TACC Technical Report IMP-05

Processors in the Integrative Model

Victor Eijkhout*

October 25, 2017

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

Abstract

The IMP model has tasks as the basic units of execution, rather than processors or processes. Here we define a process or processor as a collection of tasks, and we derive some properties of this concept.

The following IMP reports are available or under construction:

IMP-00 The IMP Elevator Pitch
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 (under construction)
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 (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)
IMP-19 Report on NSF EAGER 1451204.
IMP-20 A mathematical formalization of data parallel operations
IMP-21 Adaptive mesh refinement (under construction)
IMP-22 Implementing LULESH in IMP (under construction)
IMP-23 Distributed computing theory in IMP (under construction)
IMP-24 IMP as a vehicle for software/hardware co-design, with John McCalpin (under construction)
IMP-25 Dense linear algebra in IMP (under construction)
IMP-26 Load balancing in IMP (under construction)
IMP-27 Data analytics in IMP (under construction)

1 Motivation

The Integrative Model for Parallelism (IMP) is based on an execution model in terms of a dataflow graph[1] of tasks, rather than more permanent notions such as process or processor. This leaves us with flexibility in how to assign these tasks to actual physical processors. Here we define processors or processes as – not necessarily disjoint – subsets of the task graph, and we derive properties of these processors from the synchronization mechanisms in *IMP-04*.

2 Processors and synchronization

2.1 Tasks

So far we have considered a single kernel and the distribution transformations it needs. To implement a full algorithm we typically need multiple kernels. The multiple distribution transformations then turn into a graph of task dependencies.

Datatype Task: computation of the part of a kernel on a specific processor

$$\text{Task} \equiv \text{Kernel} \times P$$

A formal definition of an algorithm would require too much notation, so let's consider a simple case. Let $n > 0$, and let, for $i < n$, $K_i \in$ Kernel and, for $i \leq n$, $x_i \in$ DistrArray such that

$$\forall_{i<n}\colon x_{i+1} = f_i x_i$$

For each i we name the distributions of input and output

$$\alpha_i = \text{distr}(x_i), \qquad \gamma_i = \text{distr}(x_{i+1})$$

and σ_f is the signature function of K_f. We define $\beta_i = \sigma_f \gamma_i$, and we obtain the transformation $T_i = T(\alpha_i, \beta_i)$, defined as **??**.

Now we define a graph of tasks where each node has a number

$$\langle i, p \rangle \qquad \text{where} \qquad i \leq n, p \in P \tag{1}$$

and edges are defined as

$$\big\langle \langle i, q \rangle, \langle i+1, p \rangle \big\rangle \qquad \text{iff} \qquad q \in T_i(p).$$

In algorithms with a more complicated composition of kernels we replace the 'i' kernel numbering by a more abstract partial ordering.

In this partial ordering we reserve the notation $t_1 < t_2$ for tasks that have a direct predecessor relation; for the transitive relation we write $t_1 <^+ t_2$. For a task t, the set of immediate predecessors is denoted $pred(t)$.

1. To be precise, this graph is formally derived from a description in terms of distributions; see *IMP-01*.

2.2 Task assignment to processors

We formally define a processor as a subset of the task graph.

Definition 1 *Let T be the task graph (which is partially ordered; see section 2.1), then we define the set of* processors $p \mapsto C_p$ *as a covering of T:*

$$T = \cup_p C_p.$$

Processors do not need to be disjoint: by assigning a task to more than one processor we can introduce resilience or reduce synchronization; see below.

In the overall task graph we identify two subsets:

$$\begin{cases} T_0: & \text{initial tasks, having no predecessor} \\ T_\infty: & \text{final tasks, having no successor} \end{cases}$$

2.3 Definition of synchronization

Once we have a task graph we have to worry about how to execute it. In this section we give a rigorous definition of synchronization between processors as it is induced by the task graph, and we will analyze execution behaviour of the processors in terms of synchronization.

The following definition of a synchronization point makes immediate sense in the context of message passing:

Definition 2 *We define a task $t \in T$ to be a* synchronization point *if it has an immediate predecessor on another processor:*

$$t \in C_p \wedge \exists_{t'<t}: t' \in C_{p'} \quad \text{where } p \neq p'.$$

A synchronization point in MPI corresponds to a process that sends a message; in a shared memory task graph it corresponds to a child task.

