Available online at www.sciencedirect.com

ScienceDirect

Procedia Computer Science 00 (2012) 1–10

Procedia Computer Science

International Conference on Computational Science, ICCS 2012

A Theory of Data Movement in Parallel Computations

Victor Eijkhout

Texas Advanced Computing Center, The University of Texas at Austin, Austin TX 78758

Abstract

We propose a set-theoretic model for parallelism. The model is based on separate distributions of data and work. The major theoretic result is that communication can then be derived by formal reasoning. While the model has an immediate interpretation in distributed memory parallelism, we show that it can also accomodate multicore shared memory programming, as well as clusters with accelerators.

The model gives rise in a natural way to objects that resemble the VecScatter construct in the PETSc library, or active messages in such packages as Charm++. Thus we argue that the model offers the prospect of an abstract programming system that can be compiled down to proven high-performance constructs.

Keywords: Parallel Programming, Programming models

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 largely missing from the development of programming tools.

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 in 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 computer science community little real support for these communication problems is offered. Although there is research in parallel programming languages aimed at increasing programmer productivity, little is done to address the 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 [1].

This puts such languages at a disadvantage compared to codes using the MPI library [2]. 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. Arguably this is not a high-productivity approach since the programmer is forced to spell out many details. Also, the efficiency attained is likely not to be portable to other architecture types. High-productivity

Email address: `eijkhout@tacc.utexas.edu` (Victor Eijkhout)
URL: `http://www.tacc.utexas.edu/~eijkhout` (Victor Eijkhout)

tools and languages leave it to lower software levels to achieve this efficiency. Typically, such an approach fails 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. Such aggregation 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 ("just in time"). However, that approach removes the possibility of overlapping communication and computation ("latency hiding") by starting the communication at the earliest possible time.
- The same communication pattern is often used several times in a row. Thus, any preprocessing for optimizing the communication schedule can be amortized. However, languages that do not explicitly manage communication leave this possibility by the wayside.

In this paper we give a theoretical treatment of parallel algorithms based on an explicit acknowledgment of the importance of data movement. We show that by a judicious design of the framework for algorithm descriptions the communication becomes an explicit object, which, moreover, can be formally derived and not left to a runtime system.

1.1. A vision for global parallel programming

We believe that high performance is attainable given enough tuning effort with currently existing tools, and that the main problem is to make these tools available in a high level flexible and integrative manner.

We present a new theoretical model for expressing parallelism; our main claim is that this model allows for formal derivation of the parallel data movement in an algorithm. In our Integrative Model for Parallelism (IMP) programmers can express algorithms in architecture-agnostic terms, yet the model can be compiled into MPI, active messages packages such as Charm++, thread libraries, or new systems such as Concurrent Collections; the model can fit itself to architectures as diverse as many-core, distributed memory, or clusters with attached accelerators.

Our model is graph based, with communication following from a formal derivation process, rather than coded by the programmer. We show by some examples how nontrivial communication patterns can be derived as first-class objects in the model. Our model is surprisingly versatile in incorporating algorithm and memory structures, hence the name Integrative Model for Parallelism (IMP).

A few things our model is not. It is a programming model, so we offer no transformations of existing codes. It is not a cost model, though we will attempt to include cost in our formal derivations. We do not propose a new programming language: just as CUDA and Cilk build on plain C/C++, we feel that high performance can be reached by compiling down to already existing tools. We do not claim to be able to derive optimal algorithms, routing, or scheduling: the programmer still has the responsibility for the algorithm design; we offer a high level, high productivity way of expressing the design. However, as indicated, our model will effect message aggregation and latency hiding.

1.2. Inspiration

One of the few tools in general use that explicitly acknowledges the existence and importance of data traffic is the PETSc library [3], in its `VecScatter` object. Notably, it offers a collective view to data movement: overall data movement is programmed, and the actual sends and receives are derived by the software.

A clear example of the importance of `VecScatter` objects is its use in the distributed sparse matrix vector product, which occurs in each iteration of an iterative linear system solution. In the context of Finite Element Method (FEM) calculations such communication follows no predetermined pattern, but is limited in execution to processes that model neighboring parts of the computational domain. 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.

