Abstract

In this report we give the philosophy, the basic concepts, and some demonstration, of the Integrative Model for Parallelism (IMP). We show that a judicious design of programming abstractions can lead to a system that accomplishes the holy grail of parallel programming:

1. High level expression,
2. translating to efficient use of low level primitives,
3. in a wide range of applications.

The following IMP reports are available or under construction:

IMP-01 IMP Distribution Theory
IMP-02 The deep theory of the Integrative Model
IMP-03 The type system of the Integrative Model
IMP-04 Task execution in the Integrative Model
IMP-05 Processors in the Integrative Model
IMP-06 Definition of a ‘communication avoiding’ compiler in the Integrative Model
IMP-07 Associative messsaging in the Integrative Model (under construction)
IMP-08 Resilience in the Integrative Model (under construction)
IMP-09 Tree codes in the Integrative Model
IMP-10 Thoughts on models for parallelism
IMP-11 A gentle introduction to the Integrative Model for Parallelism
IMP-12 K-means clustering in the Integrative Model
IMP-13 Sparse Operations in the Integrative Model for Parallelism
IMP-14 1.5D All-pairs Methods in the Integrative Model for Parallelism (under construction)
IMP-15 Collectives in the Integrative Model for Parallelism
IMP-16 Processor-local code generation (under construction)
IMP-17 The CG method in the Integrative Model for Parallelism (under construction)
IMP-18 A tutorial introduction to IMP software (under construction)
1 Introduction

In this report we describe the Integrative Model for Parallelism (IMP). This model purports to be a better solution to parallel programming in HPC than currently existing models.

IMP promises the following:

- Ease of programming: an IMP program is essentially sequential, with the user specifying algorithm steps in high level terms operating on distributed objects.
- Mode-independent parallel programming: the same program runs with MPI on distributed memory, task models on shared memory, or hybrid combinations of these.
- Asynchronous execution: despite the sequential user language, IMP generates a realization of the code in terms of asynchronously executing tasks, only governed by their dependencies.
- Latency-tolerance: the IMP system can analyze codes to realize overlap of communication and computation, or even do a ‘communication avoiding’ analysis.

In the remainder of this introduction we motivate some of the choices made in the Integrative Model, while in later sections we discuss the basic concepts and some practical results.

1.1 Motivation from programmability

We argue that many parallel programming systems are too general, and closely inspired by the target hardware. For instance message passing is made necessary by the presence of distributed memory, while Directed Acyclic Graph (DAG) models require shared memory and are naturally attractive in threaded environments. Both these models have practical disadvantages: the whole program behaviour is a non-trivial combination of the behaviours of the strands of execution that the user codes. Larus and Sutter phrase it as follows [11]:

[H]umans are quickly overwhelmed by concurrency and find it much more difficult to reason about concurrent than sequential code. Even careful people miss possible interleavings among even simple collections of partially ordered operations.

On the other hand, on a conceptual level, a parallel program can be fairly simple. For instance, a matrix-vector or matrix-matrix multiplication is conceptually a single, even data parallel, operation. All the intricacies of programming the data distributions needed for a scalable execution of such operations are really implementation details, and ought not to concern the user.

The conceptual attraction of such an approach is eloquently formulated in [7]:

[A]n HPF program may be understood (and debugged) using sequential semantics, a deterministic world that we are comfortable with. Once again, as in traditional programming, the programmer works with a single address space, treating an array as a single, monolithic object, regardless of how it may be distributed across the memories of a parallel machine. The programmer does specify data distributions, but these are at a very high level and only have the status of hints to the compiler, which is responsible for the actual data distribution[.]
Thus we argue for a programming model that is on an essentially higher level than the execution model. The programming model should be more than syntactic sugar around the execution primitives: it should be expressed on a level closer to application terms.

1.2 On the difficulty of realizing sequential semantics

Of course, designing a system with sequential semantics is not easy. The above mentioned HPF was a notable failure [5] for a variety of reasons, and UPC and CAF do not seem to scale well to clusters without further provisions. The reason for this shortfall in performance is that a data distribution is only half the story. Communication and synchronization follow from the interaction between the design of parallel data, and properties of the algorithm. Current programming models do not express this interaction.

