A Unified Approach to Parallel Programming

Victor Eijkhout

***Abstract*—We propose a new theoretical model for parallelism. The model is explictly based on data and work distributions, a feature missing from other theoretical models. The major theoretic result is that data movement 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 shared memory and hybrid architectures such as clusters with accelerators.**

The model gives rise in a natural way to objects appearing in widely different parallel programming systems such as the PETSc library or the Quark task scheduler. Thus we argue that the model offers the prospect of a high productivity programming system that can be compiled down to proven high-performance environments.

***Index Terms*—Parallel programming, DAG model, distributed memory**

I. Introduction

As computer architectures become larger in scale and more sophisticated in their hybrid nature (cluster, shared memory, accelerators), the problem of high productivity high performance programming is becoming acute. The problem is only to a limited extent one of the low level programming models: the major part of the problem is the parallel coordination of cores, devices, cluster nodes, co-processors, et cetera.

Solutions such as CUDA or MPI have any number of limitations, foremost among which that they are all special purpose, so it is not possible to write a code that is portable between systems. Also, such programming systems are often of a low level, asking the programmer to be concerned with details that are not essential to the application.

In this paper we give the design of a system that takes an abstract approach to implementing parallel algorithms where specific architectural details can be explicitly modeled. Our Integrative Model for Parallelism (IMP) allows for an abstract description of an algorithm, that can be made explicit through successive transformations, one of which being its convolution with an abstract description of hardware.

Specifically, the IMP model is based on kernels, which correspond to parallel tasks without data dependencies. From the distribution of work and data, data movement is theoretically derived as a first-class object. A single kernel typically corresponds to a collective operation, or a bulk data transfer such as exists in the PETSc [1] and Trilinos [2] libraries.

By composing IMP kernels we arrive at an 'abstract algorithm', which is takes the form of a directed acyclic graph (DAG) or dataflow diagram. Actual data motion results from an assignment of tasks to 'computing locales': threads, cores, nodes, et cetera. The strength of the IMP model is that this assignment is again an explicit feature of the model, so data motion becomes derivable from the abstract algorithm, rather than being explicitly coded as MPI messages, or implicitly resulting as side-effect of the execution. We will explain this in detail, and give motivating examples.

Manuscript received July 18, 2012
V. Eijkhout is with the Texas Advanced Computing Center (TACC) of The University of Texas at Austin. E-mail: `eijkhout@tacc.utexas.edu`

Having an explicit data motion object has several advantages: for one, it means that a programmer does not have to code in terms of send/receive. For another, the same communication pattern is often used several times in a row. Thus, having a data motion object allows for any preprocessing for optimizing the communication schedule to be amortized. This is known as the 'inspector-executor' model [3].

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 cost can be included in our formal derivations. We do not propose a new programming language: 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.

II. The basic I/MP model

In this section we develop the basic theoretical framework of the IMP model.

A. Basics

We define a *kernel* in the IMP model 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 $\mathrm{In}, \mathrm{Out}$ are data structures, and E is a set of (α, β) elementary computations, where $\alpha \in \mathrm{In}, \beta \in \mathrm{Out}$ and 'elementary computations' are simple computations between a single input and output. This is a restriction that we will address in section II-C.

To parallelize a kernel 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. The communication in a parallel algorithm will be seen to follow directly from 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:

$$\begin{aligned}\mathrm{In}(E_p) &= \{\alpha\colon (\alpha,\beta)\in E_p\},\\ \mathrm{Out}(E_p) &= \{\beta\colon (\alpha,\beta)\in E_p\}.\end{aligned}$$

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

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

We see that with this basic model we have managed to capture simple message passing: if $(\alpha,\beta)\in E_p$ and $\alpha\in\mathrm{In}_q$ (and $\alpha\notin\mathrm{In}_p$) then a message needs to be sent from processor q to p. We will now expand this basic model.

B. From kernels to algorithms

Above we defined the concept of a parallel kernel. By composing multiple kernels we arrive an 'abstract algorithm': a description of data dependencies between parallel processes, but without regard for architectural details. This corresponds to the much-studied dataflow model where a task can start ('fire') if all of its inputs are available, which is when the earlier tasks have finished; see for instance [4].