The base of a subset L consists of the set of tasks with synchronization points.

Definition 3 *Given a set of tasks $L \subset T$, we define its base B_L as those predecessors that are not in level L:*

$$B_L = pred(L) - L.$$

If $\{L_{k,p}\}_{k,p}$ is a two-parameter covering of L, we similarly define $B_{k,p}$ as the base, for as far as it is local to p:

$$B_{k,p} = B_{L_k} \cap C_p.$$

Thus, B_{L_k} are all elements that are ready to be used in level L_k, having been computed in levels $L_{k-1}, L_{k-2}, \ldots$. If we take processor elements into account, B_{L_k} falls apart in the elements $B_{k,p}$ that are available immediately to processor p, and the remainder $B_k - B_{k,p}$ which need to be sent to process p.

Definition 4 *We call a two-parameter covering $\{L_{k,p}\}_{k,p}$ of T a set of* local computations *if*

1. *the p index corresponds to the division in processors, again, not necessarily disjoint:*

$$C_p \supset \cup_k L_{k,p}.$$

2. *the synchronization points synchronize only with previous levels:*

$$B_{k,p} \subset \bigcup_{\ell<k} L_\ell \tag{2}$$

Lemma 1 *For a local computations covering, the k index corresponds to the partial ordering on tasks: the sets $L_k = \cup_p L_{k,p}$ satisfy*

$$t \in L_k \wedge t' < t \Rightarrow t' \in \bigcup_{\ell \le k} L_\ell.$$

Tasks inside an $L_{k,p}$ need not be partially ordered.

Theorem 1 *For a given k, all $L_{k,p}$ can be executed independently.*

Proof. All predecessors that are in a different processor are also in a previous level: by condition 2 if $t \in L_{k,p}$,

$$t' <^+ t \Rightarrow \begin{cases} \text{case } t \notin B_{k,p} & \text{so } t \in L_{k,p} \\ \text{case } t \in B_{k,p} \text{ and } t' \in C_p & \text{so } t \in L_{k',p} \text{ with } k' < k \\ \text{case } t \in B_{k,p} \text{ and } t' \notin C_p & \text{so } t \in L_{k',p'} \text{ with } k' < k \text{ and } p' \neq p \end{cases}$$

We illustrate this in figure 1. Case (a) is the normal single step grid update, much like our motivating example. Case (b) shows that for a second update we would need a point on the same k-level, so this is not a well-formed local computation by the above definition. Case (c) shows how this is solved by transferring a larger halo, and computing one point redundantly.