IMP, on the other hand, is based on the construction of an object that brings together data and algorithm properties. To guide the reader’s thoughts, in a stencil update there occurs a data distribution giving the ‘halo regions’ of the processors: their shape is determined by both the array layout, and way the algorithm uses a stencil. IMP is based on a mathematical foundation of this distribution, which will be shown to underlie many different parallel scenarios.

Thus we characterize the failure of earlier systems with sequential semantics by the fact that they are unable to construct this algorithm-induced data distribution, and therefore are unable to schedule efficient data transfer. For instance, systems like UPC and Co-Array Fortran are notoriously unable to aggregate small messages. To do so requires a compiler that understands the relation between data distributions and data access patterns. Since this is a very hard problem, the matter is shifted to the runtime system, with an obvious performance penalty.

1.3 On the design of a universal parallel execution model

Despite all technical progress of decades of processor design, the execution model is still that of a Von Neumann machine. We can describe this as ‘program order’ semantics: no matter how out of order and multithreaded the execution gets, the API is as if a Von Neumann machine executed the program steps fully sequentially.

This model is not realistic in current times. For instance, it can not account for distributed memory parallelism, and it is a gross simplification of shared memory threaded parallelism.

Instead, we propose ‘dataflow’ as a unification of execution models. In dataflow, small chunks of execution, which we will call ‘tasks’, are related through a DAG where each task/node has incoming edges corresponding to the input data dependencies.

- Dataflow trivially generalizes program-order execution, since the DAG can always be linearized.
- Dataflow is a better match for threading models, since tasks are short-lived entities, comparable to threads that can be created and destroyed at will.
• Dataflow is even a model for message passing: in this case a processor corresponds to a set of tasks, and the incoming edges of the dataflow graph correspond to actual messages between processors.

Dataflow was at some time considered as an execution model that a compiler for a traditional language would target [9]. This promise has proved largely illusory.

On the other hand, dataflow is still a feasible execution model. Modern processors actually contain a dataflow engine to deal with out-of-order execution of single instructions. One can also consider task-based execution systems such as the tasks of OpenMP version 3 as a dataflow engine: such tasks can be added to a queue in any order, and they are executed by a scheduler in such a way that dependencies are obeyed.

1.4 Separation of concerns: synchronization versus execution

In the context of parallel and scientific computing it is clear where the strengths and weaknesses of automated code analysis, through compiler or source-to-source transformations lie. Traditionally, code that realizes a single execution thread can be optimized by compilers and source transformation tools to the extent that writing assembly is not done except in highly exceptional cases.

On the other hand, the synchronization of execution threads has eluded automated analysis, to the extent that parallel programmers routines write in low level tool such as MPI or threading libraries. The MPI library is often refered to as ‘the assembly language of parallel programming’.

The IMP model is a fargoing attempt at a systematic handling of the synchronization aspect of parallel programming. The IMP vocabulary handles synchronization between single-thread code blocks, while the programmer still has to write these blocks in a traditional language.

Such a separation is also found in the design of any MPI program, where MPI calls break up a code into blocks that execute local to the processor. In task systems such as DAGue this separation is even clearer: the local execution blocks are passed as function pointers to a scheduling system. IMP improves on these in the sense that the MPI calls and the task dependency structure are not user-specified, but derived from a higher level specification.

1.5 Outline

In the remainder of this report we start with a discussion of the concepts behind IMP and how they relate to earlier work (section 2). We then use a simple example to motivate the basic concepts of IMP in section 3; the basic concepts are formally defined in section 4. We show a prototype implementation in section 5, and discuss applications in section 6. A more rigorous definition of the model can be found in other reports in this series.
2 Design concepts of the Integrative Model for Parallelism

In the Integrative Model for Parallelism (IMP) we have realized a programming system with sequential semantics, while maintaining the efficiency of traditional programming systems such as MPI or OpenMP tasks. This success is due to the following design decisions.

2.1 Sequential semantics

As argued above, the programming concepts of the IMP model are based on sequential semantics: we specify the major steps in an algorithm, where each step has a distributed realization, which is however not visible on the programming level. The main concept here is that of a ‘kernel’: a specification of a function that is applied in parallel on one distributed object, yielding as result another distributed object.