As a notation for multiple kernels we use superscripts to identify the proper sets. If σ and τ are two kernels we denote them formally as

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

We now have several ways of denoting kernel composition. For simple comosition we can write $y=\tau(\sigma(x))$ or $y=\tau\circ\sigma(x)$. Interpreting the E^σ, E^τ edge sets as functional mappings from their inputs to their outputs we also write

$$\mathrm{Out}^\tau \leftarrow E^\tau \leftarrow E^\sigma(\mathrm{In}^\sigma).$$

The advantage of this arrow notation is that it becomes easy to express graphically a DAG of kernels[1]: one kernel can feed into more than one, breaking the linearity of simple composition.

Finally, if for each kernel we draw up the adjacency matrix we find a linear algebra description of the abstract algorithm:

$$\mathrm{Out}^\tau = A^\tau\cdot A^\sigma \mathrm{In}^\sigma,$$

where A^σ, A^τ are the adjacency matrices of the σ,τ operations, and $\mathrm{In}^\sigma, \mathrm{Out}^\tau$ are rendered as vectors.

[1]Note: this is not the DAG of tasks that appears in much current research, for which see later.

C. Normal form

We defined the edges in an IMP kernel to map a single input to a single output, which could be restrictive. We solve this by recognizing that the integration of computation and data movement in the elementary computations of a kernel does not exist in practice: communication and computation are separate activities. Thus, we can decompose an IMP kernel into two kernels, where one has arcs that are pure data movement, and one that has pure computation.

- Since we are mostly interested in data movement, the computation kernels will be omitted.
- The objection that arcs might connect subsets of In or Out disappears: a computation that has multiple inputs (and for instance combines them on the target) can be split into multiple arcs that simply move data, followed by a combination on the target task.

III. Distributions

The model as explained so far incorporates distributed data. We will now introduce distributions as formal entities. One justification for this is that the partitioning $p\mapsto\mathrm{In}_p$ is obviously a distribution, but $p\mapsto\mathrm{In}(E_p)$ is also one. The former is often disjoint; the latter commonly not. Thus, the transformation from one distribution to another is also an important aspect of our model. Also, formulating algorithms in terms of distributions lifts the interpretation of the model from simple message passing to a more global description.

Let us consider a vector[2] of size N and P processors. A distribution is a function that maps each processor to a subset of N. We make the common identification of $N=\{0,\ldots,N-1\}$ and $P=\{0,\ldots,P-1\}$; likewise N^M is the set of mappings from M to N, and thereby 2^N is the set of mappings from N to $\{0,1\}$; in effect the set of all subsets of $\{0,\ldots,N-1\}$. A distribution is then

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

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

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

$$b\equiv p\mapsto [p\beta,\ldots(p+1)\beta-1], \tag{1}$$

the cyclic distribution

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

and the redundant replication

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

Finally, we consider the 'natural' distribution

$$\uparrow\equiv p\mapsto[f(p),\ldots,f(p+1)-1]$$

where $f(p)$ is the number of elements stored on processors $0\ldots p-1$. If this distribution is induced by another distribution u, meaning that $f(p)=|u(p)|$, we denote this as $u\!\uparrow$.

[2]We can argue that limiting the exposition to vectors is no limitation, as any object will have a linearization of some sort.

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.

As observed above, the most important application of distributions is converting a vector between one distribution and another. We use the notation $T(u, v)$ for this conversion, so

$$x(v) = T(u, v)x(u). \tag{2}$$

A. The software context

Distributions can be used in software as follows. Suppose we are adding an kernel K with input x and output y in a code segment where certain distributions hold:

```
... code ...
{ x is distributed as x(u) }
// the operation mapping x->y goes here
{ y is distributed as y(v) }
... code ...
```

Suppose the kernel is defined as $K = \langle x(u'), y(v'), E^K \rangle$, then we need to surround the kernel by transformations T according to equation (2):

```
{ x is distributed as x(u) }
x(u') = T(u,u') x(u)
// apply the kernel on x(u'), giving y(v')
y(v') = T(v',v) y(v)
{ y is distributed as y(v) }
```

B. Composing distributions

If v and w are distributions, we can compose them, and indicate the relationship of the composition to the constituent parts:

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

This means that we can describe x distributed according to w in terms of the v distribution, involving the communication described as $v^{-1} \circ w$. We also denote this conversation between distributions as $T(v, w)$.

