## TACC Technical Report IMP-04

# Task execution in the Integrative Model

Victor Eijkhout*

February 8, 2018



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

**Abstract**

IMP distributions are defined with respect to abstract processing entities, leading to a concept of tasks, rather than processes. In this note we show how processors can be defined, and we describe their interaction as it arises from the task dataflow.

# 1 Task execution

The Integrative Model for Parallelism (IMP) model leads to a dataflow view of task execution; see for instance *IMP-01* [1].

Dataflow is typically considered in shared memory where task coordination and synchronization is relatively easy, and its mechanisms can be largely ignored. If our model wants to incorporate distributed memory, we need to become explicit about a number of aspects of parallel task execution.

The main problem we want to address here is that a data dependency $q \to p$ (that is, $q$ preceeds $p$ by producing data for it) involves $q$ producing data that may need to be released if it is in a temporary buffer. We will give a sufficient condition for this.

We start with a detailed description of the interaction of two tasks that are in a data dependency relation.

## 1.1 Task lifeline

We formalize a *task* as a Finite State Machine (FSM) with five states. Some state transitions are ε-transitions, buth others are effected by receiving control messages from other tasks. Conversely, a task sends out control messages to other task FSMs upon entering or leaving a state.

The states of a task are the following; their only possible occurence is in this sequence.

**requesting** Each task starts out by posting a request for incoming data to each of its predecessors.

**accepting** The requested data is in the process of arriving or being made available.

**exec** The data dependencies are satisfied and the task can execute locally; in a refinement of this model there can be a separate exec state for each predecessor.

**avail** Data that was produced and that serves as origin for some dependency is published to all successor tasks.

**used** All published origin data has been absorbed by the endpoint of the data dependency, and any temporary buffers can be released.

The easiest depicting of the state transitions and their dependence on control messages is as follows:

$$\begin{array}{ccccc} q \text{ exec} & \longrightarrow & \text{avail} & \longrightarrow & \text{used} \\ & \downarrow & & \uparrow & \\ \text{request} & \longrightarrow & \text{accepting} & \longrightarrow & p \text{ exec} \end{array}$$

where the transition exec $\to$ avail of task $q$ sends a control message to all successors $p > q$ that effects the transition requesting $\to$ accepting. Similarly, the transition of $p$ to state exec releases a control message that tells its predecessors that send buffers can be released or overwritten.

Here is the full definition of the states of a task $p$, their transitions, and all control messages involved, taking into account that a task can have multiple predecessors. This is graphically depicted in figure 1.

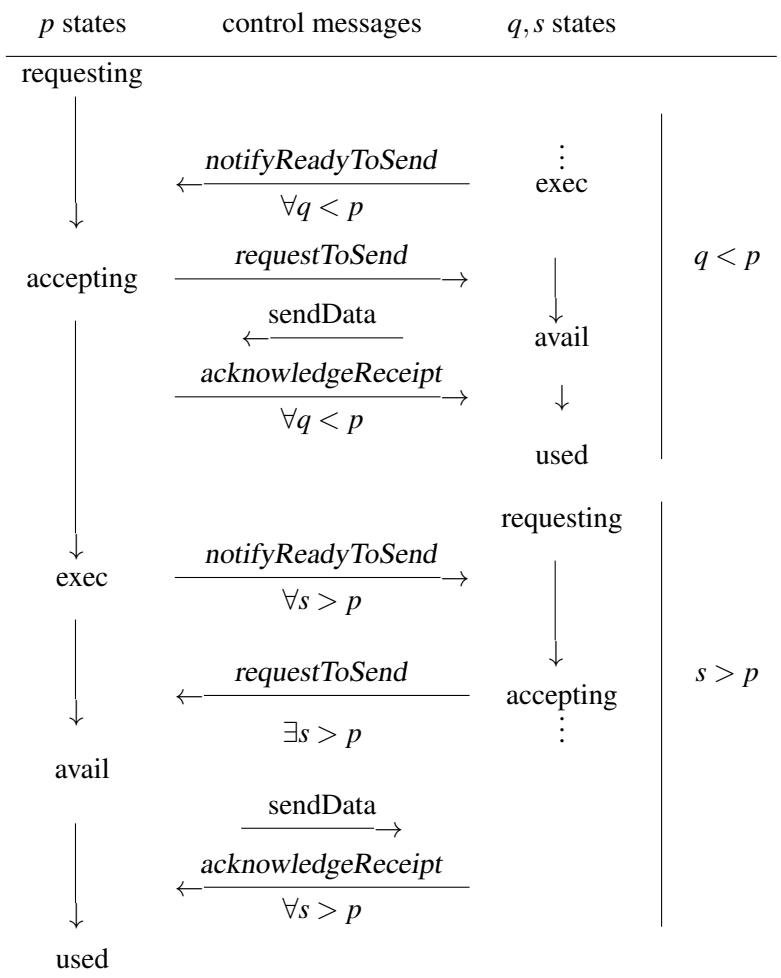


Figure 1: State diagram of a task $p$, related to predecessor tasks $q < p$ and successor tasks $s > p$

Note that *notifyReadToSend*, *requestToSend*, *acknowledgeReceipt* are *control messages*, as opposed to data.

**requesting** This is the starting state for a task.

***notifyReadyToSend*** is a control message coming from a predecessor $q$. The transition to the next state, 'accepting' is effected by receiving this message from $\forall_{q<p}$.

**accepting** This is the state where a task is receiving or reading data from a predecessor task. In this state:

- $p$ sends a '*requestToSend*' message to all $q < p$;
- $q$ does the actual send;
- $p$ sends an *acknowledgeReceipt* message, signalling that $q$ can release its send data.