Figure 1 shows the kernel structure of a heat equation (left), which is what the programmer specifies, and the task structure (right) which is derived by the IMP system.

2.2 Inspector-executor

The notion of inspector-executor reflects the fact that a code commonly needs some amount of analysis of runtime conditions, and that this analysis can often be reused. Thus, the inspector-executor paradigm was first invented as a compiler technique [6], and later as a more general code design technique [10]. In the latter case, a runtime component does a one-time analysis of the data access pattern of the code, and stores this in a condensed form that can be used at high speed during execution.

The prime example of this mechanism is the explicit construction of halo regions, and the messages needed to populate them, in sparse linear algebra packages such as the PETSc and Trilinos libraries [3, 4]. We can also consider task graph packages such as Quark and SuperMatrix [13, 1, 8] as instances of this mechanism: the task scheduler can perform some amount of introspection of the structure of the computation before the computation is actually performed.

Thus, the kernels of the IMP are separately declared, analyzed, and executed.

2.3 Generalized data parallelism

Parallelism in High Performance Computing (HPC) is often of a generalized data parallel type: matrix times vector, even irregular sparse; mesh operations including adaptive refinement; most linear algebra operations. In IMP, the user indicates explicitly the data parallelism in an operation by specifying the pointwise operation and the distribution to which it is applied. While this may seem a large burden on the user, given the right programming model this is actually not the case. We will address this matter below.

This programming model differs considerably from traditional engineering codes, but it actually has aspects in common with with current task-driven systems.
Figure 1: Kernel dataflow (left) and task dataflow (right) for the one-dimensional heat equation, executing 15 steps on 6 processors.
2.4 Programming with distributions

Our data model is that of distributions. While distributions have been used many times before, there are distinct innovations in our use of the term.

1. Rather than considering a distribution as mapping data to processors, we consider it a mapping of processors to data. This makes redundant replication and redundant computation elegantly expressible.

2. The reason that distributions have never been a convincing winning strategy is that compilers can not relate the distribution to the algorithm. IMP is based on a formalization of the concept of ‘halo’ to general parallelism: a ‘halo’ is a distribution that convolves the user-declared data distribution and the algorithm structure. From the halo distribution it is possible to derive all communication / synchronization of the algorithm.

Thus the key to the success of IMP is that it lets the programmer spell out

1. Available data parallelism
2. The data access pattern of the algorithm.

This makes the model more powerful than existing systems, which derive this information through a combination of the compiler and the runtime system.

As we will see in the examples below, this specification can take a number of elegant forms that are no great imposition on the programmer.

3 A motivating example for the basic concepts

We consider a simple data parallel example, and show how it leads to the basic concepts of IMP: the three-point operation

\[ \forall i: y_i = f(x_i, x_{i-1}, x_{i+1}) \]

which describes for instance the 1D heat equation

\[ y_i = 2x_i - x_{i-1} - x_{i+1}. \]

(Stencil operations are much studied; see e.g., [12] and the polydral model, e.g., [2]. However, we claim far greater generality for our model.) We illustrate this graphically by depicting the input and output vectors, stored distributed over the processors by contiguous blocks, and the three-point combining operation:
The distribution indicated by vertical dotted lines we call the $\alpha$-distribution for the input, and the $\gamma$-distribution for the output. These distributions are mathematically given as an assignment from processors to sets of indices:

$$\alpha: p \mapsto [i_{p,\text{min}}, \ldots, i_{p,\text{max}}].$$

The traditional concept of distributions in parallel programming systems is that of an assignment of data index to a processor, reflecting that each index ‘lives on’ one processor, or that that processor is responsible for computing that index of the output. We turn this upside down: we define a distribution as a mapping from processors to indices. This means that an index can ‘belong’ to more than one processor.

We use this notion of distribution to introduce a further distribution that expresses properties of the algorithm. The picture shows how some of the output elements on processor $p$ need inputs that are not present on $p$. For instance, the computation of $y_i$ for $i_{p,\text{min}}$ takes an element from processor $p-1$. This we call the $\beta$-distribution:

$$\beta(p)$$ is the set of indices that processor $p$ needs to compute the indices in $\gamma(p)$.