Example.: Let x be distributed as $x(v)$. To transform it to $x(*)$, we need to move data accoording to $v^{-1} \circ *$.

$$v^{-1} \circ *(p) = v^{-1}(N) = P.$$

After all this complication about composing distributions, we note that often we do not 'track' data through multiple redistributions: we take one distribution for given and only consider a single redistribution. After this, the resulting distribution is again interpreted as a natural redistribution, to be redistributed further again. This is modeled by the object $T(u, u\uparrow)$.

C. Sparse distributions

For dealing with irregular data access we extend the distribution notation further. Let G be an boolean adjacency matrix, and define

$$G(i) \equiv G_i \equiv \{j : g_{ij} \neq 0\}$$

which can be justified by interpreting the matrix as a row list of column lists of nonzero positions. For example, with a tri-diagonal matrix we have $G(i) = \{i-1, i, i+1\}$.

This adjacency matrix can be used to transform distributions: with

$$u : p \mapsto u(p) \in 2^N$$

we define

$$Gu \equiv p \mapsto \cup_{i \in u(p)} G_i \in 2^N \tag{3}$$

In the tridiagonal example, and using the block distribution of equation (1), we have:

$$\begin{cases} \text{let } u_0, u_1 \text{ be s.t. } u(p) = [u_0, \ldots, u_1], \\ \text{then } Gu(p) = [u_0 - 1, \ldots, u_1 + 1]. \end{cases}$$

The justification for such transformations on distribution is for irregular data access, such as in the sparse matrix-vector product. With u describing $p \mapsto \mathrm{In}_p$, and G the sparsity pattern of the matrix A, Gu corresponds to $\mathrm{In}(A_p)$.

IV. Examples

We will show two examples of how algorithms are expressed and analyzed in the IMP model.

A. Matrix-vector product

As a simple example we consider the dense matrix-vector product. There are several ways of implementing this algorithm, such as with the matrix distributed by rows, columns, or blocks (see for instance [5]), and each of the three variants requires a radically different implementation. In this section we show that it is possible to formulate the algorithm on a high level, such that data traffic is automatically correctly derived.

We split the computation into two I/MP kernels[3], corresponding to computation of temporaries and subsequent reduction:

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

For an I/MP kernel we need to distribute data and work. For the data we use some distribution u for both input and output. The work distribution is induced by the decision that t_{ij} is computed where a_{ij} lives.

We now consider the product by rows. In distribution notation the algorithm is:

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

and we reason as follows:

- $A(v, *)$ describes the distribution of A by rows.
- x is distributed on input as $x(v)$, so transforming it to $x(*)$ is an allgather.

[3]We only discuss 1D distributions; 2D is possible with slightly more notation.

- $t(v,*)$ is correctly distributed for the reduction, so no communication is needed there.
- $y(v)$ is the resulting distribution as desired.

For the product by columns we reason similarly. First of all, we express the algorithm in distribution notation:

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

The differences in reasoning are that x is correctly distributed as $x(v)$ on input, so no communication is needed here. On the other hand, the output of the first stage $t(*,v)$ is not correctly distributed for the reduction, and we conclude that a data transposition is needed.

The conclusion from this simple example is that it is possible to describe the algorithm on a high level, with a Matlab-like global notation, while the communication is formally derived from the distributions on the data.

B. N-body problems

Algorithms for the N-body problem need to compute in each time step the mutual interaction of each pair out of N particles, giving an $O(N^2)$ method. However, by suitable approximation of the 'far field' it becomes possible to have an $O(N \log N)$ or even an $O(N)$ algorithm, see the Barnes-Hut octree method [6] and the Greengard-Rokhlin fast multipole method [7].