Having communication patterns as first-class objects achieves the 'inspector-executor' model [4]: constructing this `VecScatter` object can be expensive, potentially involving all-to-all communication, but its application can be efficient, involving only nearest-neighbor processes.

The `VecScatter` also constructs a single message between any two processors, so communication is optimally aggregated; and it uses nonblocking communication so traffic can be initiated at the earliest possible time.

Another inspiration is the `VecPipeline` object of the ParPre library [5], which combines the abstract formulation of the `VecScatter` with a concept of dataflow between processors, essentially introducing a directed acyclic graph (DAG) between tasks. The paradigmatic example here is the propagation of vector elements in a Gauss-Seidel iteration.

1.3. Survey of earlier research

We briefly survey previous work on parallel programming, focusing on those aspects relevant to our model.

Early work on parallelism

A great deal of research has been done on communicating processes. This topic was in fact formalized long before it became relevant on parallel hardware. A landmark publication was Hoare's *Communicating Sequential Processes* [6]; many later publications built on this (for instance [7, 8]) and the echoes of this work can still be found in systems such as *Aspen* [9], which has queues that behave much like an implementation of the CSP channels, or Dryad (`http://research.microsoft.com/en-us/projects/Dryad/`).

However, all these systems have an implicit model of truly independent processes that occasionally communicate. Thus, most of the existing research is concerned with matters of deadlock, completion, and resource contention, none of which plays a significant role in our intended focus area of tightly coupled scientific applications.

Furthermore, such systems abstract away from the hardware completely: communication is formulated the same whether the processes are time-sharing on a single CPU, using a multicore CPU with shared memory, or using message passing on a cluster. Thus, architecture-specific optimizations are left to the communication layers and hardware.

Message passing

Starting around 1990 a great many software packages were developed that, with high performance in mind, systematized communication (especially for distributed-memory systems), but putting a considerable burden on the programmer by explicitly declaring all communication in processor-local terms. Foremost among these is the Message Passing Interface (MPI) [2, 10]. While MPI solely used a two-sided communication model (until the MPI-2 standard was formalized), around the same time several one-sided models were developed, such as *shmem* [11] and Charm++ [12]. The latter package offers "active messages," which are an important generalization of plain data transfer: in addition to being one-sided, they associate operations with the transferred data.

Inspector-corrector model

In HPC communications are often repetitions of the same irregular pattern. Thus it makes sense to have an 'inspector-executor' model, where the user would first declare a communication pattern to an inspector routine, which would then yield an object whose instantiation was the actual communication. This idea originated in an HPC system which in fact predates MPI: the 'Parti primitives' [4]. The inspector-executor model is currently available in the `VecScatter` object of the PETSc library [3], which served as an inspiration for our model; a similar construct is also available in Trilinos [13]. Such constructs improve over MPI in that they offer a global, collective, view of communication. We adopt this view in our model.

Distributions

Many languages and systems for parallelism have realized that data layout is a crucial aspect. Thus, UPC [14], HPF [15] have ways of specifying array distributions. Later systems such as Petsc and Trilinos, Elemental [16] and Chapel [17] have distributions as first-class objects, which is crucial for the flexible treatment of irregular parallelism.

Parallel languages

While the strict notion of a parallelizing compiler has died long ago, languages that allow high level expression of parallelism still exist. PGAS languages, such as UPC offer easy programmability; however, they leave too much responsibility to the lower software layers. Hence, high performing codes in these languages require considerable tweaking.

BSP

Our proposed programming model is reminiscent (apart from the inspiration from PETSc as mentioned above) of two very different models for parallelism. On the one hand, our graph description is similar to the supersteps in bulk synchronous parallelism (BSP) [18, 19]. On the other hand, as the reader will see in the examples, we lack the barrier synchronization. The fact that elementary computations can fire once their inputs are available is similar to dataflow (see, for an example of one model, [20]). The fact that our model is a graph model seems to reinforce this similarity, but important differences exist, as we will point out.