The second illustration depicts the different distributions for one particular process:

Observe that the $\beta$-distribution, unlike the $\alpha$ and $\gamma$ ones, is not disjoint: certain elements live on more than one processing element. It is also, unlike the $\alpha$ and $\gamma$ distributions, not specified by the
programmer: it is derived from the \( \gamma \)-distribution by applying the shift operations of the stencil. We will formalize this derivation below.

### 3.1 Definition of parallel computing

This gives us all the ingredients for reasoning about parallelism. Defining a kernel as a mapping from one distributed data set to another, all data dependence results from transforming data from \( \alpha \) to \( \beta \)-distribution. By analyzing the relation between these two we derive at dependencies between processors or tasks: each processor \( p \) depends on some predecessors \( q \), and this set of predecessors can be derived from the \( \alpha, \beta \) distributions.

In message passing, these dependences obviously corresponds to actual messages: for each process \( p \), the processes \( q \) that have elements in \( \beta(p) \) send data to \( p \). (If \( p = q \), of course at most a copy is called for.) Interestingly, this story has an interpretation in tasks on shared memory too. If we identify the \( \alpha \)-distribution on the input with tasks that produce this input, then the \( \beta \)-distribution describes what input-producing tasks a task \( p \) is dependent on. In this case, the transformation from \( \alpha \) to \( \beta \)-distribution gives rise to a dataflow formulation of the algorithm.

### 3.2 Programming the model

Clearly the \( \beta \) distribution is important since it combines properties of the data distributions and of the algorithm’s data dependencies. This distribution generalizes concepts such as the ‘halo region’ in distributed stencil calculations.

It remains to be shown that the \( \beta \) distribution can actually be formally derived. For this, we associate with the function \( f \) that we are computing a dependency signature called the ‘signature function’, denoted \( \sigma_f \). For instance for the computation of \( y_i = f(x_i, x_{i-1}, x_{i+1}) \), the signature function is

\[
\sigma_f(i) = \{i, i-1, i+1\}.
\]

With this, we state without proof (for now; see section 3.2) that

\[
\beta = \sigma_f(\gamma).
\]
That is, if the programmer can specify the data dependencies of the algorithm (and the work on stencil compilers such as the Pochoir compiler [12] shows that this is possible at least in some contexts), then a compiler/runtime system can derive the $\beta$ distribution, and from it, task dependencies and messages for parallel execution.

4 Basic concepts

In IMP-03 we give a full mathematical definition of the IMP model. Here we only list some of the basic concepts.

Suppose we have a set of indices $N$, and subsets $\text{Ind} = 2^N$. We also have a set $P$ of processors. Our first major concept then is that of distribution: an assignment from processors to subsets of the indexset:

<table>
<thead>
<tr>
<th>Datatype Distr: distributions of data over the processing elements</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\text{Distr} \equiv \text{Proc} \rightarrow \text{Ind}$</td>
</tr>
</tbody>
</table>

We concern ourselves with the data parallel application of a function:

<table>
<thead>
<tr>
<th>Datatype Func: functions with a single output</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\text{Func} \equiv \mathbb{R}^k \rightarrow \mathbb{R}$</td>
</tr>
</tbody>
</table>

The signature function is a description of what indices of the input are needed to compute an index of the output:

<table>
<thead>
<tr>
<th>Datatype Signature: Signature of data parallel functions</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\text{Signature} \equiv N \rightarrow \text{Ind}$</td>
</tr>
</tbody>
</table>

A distributed array is an array plus a distribution on it:

<table>
<thead>
<tr>
<th>Datatype DistrArray: distributed arrays</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\text{DistrArray} \equiv \text{Array} \circ \text{Distr} = \text{Proc} \rightarrow 2^{\mathbb{R}}$</td>
</tr>
</tbody>
</table>

We can now define a ‘kernel’ as a function applied between distributed objects:
Datatype Kernel: Data parallel functions between distributed arrays

\[
\text{Kernel} \equiv \text{Func} \times \text{DistrArray} \times \text{DistrArray}.
\]

An important result is that the beta distribution, our generalization of the concept of ‘halo’, is formally constructable in this model:

Function $\beta$: the $\beta$ distribution of a kernel

\[
K = (f, x, y) \quad \text{then} \quad \beta(K) = \sigma_f(\gamma(K))
\]

where $\sigma_f$ is the ‘signature function’ of the data parallel function.