The naive way of coding these algorithms uses a form where each particle needs to be able to read values of in principle every cell. This is easily implemented with shared memory or an emulation of it. Pseudo-code would look like:

```
parallel over all particles p:
  cell-list = all top level cells
  sequential over c in cell-list:
    if c is far away, evaluate forces p<->c
    otherwise
      open c and add children to cell-list
```

However, this algorithm can be implemented just as easily in distributed memory, using message passing [8]. To show that an implementation can be formally derived we consider the following form of the N-body algorithms (see [9]):

- The field due to cell i on level ℓ is given by
$$g(\ell,i) = \oplus_{j\in C(i)} g(\ell+1,j)$$
where $C(i)$ denotes the set of children of cell i and $\oplus$ stands for a general combining operator, for instance computing a joint mass and center of mass;
- The field felt by cell i on level ℓ is given by
$$f(\ell,i) = f(\ell-1,p(i)) + \sum_{j\in I_\ell(i)} g(\ell,j)$$
where $p(i)$ is the parent cell of i, and $I_\ell(i)$ is the interaction region of i: those cells on the same level ('cousins') for which we sum the field.

1) Kernel implementation: We can model the above formulation straightforwardly in terms of IMP kernels: the g computation has $E^{(g)} = E^\tau \cup E^\gamma$, where

$$\begin{cases} E^\tau = \{\tau^\ell_{ij}\} \\ E^\gamma = \{\gamma^\ell_i\} \end{cases}, \begin{cases} \forall_i \forall_{j\in C(i)}: & \tau^\ell_{ij} = \text{`}\ t^\ell_{ij} = g^{\ell+1}_j\ \text{'} \\ \forall_i: & \gamma^\ell_i = \text{`}\ g^\ell_i = \oplus_{j\in C(i)} t^\ell_{ij}\ \text{'} \end{cases} \tag{4}$$

The t^ℓ_{ij} quantities are introduced so that their assignment can model data communication: as in the matrix-vector example above, the g^ℓ_i reduction computation is then fully local.

Similarly, the f computation is $E^{(f)} = E^\rho \cup E^\sigma \cup E^\phi \cup E^\eta$, where

$$\begin{cases} E^\rho = \{\rho^\ell_i\} \\ E^\sigma = \{\sigma^\ell_{ij}\} \\ E^\eta = \{\eta^\ell_i\} \\ E^\phi = \{\phi^\ell_i\} \end{cases}, \begin{cases} \forall_i: & \rho^\ell_i = \text{`}\ r^\ell_i = f^{\ell-1}_{p(i)}\ \text{'} \\ \forall_i \forall_{j\in I_\ell(i)}: & \sigma^\ell_{ij} = \text{`}\ s^\ell_{ij} = g^\ell_j\ \text{'} \\ \forall_i: & \eta^\ell_i = \text{`}\ h^\ell_i = \sum_{j\in I_\ell(i)} s^\ell_{ij}\ \text{'} \\ \forall_i: & \phi^\ell_i = \text{`}\ f^\ell_i = r^\ell_i + h^\ell_i\ \text{'} \end{cases}, \tag{5}$$

These formulations can immediately be translated to a message passing implementation.

Transformations of the algorithm are possible. For instance, in the statement

$$\forall_i \forall_{j\in I_\ell(i)}: s^\ell_{ij} = g^\ell_j$$

we recognize a broadcast of g^ℓ_j to all nodes i such that $j \in I_\ell(i)$. We can reformulate it as such by exchanging the quantifiers:

$$\forall_j \forall_{i\in J_\ell(j)}: s^\ell_{ij} = g^\ell_j$$

where

$$J_\ell(i) = \{j : j \in I_\ell(i)\}.$$

2) Distribution implementation: Instead of individual message passing, we can derive an implementation using distributions. Consider for instance the g calculation of equation (4)

$$\begin{cases} \forall_i \forall_{j\in C(i)}: & \tau^\ell_{ij} = \text{`}\ t^\ell_{ij} = g^{\ell+1}_j\ \text{'} \\ \forall_i: & \gamma^\ell_i = \text{`}\ g^\ell_i = \oplus_{j\in C(i)} t^\ell_{ij}\ \text{'} \end{cases},$$

As in section III-C we define an adjacency matrix A for this operation by

$$A_i = \{j : j \in C(i)\}.$$

Now let g be distributed with some distribution u, then t is distributed as $t(Au)$, and summing the t^ℓ_{ij} terms is then a local operation.