2. The basic I/MP model

We define a computation as a directed bipartite graph, that is, a tuple comprising an input data set, an ouput data set, and a set of elementary computations that take input items and map them to output items:

$$A = \langle \mathrm{In}, \mathrm{Out}, E\rangle$$

where

$$\mathrm{In}, \mathrm{Out} \text{ are data structures and}$$

$$E \text{ is a set of } (\alpha,\beta) \text{ elementary computations, where } \alpha \in \mathrm{In}, \beta \in \mathrm{Out}.$$

To parallelize a computation over P processors, we define

$$A = \langle A_1, \ldots, A_P\rangle, \qquad A_p = \langle \mathrm{In}_p, \mathrm{Out}_p, E_p\rangle,$$

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

$$\mathrm{In} = \bigcup_p \mathrm{In}_p, \mathrm{Out} = \bigcup_p \mathrm{Out}_p, E = \bigcup_p E_p;$$

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 In_p, nor their output in 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:

$$\mathrm{In}(E_p) = \{\alpha \colon (\alpha,\beta) \in E_p\}, \quad \mathrm{Out}(E_p) = \{\beta \colon (\alpha,\beta) \in E_p\}.$$

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 related to $\mathrm{In}_p, \mathrm{Out}_p$ but are not identical: in fact we can now characterize the communication involved in an algorithm as

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

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

$$\mathrm{Out}(E_p) = \mathrm{Out}_p,$$

that is, each processor computes the elements of its part of the output data structure and no data is communicated after being computed. If additionally $\mathrm{In}(E_p) = \mathrm{In}_p$, we have an embarrassingly parallel computation because no communication occurs before or after computation.

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

2.1. Functional composition of kernels

The above definitions pertain to a single operation; if we compose two operations we use superscripts to identify the proper sets. If σ and τ are two operations we denote them formally as

$$A^\sigma = \langle \mathrm{In}^\sigma, \mathrm{Out}^\sigma, E^\sigma \rangle, \quad A^\tau = \langle \mathrm{In}^\tau, \mathrm{Out}^\tau, E^\tau \rangle.$$

Instead of the tradition notation $y = \tau(\sigma(x))$, we use a left-to-right functional notation

$$\mathrm{In}^\sigma \to E^\sigma \star E^\tau \to \mathrm{Out}^\tau;$$

that is, we interpret the E^σ, E^τ edge sets as functional mappings from their inputs to their outputs.

2.2. 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,

$$\mathrm{In}_p \equiv \mathrm{In}, \qquad \mathrm{Out}_p \equiv \mathrm{Out}$$

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

As already remarked above, if we have

$$\mathrm{Out}(E_p) = \mathrm{Out}_p, \quad \mathrm{In}(E_p) = \mathrm{In}_p$$

and $E = \cup_p E_p$ is disjoint, the computation is ‘conveniently parallel’ since no communication is needed.

3. An illustrative example

By way of example, we show how to apply our model to the parallel matrix-vector product. This algorithm was chosen because it offers a clear illustration; more complicated algorithms such as sorting or N-body problems can also be realized in the model. 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.

3.1. Shared memory implementation

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

$$\begin{cases} \mathrm{In}_p = \mathrm{In}(E_p) = x \\ Out_p = \mathrm{Out}(E_p) = y \\ E_p = \{\text{compute } y_i \colon i \in I_p\} \end{cases}$$

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.

3.2. Distributed memory implementation

The distributed case of the matrix-vector product is the first non-trivial example we consider. We split the computation into two kernels:

$$\begin{aligned} \forall_i \colon y_i &= \textstyle\sum_j a_{ij} x_j \\ &= \textstyle\sum_j t_{ij}, \quad t_{ij} = a_{ij} x_j. \end{aligned}$$

The formal presentation considers 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 $y \leftarrow Ax$ (where x, y are of size N) a composition (with $\cdot \star \cdot$ indicating left-to-right function composition) of two algorithms

$$A = A^\tau \star A^\eta, \qquad A^\tau = \langle x, t, E^\tau \rangle, A^\eta = \langle t, y, E^\eta \rangle$$

where

$$\begin{cases} E^\eta = \{\eta_i \colon i < N\} & \eta_i := \text{‘}\, y_i \leftarrow \sum_j t_{ij}\,\text{’} \\ E^\tau = \{\tau_{ij} \colon i, j < N\} & \tau_{ij} := \text{‘}\, t_{ij} \leftarrow a_{ij} x_j\,\text{’}. \end{cases}$$