We get to specific execution concept by defining a task, as a kernel, executed on a specific processor:

Datatype Task: A kernel executed on a specific processor

\[
\text{Task} \equiv \text{Kernel} \times \text{Proc}
\]

We can now define, but will not do so here, concepts such as messages, or the dependency graph of tasks.

Remark 1 Data parallel functions of more than one distributed input can be accommodated by having a signature function per input object.

5 Software realization and proof of concept

In this section we present the outline of a software implementation of the above theoretical notions. At first we consider the motivating example, then we discuss in more detail the implementation of the $I_f$ function.

5.1 Basic constructs

Each program starts by creating an environment object that keeps track of commandline arguments, as well as the parallel environment:

```c
IMP_environment *problem_environment =
    new IMP_environment(argc,argv);
```
There are many ways of creating a distribution. The simplest is a block distribution:

```c
IMP_distribution *blocked =
    new IMP_distribution
    (problem_environment,"disjoint-block",globalsize);
```

Given a distribution, creating an object is simple:

```c
IMP_object *input_vector = new IMP_object( blocked );
IMP_object *output_vector = new IMP_object( blocked );
```

Defining a kernel takes a couple of steps. The basic definition takes two vectors; after that we specify the function that is executed locally after the $\alpha \rightarrow \beta$ communication:

```c
IMP_kernel *update_step =
    new IMP_kernel(input_vector,output_vector);
update_step->localexectfn = &threepoint_execute;
```

The trickiest part is specifying the mechanism by which the $\beta$-distribution is constructed. For a three-point kernel that is by three shift operators:

```c
update_step->add_beta_oper( new ioperator(">>1") );
update_step->add_beta_oper( new ioperator("<<1") );
update_step->add_beta_oper( new ioperator("none") );
```

For a general sparse matrix we let the beta distribution be derived from the adjacency matrix:

```c
class mpi_spmvp_kernel : virtual public mpi_kernel {
public:
    mpi_spmvp_kernel( object *in,object *out,mpi_sparse_matrix *mat)
        : kernel(in,out),mpi_kernel(in,out) {
        set_name("sparse-mvp");
        dependency *d = last_dependency();
        d->set_index_pattern( mat );
        setlocalexectfn( &local_sparse_matrix_vector_multiply );
        localexectctx = (void*)mat;
    }
    virtual void analyze_dependencies() override {
        mpi_kernel::analyze_dependencies();
        mpi_sparse_matrix *mat = (mpi_sparse_matrix*)localexectctx;
        if (mat->get_trace())
            fmt::print("{}\n",get_out_object()->mytid(),mat->as_string());
    }
};
```

### 5.2 Execution

In an MPI context it is possible to execute a single kernel:

```c
update_step->execute()
```
The general mechanism uses a task queue:

```c++
IMP_task_queue* queue = new IMP_task_queue(problem_environment);
queue->add_kernel(step,update_step);
```

on which we perform analysis, and which, by the inspector-executor model, can be executed multiple times:

```c++
queue->analyze_dependencies();
queue->execute();
```

### 5.3 Conjugate gradient

We have implemented a conjugate gradient algorithm. The following snippet shows how algorithm steps are added to the queue. Kernels such as innerproducts and vector updates are given here as user-level primitives; however, they are easily implemented in the IMP basic system as shown in examples in section 6.

```cxx
// template_cg.cxx
kernel *rnorm = new IMP_norm_kernel( r,rnorms[it] );
queue->add_kernel(rnorm); rnorm->set_name("r norm");

kernel *precon = new IMP_preconditioning_kernel( r,z );
queue->add_kernel(precon);

kernel *rho_inprod = new IMP_innerproduct_kernel( r,z,rr );
queue->add_kernel(rho_inprod); rho_inprod->set_name("compute rho");

if (it==0) {
  kernel *pisz = new IMP_copy_kernel( z,pnew );
  queue->add_kernel(pisz); pisz->set_name("copy z to p");
} else {
  kernel *beta_calc = new IMP_scalar_kernel( rr,"/",rrp,beta );
  queue->add_kernel(beta_calc); beta_calc ->set_name("compute beta");

  kernel *pupdate = new IMP_axbyz_kernel( '+',one,z, '+',beta,p, pnew );
  queue->add_kernel(pupdate); pupdate ->set_name("update p");

  kernel *rrcopy = new IMP_copy_kernel( rr,rrp );
  queue->add_kernel(rrcopy); rrcopy ->set_name("save rr value");
}

rrp = new IMP_object(scalar,rrp0->get_data());

kernel *matvec = new IMP_spmvp_kernel( pnew,q,A );
queue->add_kernel(matvec);
```
kernel *pap_inprod = new IMP_innerproduct_kernel( pnew, q, pap );
queue->add_kernel(pap_inprod); pap_inprod->set_name("pap inner product");