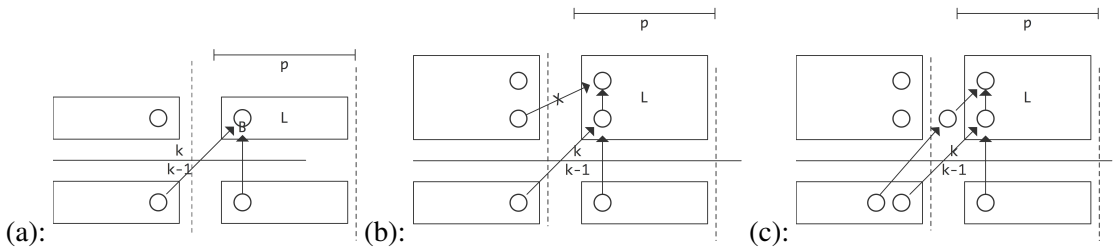

Figure 1: Three cases of L/B relations

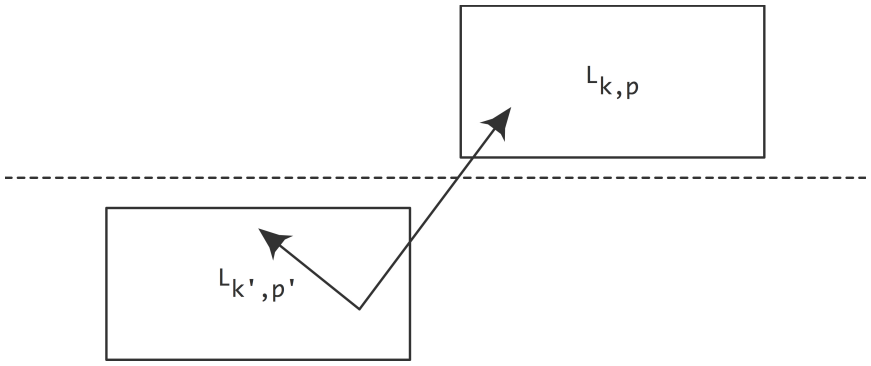

Figure 2: Overlapping execution of partially ordered local computations

2.4 Execution and synchronization

At this point we should say something about synchronization. The easiest way to visualize execution of local computations (in the above definition) is to imagine a global synchronization point after each level L_k. This reduces our model to Bulk Synchronous Parallelism (BSP) [2]. However, this is overly restrictive. Tasks in $L_{k,p}$ can start execution when their synchronization points in $L_{k-1}, L_{k-2}, \ldots$ have executed. This means that local computations in different k-levels can be active simultaneously.

In fact, if $pred(B_{k,p})$ has elements in $L_{k',p'}$, it is quite possible for $L_{k',p'}$ not to have finished execution when $L_{k,p}$ starts; see figure 2. Conversely, tasks in $L_{k,p} - B_{k,p}$ can execute before all of $B_{k,p}$ finishes. Thus, we arrive at a conception of a local computation that produces streaming output and accepts streaming input during execution. This is a first interpretation of the concept of overlapping communication and computation. We will further derive this in *IMP-06*.

We can also give a sufficient condition for overlap of communication and computation.

Theorem 2 *If we have the more restrictive condition (compared to equation (2))*

$$pred(B_{k,p}) - C_p \subset L_{k-2} \cup \cdots$$

for all k,p, the communication can be overlapped with computation.

> *Proof. All sends for level k can be initiated at level $k-2$, so they can overlap with the computation of level $k-1$.*

We can now define the *granularity* of a computation:

Definition 5 *The granularity of a computation $L_{k,p}$ is*

$$g = \min_{k,p} |L_{k,p}|.$$

The granularity of a computation is the guaranteed minimum time between synchronizations. (In regular computations one could also define granularity as the ratio between computation time and communication time.) If a computation has overlap of communication and computation it is the amount of latency that can be hidden.

3 Task graphs

There are some practical matters to observe regarding execution of the task graph. On shared memory, things are simple: there is one scheduler, and it can see the entire graph. In distributed memory, on the other hand, each address space has a partial task graph, which needs to be executed in such a way that the global task graph is respected. In this section we focus on this distributed case.

With message passing, any direct dependence between tasks corresponds to a message. In the simplest realization we would, on each address space:

- activate all tasks for that address space,
- all tasks gather incoming messages,
- when all messages for a task are satisfied, the task performs its local execution, and sends outgoing messages.

The problem with this approach is the necesary context switching between tasks, not to mention that this switching needs to be driven by the arrival of messages. Thus we would need a supervisor process of sorts.

To illustrate the basic problem, figure 3 (left) shows two tasks t,t' on the same processor that are dependent $t <_m t'$ through messages in the global graph, but have no local message dependency. In the absence of an explicit scheduler, say in a single-threaded context, we want to invoke t and t' in an order $t <_x t'$ such that