We will only consider one-dimensional distributions of the matrix, but our analysis works for the two-dimensionally distributed case too. To define the distributions of the input and output, we introduce 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

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

where $x(I_p)$ is shorthand for those elements of x with indices in I_p. The distribution of t will be the key to the parallel algorithm: we create A and t to be similarly distributed since we want $t_{ij} = a_{ij} x_j$ to be computed on the processor that owns a_{ij}.

The η_i and τ_{ij} operations fit together as follows:

$$x(I_p) = \mathrm{In}^\tau_p \xrightarrow{E^\tau_p} \mathrm{Out}^\tau_p = \mathrm{In}^\eta_p \xrightarrow{E^\eta_p} \mathrm{Out}^\eta_p = y(I_p)$$

We will now derive what communication needs to take place in the parallel algortihm.

For the reduction part we use the “owner computes” rule, so that for any i, y_i is computed on the processor that owns y_i:

$$E^\eta_p = \{\eta_i \colon i \in I_p\} \Rightarrow \mathrm{Out}(E^\eta_p) = y(I_p).$$

We note that this gives $\mathrm{Out}(E^\eta_p) = \mathrm{Out}^\eta_p$. The reduce operation here can be a logical or a physical reduction, depending on distribution of the data. Thus, the programmer specifies the structure of the operation, and the compiler and runtime will generate communication instructions as needed.

Under the assumption that no communication happens during the reduction, we have

$$\mathrm{In}(E^\eta_p) = \mathrm{In}^\eta_p = t(I_p, *)$$

and the remaining part of the algorithm is the calculation of E^τ which satisfies

$$x(I_p) = \mathrm{In}^\tau_p \xrightarrow{E^\tau_p} \mathrm{Out}^\tau_p = t(I_p, *)$$

and during which all the communication takes place. 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 storing $A(I_p, *)$, means that

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

Since $\mathrm{In}^\tau_p = x(I_p)$ and $\mathrm{In}(E^\tau_p) = x(*)$, we conclude that an allgather of x is needed.

Next, the reduction is local, since both $\mathrm{In}^\eta_p = t(I_p, *)$, $\mathrm{In}(E^\eta_p) = t(I_p, *)$.

***Distribution by columns*.** The general principle is that we compute $t_{ij} = a_{ij} x_j$ on the processor that owns a_{ij}. Therefore, if A is distributed by columns, with processor p storing $A(*, I_p)$, we have

$$E^\tau_p = \{\tau_{ij} \colon j \in I_p\}.$$

It follows that $\mathrm{In}(E^\tau_p) = x(I_p)$. Since also $\mathrm{In}^\tau_p = x(I_p)$, no communication is needed prior to t_{ij} computation. While the user code is identical as in the row case, now the lower layer will not generate any communication instructions.

On the other hand,

$$\mathrm{Out}^\tau_p = \mathrm{In}^\eta_p = t(I_p, *) \qquad \text{and} \qquad \mathrm{Out}(E^\tau_p) = t(*, I_p),$$

so the t_{ij} have to be communicated after their computation, in what is either a global data transpose, or a reduce-scatter if we merge this with the subsequent reduction.

3.3. Other examples

Lack of space prevents us from giving more sophisticated examples. However, we note that nothing in our model limits us to dense linear algebra. The general concepts of input and output sets and partitioning fully allows for sparse matrix operations.

With functional composition of operations, and using subsets of the input/output, we see that division-based operations such as sorting are expressible in our model. Combining all these concepts allows us to express multi-level sparse operations such as N-body problems. We will report on this elsewhere.

4. Distributions

In examples such as in section 3 we showed how the abstract model leads to transformations of linear algebra objects. We now show how this linear algebra notion can be interpreted in programmatic terms.

Let us consider a vector of size N and P processors. A distribution is a function that maps each processor to a subset of N:

$$v\colon P \to 2^N.$$

Thus, each processor stores elements of the vector; the partitioning does not need to be disjoint.