kernel *alpha_calc = new IMP_scalar_kernel( rr,"/",pap,alpha );
queue->add_kernel(alpha_calc); alpha_calc->set_name("compute alpha");

kernel *xupdate = new IMP_axbyz_kernel( '+',one,x, '-',alpha,pnew, xnew );
queue->add_kernel(xupdate); xupdate->set_name("update x");

kernel *rupdate = new IMP_axbyz_kernel( '+',one,r, '-',alpha,q, rnew );
queue->add_kernel(rupdate); rupdate->set_name("update r");

The queue is analyzed and executed:
queue->analyze_dependencies();
//queue->gather_statistics();
queue->optimize();
queue->execute();

A preliminary test shows scaling comparable to PETSc:

![CG weak scaling chart]

The performance is lower than PETSc, but this is due to the fact that the IMP code has very little hardwired and almost everything derived from the algorithm. This leaves plenty of space for optimizations.

We briefly remark on collectives. Semantically, IMP will interpret a collective such as an all-reduce as the sum of its dependencies, and therefore implement it as a series of sends and receives. This is very inefficient, so we added a grouping mechanism to IMP:

- an all-reduce starts as all-to-all sends and receives in process groups,
followed by all-to-all reducing the groups. Effectively, this reduces the \( O(P^2) \) messages of the strict implementation to \( O(P\sqrt{P}) \), where optimally it would be \( O(P\log_2 P) \).

- Part of our performance loss is due to this log-versus-root order of complexity;
- with some more sophistication we can achieve the logarithm;
- our approach has the advantage that all operations in the reduction are actually tasks in the IMP model.

### 6 Applications of the IMP model

In this section we will show how the basic elements of the IMP system can be used to define the user-level kernels for various algorithms.

#### 6.1 Flexible treatment of distributions

Simple operations such as vector addition become complicated if the vectors concerned do not have the same distribution. For instance, the PETSc library requires matrices and vectors that appear in one operation, such as \( \text{MatMult} \) or \( \text{VecAXPY} \), to be identically distributed.

In IMP, a vector addition would mathematically be

\[
y(d) \leftarrow x_1(d) + x_2(d)
\]

where \( d \) is a distribution. If the \( \alpha \)-distributions of \( x_1, x_2 \) are not \( d \), data motion is automatically generated.

For instance, this is the definition of the vector sum operation:

```cpp
class sum_kernel : virtual public kernel {
public:
    sum_kernel( object *in1, object *in2, object *out ) : kernel(in1, out) {
        dependency *d;
        set_name("vector sum");
        d = last_dependency(); d->set_explicit_beta_distribution(out);
        add_in_object(in2);
        d = last_dependency(); d->set_explicit_beta_distribution(out);
        localexecutefn = &vectorsum;
    }
};
```

### 6.2 Regular stencils

The motivating example above used mechanism of shifts on the output distribution. These shifts are very similar to the mechanism used to specify stencils in the Pochoir compiler [12].
6.3 Irregular sparse matrices

We implement distributed sparse matrices with a small derived class:

class mpi_spmvp_kernel : virtual public mpi_kernel {
public:
    mpi_spmvp_kernel( object *in, object *out, mpi_sparse_matrix *mat) :
        kernel(in, out), mpi_kernel(in, out) {
        set_name("sparse-mvp");
        dependency *d = last_dependency();
        d->set_index_pattern(mat);
        set_localexecutefn(&local_sparse_matrix_vector_multiply);
        localexecuctext = (void*)mat;
    }
    virtual void analyze_dependencies() override {
        mpi_kernel::analyze_dependencies();
        mpi_sparse_matrix *mat = (mpi_sparse_matrix*)localexecuctext;
        if (mat->get_trace())
            fmt::print("[{}]
            get_out_object()->mytid(), mat->as_string());
    }
};

6.4 Redundant computation

Tree-structured computations are problem case for distributed computing: at
the highest levels in the tree there will be fewer nodes per level than proces-
sors. This means that we have to de-
cide either to let processors go inac-
tive, or to have partial or total redun-
dance calculation.