$$t <_m t' \Rightarrow t <_x t'$$

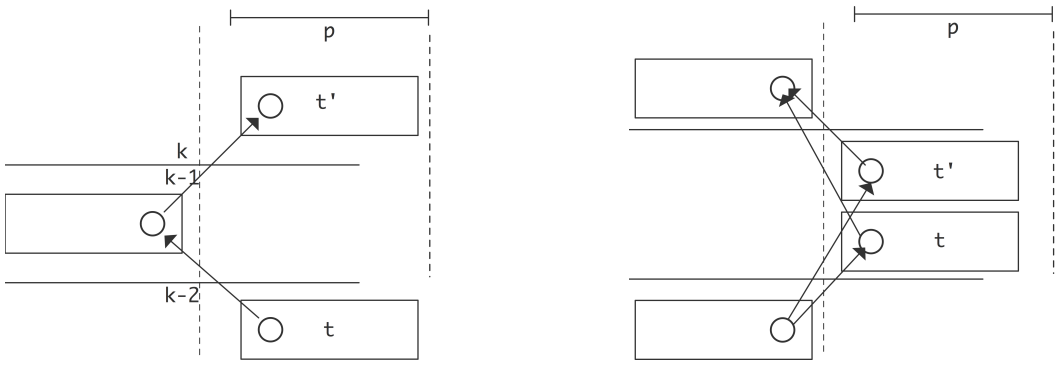

Figure 3: Two tasks t,t' without local dependencies, connected with a message dependency $t <_m t'$ (left), or truly independent in the global task graph (right)

Definition 6 *We define the execution ordering as*

$$t \in k \wedge t' \in k' \wedge t,t' \in C_p \Rightarrow t <_x t' \equiv k < k'$$

Definition 7 *Kernel ordering follows message ordering*

$$t \in k \wedge t' \in k' \wedge t,t' \in C_p : t <_m t' \Rightarrow k < k'$$

Corollary 1 *Then given two tasks on the same processor $t,t' \in C_p$, and the execute order $t <_x t'$ and kernel ordering $k < k'$ as above,*

$$t <_m t' \Rightarrow t <_x t'$$

Figure 3 (right) illustrates that this is an overspecification: tasks with no local ordering can sometimes be truly unordered.

We see this mechanism implemented as follows. With a task id implemented as $\langle \text{step}, \text{domain} \rangle$, we realize a dependency of task k,p on k',p' by installing a dependence on k',p.

```
%% mpi_base.cxx
void mpi_task::declare_dependence_on_task( task_id *id ) {
  int step = id->get_step(); auto domain = this->get_domain();
  try {
    add_predecessor( find_other_task_by_coordinates(step,domain) );
  } catch (std::string c) {
```

```
    fmt::print("Task <<{}>> error <<{}>> locating <{},{}>.\n{}\n",
       get_name(),c,step,domain.as_string(),this->as_string());
    throw(std::string("Could not find MPI local predecessor"));
  };
};
```

3.1 Clocks

In distributed processing the question of how to define a clock often comes up. First formalized by Lamport [1], a (locally defined) clock is any scalar function $C(\cdot)$ on the events on a process such that 'if a causes b, then $C(a) < C(b)$'. The definition of 'causing' is the transitive closure of the following relations:

- if a and b are events on the same process, and a happens before b, then a is said to cause b;
- if a is a sending event and b receives a message send by a, then a is also said to cause b.

In our model it is easy to define a clock satisfying this consistency criterium. If we identify events with tasks, then an event/task is uniquely described by a step-domain pair. For the clock, both locally and globally, we use the step counter. Both clauses of the causality definition imply that $s(a) < s(b)$, which is the correct clock ordering.

4 Task migration

Our concept of processors seems to go back and forth.

- A distribution is mappings from processors to indices or data. In the context of MPI these processors can be actual processors.
- On the other hand, when we derive tasks, the processor number just becomes part of the task numbers, and physical processors are sets of these tasks (definition 1).
- On the other other hand, after these tasks (with logical processor numbers) are assigned to actual processors, we may want to do task migration, which means that the logical→actual mapping needs to be virtualized.

In order to migrate a task, we need to take care of three things.

- Its predecessors need to send data elsewhere, so they need to be told that the logical-physical mapping has been updated. (This is only a problem if there is an actual message.) The problem is that they may not be expecting this update message. For this we can use the non-blocking barrier mechanism.
- Its successors will receive data from elsewhere. This may take a similar update; on the other hand, if we identify messages by tag (report IMP-07) nothing is needed there.
- The execution context needs to be moved. That is the simplest part of the story.

References

[1] Leslie Lamport. Time, clocks and the ordering of events in a distributed system. *Communications of the ACM*, 21:558–565, 1978.

[2] Leslie G. Valiant. A bridging model for parallel computation. *Commun. ACM*, 33:103–111, August 1990.