3) Practical aspects: We have here given two implementations of tree algorithms for N-body problems. Both implementations can cope with the difficulties that distributed memory imposes; as indicated above, shared memory implementations are considerably easier to describe. However, our implementations essentially give a dependency graph of tasks, hence they can also serve as shared memory implementations.

In the distributed memory case we invoke the inspector-executor paradigm (see the introduction): we determine which elements need to communicate, in particular the $I_\ell(i)$ and $C_\ell(i)$ sets, and use this information to evaluate irregular gathers repeatedly.

V. From algorithm to implementation

The model of section II was built around the concepts of 'kernel' and 'abstract algorithm'. Algorithms were purely formulated in terms of dataflow and data dependencies; in particular no architectural considerations were taken into account. This model implicitly uses a flat processor structure: all processes can send to and receive from all others and all communications are treated equally. Thus, the model has direct applicability to distributed memory parallel computing with an interconnect that is essentially all-to-all, such as a fat-tree.

However, in other circumstances it fails to account for several aspects:

- A distributed memory architecture can have a mesh interconnect, or other scheme where certain processor pairs do not have a direct connection.
- In a cluster with accelerators the accelerators do not connect to the network but only to a host processor.
- The model does not suggest any scheduling for operations, and the available parallelism can greatly exceed the number of physically available processors.
- In shared memory, using threading, several processes live in the same physical address space. In this case, certain data dependencies correspond to a data movement no-op; also, the precise timing of tasks then becomes an issue.

In order to cover these aspects we need to include some more machinery in the IMP model. First of all, we introduce 'abstract implementations': abstract algorithms where each task receives a time stamp and is bound to a computing locus, an abstraction that will cover cores, nodes, et cetera. Formally:

$$B = \langle V, E, p, t\rangle \qquad \text{where} \qquad \begin{cases} p\colon V \to P \\ t\colon V \to \{t_0, t_1, \ldots\} \end{cases}$$

It is easy to associate an abstract implementation with an abstracct algorithm. Let $A = A^K \circ \cdots \circ A^1$ be an abstract algorithm consisting of K kernels. We express this in the form of a DAG:

$$A = \langle V, E\rangle, \qquad \begin{cases} V = \{(k,i)\colon k = 1\ldots K,\ i = 1\ldots |\mathrm{Out}^k|\} \\ E = \cup_k E^k \end{cases} \tag{6}$$

and one abstract implementation is found by associating vertex (k,i) with time k and locus i.

We call this type of graph an *abstract* implementation, since we still ignore various practical considerations:

- Arcs in this graph correspond to data movement, but they can take the form of a no-op, a cache miss, data copy, or an MPI message. We will later consider how this difference can be formalized.
- The abstract implementation can ask for more processes than there are processors, so an extra layer of mapping is needed.

Thus, we need to introduce transformations on abstract implementations in order to arrive at 'realizable implementations': abstract implementations that satisfy, or are optimized for, real-world constraints. We will do this algebraically, by applying transformations to the adjacency matrices, as explained in section II-B.

Finally, we describe how the realization properties of a realizable implementation can be derived formally: this step is necessary to account for such facts as that on-node MPI communication do not involve network traffic.

Discussion: The above transformations on adjacency graphs accomplish much the same as DAG schedulers such as Quark [10]. In fact, our transformations do not constitute a schedule optimization strategy. Finding such a strategy may be NP-complete [11], and static scheduling does not account for machine and OS 'noise'. Thus, the main point of this section is to illustrate how the IMP model can derive the DAGs that can then be scheduled by other means.

We will now show two examples of reasoning about abstract implementations.

A. *Derivation of physical data movement*

Applying our model to distributed memory with one process per processor, each data dependency corresponds uniquely to one phyisical data transfer. In architectures that feature multi-threading, shared memory cluster nodes, or co-processors, this is no longer the case. In this section we develop the mechanisms of deriving physical data movement from abstract data dependencies by considering two examples where the process-to-processor assignment mapping is not one-to-one.