A couple of examples. With $b = N/P$ (assuming for simplicity's sake that N is evenly divisible by P) we define the even distribution

$$e \equiv p \mapsto [pb, \ldots (p+1)b - 1],$$

the cyclic distribution

$$c \equiv p \mapsto \{i\colon \mod(i,b) = p\},$$

and the redundant replication

$$* \equiv p \mapsto N.$$

Let x be a vector and v a distribution, then we can introduce an elegant, though perhaps initially confusing, notation for distributed vectors:

$$x(v) \equiv p \mapsto x[v(p)]$$

That is, $x(v)$ is a function that gives for each processor p the elements of x that are stored on p according to the distribution v. As an important special case, $x(*)$ describes the case where each processor stores the whole vector.

If v and w are distributions, we can indicate their relationship:

$$x(w) \equiv p \mapsto x(v)[v^{-1} \circ w(p)]$$

This means that we can describe x distributed according to w in terms of the v distribution, involving the communication described as $v^{-1} \circ w$. (Distributions need not be invertible, but the expression v^{-1} has a well-formed interpretation regardless.)

4.1. Operations in terms of distributions

We return to the matrix-vector product and split it in purely local work, and data distributions.

$$\begin{aligned} Ax &= A \cdot (\mathrm{diag}(x)e) \\ &= (A \cdot \mathrm{diag}(x))e \end{aligned}$$

Here, $A \cdot \mathrm{diag}(x)$ is a right scaling which is a local operation if the distributions of A and x allow this; the multiplication of its result by e is a reduction.

We introduce an explicit notation for the right scaling:

$$A(v,w)\,\sigma_R\,x(w).$$

We can then declare a syntax rule for the σ_R operator, stating that the distributions match.

Product by rows. Using the distribution notation we can write the product by rows as

$$\begin{cases} t(v,*) \leftarrow A(v,*)\,\sigma_R\,x(*) \\ y(v) \leftarrow \sum_j t(v,*). \end{cases}$$

Similar to the right scaling, we declare a syntax rule that the second index of the operand of $\sum_j$ has to be $*$.

We now make the following observations.

- We are assuming that $A(v,*)$ describes the distribution of A, so no data traffic is needed there.
- On the other hand, x is distributed as $x(v)$, so $x(*)$ is a redistribution, specifically an allgather.
- The output of the scaling, $t(v,*)$, equals the input of the reduction, so no communication is needed there.
- Assuming that $y(v)$ is the desired distribution, no further communication is needed after the reduction.

Product by columns. Using the distribution notation we can write the product by columns as

$$\begin{cases} t(*,v) \leftarrow A(*,v)\,\sigma_R\,x(v) \\ y(v) \leftarrow \sum_j t(v,*) \end{cases}$$

The communication story is now as follows:

- We assume that A is distributed as $A(*,v)$, meaning that columns are distributed over the processors.
- $x(v)$ is the distribution on input, and no communication is needed; the distribution of the output is described by $t(*,v)$.
- On the other hand, the reduction still needs $t(v,*)$, so a global data transpose is needed:

$$t(v,*) \leftarrow t(*,v).$$

- The result of the reduction is again $y(v)$, which is the desired distribution.

Sparse case. For the sparse case we need to extend the distribution notation a little. Let v be a distribution:

$$v\colon P \to 2^N$$

and R a function

$$R\colon N \to 2^N$$

that gives for a row number i the location of the nonzero columns in that row.

Now we overload the comma and define a composite mapping:

$$v,w\colon p \mapsto v(p), w(p) \cap R(v(p))$$

This definition seems biased towards row distributions. However, we can define an adjoint $T\colon N \to 2^N$ to R as

$$\begin{array}{l} T(j) = \{i\colon j \in R(i)\} \\ R(i) = \{j\colon i \in T(j)\} \end{array}$$

With this, we get the equivalent definitions

$$v,w\colon p \mapsto \begin{cases} v(p), w(p) \cap R(v(p)) \\ v(p) \cap T(w(p)), w(p) \end{cases}$$

Now we can write again $A(v,*) \cdot x(*)$. However, because of the new definition of v,w, x only has to be gathered in a limited way. The resulting operation can be termed a ‘shadowed all-gather’: the input vector is sent to all processors as in the dense case, except that on each processor a shadow mask indicates which components are needed.