There are various arguments why re-
dundancy is the better strategy. The
obvious counter argument that redu-
dant computation entails more work
is countered by the fact that (in the
downward tree) there will be less com-
munication. Furthermore, redundant
computing is more ‘symmetric’, hence probably easier to reason about and to program; it will
certainly be easier to realize in the case of non-uniform refinement on the partially populated lev-
els.

Here we look at a gather operation towards the root of a tree, where the top levels of the tree are
redundantly distributed in a manner we will make precise. For purposes of illustration it is easiest
to consider figure 2.
1. The bottom two levels are disjointly distributed, and the first step of the reduction is a strictly
   local operation.
2. The top three levels are partially or totally redundant;
3. The reductions to the redundant tree levels are no longer local and involve communication.
   (If there source level is redundant, this communication is no longer uniquely determined.)
Mathematically, we derive these distributions by successive division. Let $\gamma$ be the distribution of
one level, and say
   \[ \gamma(p) = [i_0, \ldots, i_1] , \]
then $\gamma' = \gamma/2$ is the distribution
   \[ \gamma'(p) = [i_0/2, \ldots, i_1/2] . \]
This gives the desired behaviour:
\[
\begin{align*}
\gamma(p) &= [0] \\
\gamma(p+1) &= [1] \\
\gamma(p+2) &= [2] \\
\gamma(p+3) &= [2]
\end{align*}
\Rightarrow
\begin{align*}
\gamma'(p) &= [0] \\
\gamma'(p+1) &= [0] \\
\gamma'(p+2) &= [1] \\
\gamma'(p+3) &= [1]
\end{align*}

The code implementing this is relatively straightforward:
```cxx
%% unittest_struct.cxx
iooperator *div2 = new iooperator("/2"), *mul2 = new iooperator("x2");
distribution *distributions[nlevels];
product_object *objects[nlevels];
for (int nlevel=0; nlevel<nlevels; nlevel++) {
    if (nlevel==0) {
        distributions[0] = new product_block_distribution(arch,points_per_proc,-1);
    } else {
        distributions[nlevel] = distributions[nlevel-1]->operate(div2);
    }
    INFO( "level: " << nlevel << ", g=" << distributions[nlevel]->global_size() );
    objects[nlevel] = new product_object(distributions[nlevel]);
}
```
```
The indirect functions of the kernel are then given by an explicit function pointer:
```cxx
product_kernel *kernels[nlevels-1];
for (int nlevel=0; nlevel<nlevels-1; nlevel++) {
    INFO( "level: " << nlevel );
    char name[20];
    sprintf(name,"gather-%d",nlevel);
    ```
```
7 Conclusion

We have made a theoretical and practical case that it is possible to have an efficient, and efficiently programmable parallel system based on the following principles:

- There should be a separation between how algorithms are expressed and how they are executed. In particular, we argue for a programming model with sequential semantics and an execution model based on dataflow.

- Information about the algorithm and the execution should be explicitly expressed, rather be implicit and derived by a compiler or runtime system. We have shown that IMP offers a way to express these algorithm and data properties in a way that is not a great imposition to the programmer.

- For an efficient execution, it is necessary to express both information about the data and about the algorithm. IMP has a construct, the $\beta$-distribution, that contains this information and that is derived from the user specification of the data and algorithm. In particular, data dependencies (including messages) are not explicitly programmed, but rather derived, in IMP.

- It is possible to have a model for parallelism that is completely mathematically defined. This makes it possible to prove correctness, efficiency, and to define transformations on the algorithm. In work not reported here we have shown that IMP can accommodate, for instance, load balancing and redundant computing.

References