Shared memory cluster nodes: We start with a common example of distributed memory clusters where each node supports multiple processes that live on shared memory. This is illustrated in figure 1, where we have two cluster

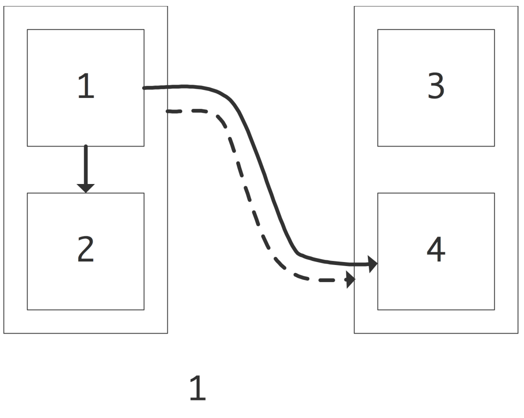

Fig. 1. Communication between processes (solid) and nodes (dotted) in a cluster/shared memory architecture

nodes with two shared memory processes each. In this case, algorithm edges $(1,2)$ or $(3,4)$ do not need an MPI message, while edges such as $(1,4)$ do.

Thus, we have two graphs: the process graph from the IMP kernel, connecting four processes, and the resulting *processor* graph, connecting two processors. We derive the latter from the former by a linear transformation. Let us consider specifically the IMP dependencies in figure 1.

First we form the embedding operator from the processes to the cluster nodes

$$I_2^4 = \begin{pmatrix} \star & \star & \cdot & \cdot \\ \cdot & \cdot & \star & \star \end{pmatrix}$$

that reflects that the first two processes live on node 1, while the third and fourth live on node 2. We denote its transpose by I_4^2.

The process edge $(1,4)$ in the figure can now be rendered with an adjacency matrix

$$G_4 = \begin{pmatrix} \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ \star & \cdot & \cdot & \cdot \end{pmatrix}.$$

If we now form the product

$$G_2 = I_2^4 \cdot G_4 \cdot I_4^2 = \begin{pmatrix} \cdot & \cdot \\ \star & \cdot \end{pmatrix}$$

we get the correct description of communications in terms of cluster nodes.

However, there is a conceptual problem with this. If we consider the intra-node edge $(1, 2)$, its transform becomes

$$G_4 = \begin{pmatrix} \cdot & \cdot & \cdot & \cdot \\ \star & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \end{pmatrix} \Rightarrow G_2 = I_2^4 \cdot G_4 \cdot I_4^2 = \begin{pmatrix} \star & \cdot \\ \cdot & \cdot \end{pmatrix}$$

stating that a message from node 1 to node 1 is required. Therefore, we introduce an extra term and form

$$(I_2^4 \cdot G_4) \circledast I_2^4 = \begin{pmatrix} \cdot & \cdot & \cdot & \cdot \\ \star & \cdot & \cdot & \cdot \end{pmatrix}$$

where $A \circledast B$ is the element-by-element computation of $a \circledast b \equiv a \wedge \neg b$. The $\circledast$-multiplication by I_2^4 has the effect of limiting the communication description to only processes that are on different processors. Redoing the above examples we now find the same G_2 matrix for the inter-node case, and

$$G_2 = \begin{pmatrix} \cdot & \cdot \\ \cdot & \cdot \end{pmatrix}$$

for the intra-node case, indicating that no MPI communication is needed.

Redundant processes: Next we consider redundant assignment of one process to two processors. As a specific example we take a domain decomposition method with two subdomains and one separator. In the forward sweep of the system solution the separator collects data from the subdomains; in the backward sweep it distributes data to them. We model this process by three processes, mapped to two processors, with the separator process redundantly assigned to both physical processors.