4.2. Discussion

These distributions fit the IMP model in a natural way. In the case of the matrix-vector product we have

$$\mathrm{In} = x(v), \qquad \mathrm{In}(E) = x(*)$$

so the communication part (an allgather) happens prior to the computation. Thus we see that the $\mathrm{In}_p \rightarrow \mathrm{In}(E_p)$ transformation can be extressed in terms of transforms. This mapping from one transform to another is reminiscent of the `VecScatter` object in PETSc.

5. Other memory models

In the above discussion we implicitly used a distributed memory model. For space reasons we can not detail how the model can be applied to other memory models, but we limit ourselves to discussing the general approach.

The IMP model as described above interprets an algorithm as a DAG, which can either be interpreted as a task graph, or as a dataflow formulation. In either case, nothing in the model forbids certain arcs; in other words, the algorithm graph is embedded in an architecture graph that is a clique. In a distributed memory cluster this makes sense, as any two nodes can communicate directly (at uniform cost in the case of a fat-tree network).

The key to incorporating other memory structures in the IMP model is to allow nontrivial architecture graphs, and to make the coupling between algorithm graph and architecture graph more flexible. For example:

- The architecture graph would not be fully connected, for instance a mesh. Different mappings of the task graph to the algorithm graph would then lead have different behaviour in terms of link congestion. These mappings can be described algebraically, leading to an analysis of the communication behaviour.
- The architecture graph can have multiple types of connections, for instance in the case of a cluster with accelerators where the cluster nodes are fully connected, and the accelerators only to their host node. One connection kind can then be factored out, with the quotient graph modeling the communications of the other type. Thus we can derive the aggregated MPI communications of an algorithm on such a cluster.
- A flexible assignment of processes to processors includes redundant assignment. Applying the process of taking quotient graphs then eliminates certain data movement.

We see that message minimization in the IMP model is a graph embedding problem, where the best mapping of the algorithm graph (which we note is formally derived from the distributions, not explicitly programmed) on the architecture graph. Details of this are still a topic of research.

To an extent, shared memory can also be modeled in IMP if we identify IMP data communication with cache misses. The realization here is that cores are processors with their own private memory, so despite the ‘shared memory’ name, there is actually distributed memory where the place of the network is taken by the shared bulk memory. Thus we can formulate optimal task scheduling by the same graph optimization strategy outlined above. Again, the details of this are still a topic of research.

6. Discussion

We have presented a new theoretical model for parallel computing, the Integrative Model for Parallelism (IMP), which unlike many previous models focuses on data *movement* rather than data *distribution* or process coordination. Such an emphasis is important for achieving high performance on current and future large scale architectures.

In our model, communication is a formally derived entity. Thus, the programmer writes in global terms without a need to code any communication explicitly. Having communication as an object has several advantages, foremost among which the adherence to the inspector-executor paradigm. In this, the expensive preprocessing to determine a communication pattern is performed once, resulting in a communication object, after which the actual communication is a (relatively inexpensive) instantiation of this object.

We have indicated how our model can be implemented in terms of transformations between different data distributions. Such transformations already exist in several software libraries, from which we draw the conclusion that our model can be implemented by compilation down to existing high-performance tools.

The model has an immediate interpretation in distributed memory where there is a one-to-one mapping between processes and processors. However, by decoupling the algorithmic dependency graph from the physical architecture connectivity graph, task scheduling in the context of shared memory multicore can also be covered with this model.

[1] R. Diaconescu, H. Zima, An approach to data distributions in Chapel, International Journal of High Performance Computing Applications 21 (3) (2007) 313–335. arXiv:http://hpc.sagepub.com/cgi/reprint/21/3/313.pdf, doi:10.1177/1094342007078451.
URL `http://hpc.sagepub.com/cgi/content/abstract/21/3/313`

[2] W. Gropp, E. Lusk, A. Skjellum, Using MPI, The MIT Press, 1994.

