object any preprocessing cost is amortized.

Next, we presented a graph/set-theoretic framework that is capable of expressing this programming modality. As we have shown in some examples, the definition of the communication object follows by a simple analysis from the distribution of data objects and executable statements. Thus, we arrived at the prospect of having a parallel language that allows for expressing algorithms in global terms, and which can be compiled to code that is comparable in efficiency to hand-written MPI code.

References