This process is illustrated in figure 2. In the left column we have the structure of the forward sweep. The algorithmic

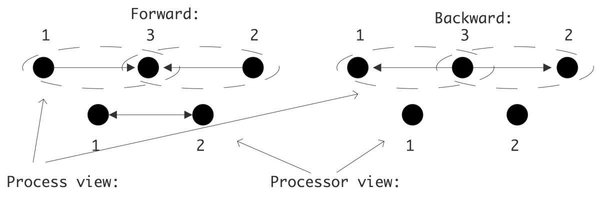

Fig. 2. Communication between logical and physical processors in the forward and backward sweep

data movement (top) takes the form of data send from the subdomains 1 and 2 to the separator 3, and the corresponding physical data movement (bottom). Since process 1 sends data to process 3, which is redundantly run on processor 2, there is a message from 1 to 2, and vice versa.

In the backward sweep, process 3 sends data to both 1 and 2, but now, since process 3 is redundantly run on both processors 1 and 2, this data movement is purely local to the processor, requiring no message passing.

We model this algebraically as follows. The forward and backward sweep are IMP kernels, which we represent by their adjacency matrices. These describe data movement between processes, that is, the algorithmic data movement:

$$L\colon G_3 = \begin{pmatrix} \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot \\ \star & \star & \cdot \end{pmatrix}, \qquad U\colon G_3 = \begin{pmatrix} \cdot & \cdot & \star \\ \cdot & \cdot & \star \\ \cdot & \cdot & \cdot \end{pmatrix}. \qquad (7)$$

The matrix embedding the logical processes in physical processors is I_2^3 and by I_3^2 we denote its transpose. We can now derive the matrix of physical communications as

$$G_2 = \left((I_2^3 \cdot G_3) \circledast I_2^3\right) \cdot I_3^2$$

similar to the previous example.

If we go through this calculation for the operators in equation (7), we find

$$L\colon G_2 = \begin{pmatrix} \cdot & \star \\ \star & \cdot \end{pmatrix}, \qquad U\colon G_2 = \begin{pmatrix} \cdot & \cdot \\ \cdot & \cdot \end{pmatrix} \qquad (8)$$

reflecting the above described behaviour that in the L sweep the processors have to exchange data, but not in the U sweep.

VI. Conclusion

In this paper we have presented the Integrative Model for Parallelism which offers a mode of describing parallel algorithms that is high-level and expressed in global terms Since data movement (such as message passing in a distributed memory context) is formally derived in the model, rather than explicitly coded it offers two prospects:

- Algorithm expression is expressed independent of any particular machine model, so we can achieve portability over architecture types; and
- since data movement is derived rather than coded, we may achieve higher programmer productivity.

Additionally we have shown how architectural features can explicitly be accomodated in this model.

References

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

[2] 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, and K. S. Stanley, “An overview of the trilinos project,” *ACM Trans. Math. Softw.*, vol. 31, no. 3, pp. 397–423, 2005.

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

[4] R. M. Karp and R. E. Miller, “Properties of a model for parallel computations: Determincay, termination, queueing,” *SIAM j. Appl Math.*, vol. 14, pp. 1390–1411, 1966.

[5] G. Stewart, “Communication and matrix computations on large message passing systems,” *Parallel Computing*, vol. 16, pp. 27–40, 1990.

[6] J. Barnes and P. Hut, “A hierarchical O(N log N) force-calculation algorithm,” *Nature*, vol. 324, pp. 446–449, 1986. [Online]. Available: http://dx.doi.org/10.1038/324446a0

[7] L. Greengard and V. Rokhlin, “A fast algorithm for particle simulations,” *J. Comput. Phys.*, vol. 73, p. 325, 1987.

[8] J. K. Salmon, M. S. Warren, and G. S. Winckelmans, “Fast parallel tree codes for gravitational and fluid dynamical n-body problems,” *Int. J. Supercomputer Appl*, vol. 8, pp. 129–142, 1986.

[9] J. Katzenelson, “Computational structure of the n-body problem,” *SIAM Journal of Scientific and Statistical Computing*, vol. 10, pp. 787–815, July 1989.

[10] A. YarKhan, J. Kurzak, and J. Dongarra, “QUARK users’ guide: Queueing and runtime for kernels,” University of Tennessee Innovative Computing Laboratory, Tech. Rep. ICL-UT-11-02, 2011.

[11] M. R. Garey and D. S. Johnson, “Complexity results for multiprocessor scheduling under resource constraints,” *SIAM J. Comput.*, vol. 4, pp. 397–411.