[3] W. D. Gropp, B. F. Smith, Scalable, extensible, and portable numerical libraries, in: Proceedings of the Scalable Parallel Libraries Conference, IEEE 1994, pp. 87–93.

[4] A. Sussman, J. Saltz, R. Das, S. Gupta, D. Mavriplis, R. Ponnusamy, Parti primitives for unstructured and block structured problems (1992).

[5] T. Chan, V. Eijkhout, Design of a library of parallel preconditioners, Intl. J. of High Perf. Comp. Appls.

[6] C. Hoare, Communicating Sequential Processes, Prentice Hall, 1985, iSBN-10: 0131532715, ISBN-13: 978-0131532717.

[7] S. E. Zenith, Process Interaction Models, CreateSpace, 2011.

[8] J. Misra, A Discipline of Multiprogramming: Programming Theory for Distributed Applications, Monographs in Computer Science, Springer Verlag.

[9] G. Upadhyaya, V. S. Pai, S. P. Midkiff, Expressing and exploiting concurrency in networked applications with aspen (2007) 13–23doi:http://doi.acm.org/10.1145/1229428.1229433.
URL `http://doi.acm.org/10.1145/1229428.1229433`

[10] W. Gropp, S. Huss-Lederman, A. Lumsdaine, E. Lusk, B. Nitzberg, W. Saphir, M. Snir, MPI: The Complete Reference, Volume 2 - The MPI-2 Extensions, MIT Press, 1998.

[11] Cray Research, Cray t3e^{TM} fortran optimization guide, `http://docs.cray.com/books/004-2518-002/html-004-2518-002/004-2518-002-toc.html`.

[12] L. V. Kale, S. Krishnan, Charm++: Parallel programming with message-driven objects, in: Parallel Programming using C++, G. V. Wilson and P. Lu, editors, MIT Press, 1996, pp. 175–213.

[13] M. A. Heroux, R. A. Bartlett, V. E. Howle, R. J. Hoekstra, J. J. Hu, T. G. Kolda, R. B. Lehoucq, K. R. Long, R. P. Pawlowski, E. T. Phipps, A. G. Salinger, H. K. Thornquist, R. S. Tuminaro, J. M. Willenbring, A. Williams, K. S. Stanley, An overview of the trilinos project, ACM Trans. Math. Softw. 31 (3) (2005) 397–423. doi:http://doi.acm.org/10.1145/1089014.1089021.

[14] C. Barton, C. Casçaval, G. Almási, Y. Zheng, M. Farreras, S. Chatterje, J. N. Amaral, Shared memory programming for large scale machines, in: PLDI '06: Proceedings of the 2006 ACM SIGPLAN conference on Programming language design and implementation, ACM, New York, NY, USA, 2006, pp. 108–117. doi:http://doi.acm.org/10.1145/1133981.1133995.

[15] K. Kennedy, C. Koelbel, H. Zima, The rise and fall of High Performance Fortran: an historical object lesson, in: Proceedings of the third ACM SIGPLAN conference on History of programming languages, HOPL III, ACM, New York, NY, USA, 2007, pp. 7–1–7–22. doi:http://doi.acm.org/10.1145/1238844.1238851.
URL `http://doi.acm.org/10.1145/1238844.1238851`

[16] J. Poulson, B. Marker, J. R. Hammond, R. van de Geijn, Elemental: a new framework for distributed memory dense matrix computations, ACM Trans. Math. Softw.Submitted.

[17] B. L. Chamberlain, S. J. Deitz, D. Iten, S.-E. Choi, User-defined distributions and layouts in Chapel: Philosophy and framework, in: 2nd USENIX Workshop on Hot Topics in Parallelism, 2010.

[18] L. G. Valiant, A bridging model for parallel computation, Commun. ACM 33 (1990) 103–111. doi:http://doi.acm.org/10.1145/79173.79181.
URL `http://doi.acm.org/10.1145/79173.79181`

[19] D. B. Skillicorn, J. M. D. Hill, W. F. Mccoll, Questions and answers about bsp (1996).

[20] A. K. Adiga, J. C. Browne, A graph model for parallel computations expressed in the computation structures language, in: ICPP, 1986, pp. 880–886.