**exec** is the local computation state; it is entered with an ε-transition from the 'accepting' state. The local computation is concluded with a '*notifyReadyToSend*' message to all $s > p$.

**avail** Receiving a '*requestToSend*' message from any $s > p$ effect the transition from exec to avail. This is the state where all send data for possible successors $s > p$ has been generated.

**used** is the state where all send data has been used by successor tasks; this state is entered when all successors have sent an *acknowledgeReceipt* message.

Remarks.

- The transition between the 'requesting' and 'accepting' states requires all predecessors to be publishing their data. This is conforming to the traditional dataflow model where all predecessors need to be available for a task to fire. In certain circumstances (for instance stencil operations, where the predecessors contribute halo data) it may be considered a limitation. However, relaxing this condition necessitates assumptions on the associativity of the operations that process the contributed data.
- The avail state is triggered by a successor requesting data. This is somewhat opposed to the accepting state needing all predecessors. However, it allows for such minimal publishing mechanisms as copy-on-write.
- The *acknowledgeReceipt* message is discussed further in section 1.3 below.

### 1.2 Communication and computation overlap

This story is not without its problems if we try to realize it in practice. For instance, in many cases a task takes up a dedicated hardware resource:

- In MPI a task is a code section, meaning that on one processor it does not overlap in execution with other tasks. Its execution corresponds to a definite contiguous time interval.
- In threading models a task takes up a thread, so the number of tasks that can be active simultaneously is limited. Furthermore, it depends on details of the runtime system to what extent a task can be suspended and reactivated. Thread suspension would be needed if the thread is waiting for a synchronization, and reactivation is needed when this synchronization is concluded.

In the above story this leaves us with two problems, at the receive and send stages.

- The problem with receiving affect performance. If an active task is expected to conclude all its actions (both communication and computation) in sequence, this means that the receive is posted only just in time. Consequently there will be no overlap of the incoming communication with other actions on the same processor.
- The problem with sending is a logical one. After a send has been posted, a task can not complete until the corresponding receive has been executed. This gets in the way of the concept just explained of task execution taking a finite and contiguous time interval

The practical solution to this problem can not be formulated in a model with only tasks: we need to invoke the definition of processors in *IMP-05* [2] as subsets of tasks, presumably on a shared

address space. Let task $t$ have coordinates $(s,p)$ (kernel step, virtual process), and a predecessor be $t'=(s',p')$ where $s'<s$ and $p'\neq p$. In that case we let $t'$ perform the *requestToSend* action of $t$, and $t$ performs the *acknowledgeReceipt* actions for $t'$.

This scheme has some implications for data management: $t$ will deallocate the send buffers of $t'$, and $t'$ needs to know the β-vector of $t$. In particular, this β-vector needs to be allocated fairly early, and it may interfere with memory management before $t$'s local execution is performed.

### 1.3 Acknowledge receipt

The *acknowledgeReceipt* message deserves special attention. It runs opposite to the actual data dependency, thus it doesn't have a place in a dataflow model. In certain special cases it exists. For instance, in MPI it shows up in the sending process as an `MPI_Request` being fullfilled. On the other hand, with typical shared memory task graphs no such explicit acknowledgement exists.

There are various reasons for needing this message. For instance, in MPI the sending process can have allocated buffers that need to be released or reused. More tricky, suppose that the sent data needs to be overwritten by a subsequent task, for instance in doing repeated Finite Element grid updates. Without explicit acknowledgement, the processor executing the tasks can not know when it is safe to do so.

(The presentation of IMP has so far implicitly used a functional / Static Single Assignment (SSA) model, where data is created but otherwise never updated. Functional languages 'solve' this by having a *garbage collector*. In scientific applications that is not tenable, so we need to find a way to support mutable data.)

Fortunately, our synchronization model gives us a way around this.

#### 1.3.1 Looped reasoning

We define a condition on task graph to allow us to reason circuitously.

**Definition 1** *A task graph is a* looped task graph *if*

$$\forall_v \forall_{v'\in succ\left(pred(v)\right)} : succ(v)\cap succ(v')\neq\emptyset \tag{1}$$

Figure 2 shows a looped and not-looped task graph.

We give two equivalent formulations of (1):

$$\forall_{u\in pred(v)}\forall_{v'\in succ(u)}\exists_{w\in succ(v)} : w\in succ(v')$$

and

$$\forall_u \forall_{v,v'\in succ(u)} : succ(v)\cap succ(v')\neq\emptyset$$

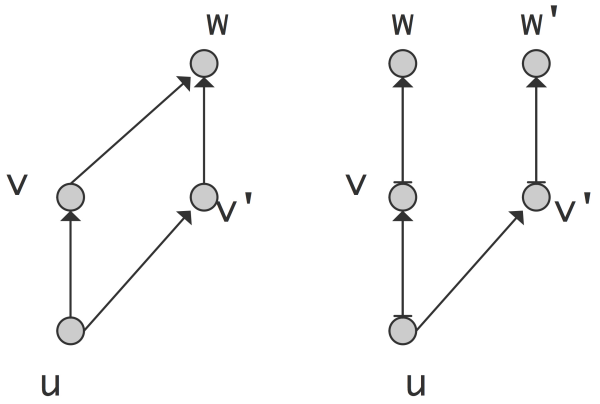


Figure 2: A looped (left) and non-looped (right) task graph

In many cases, the condition for a graph being looped is easily derived from the structure of the *signature function*.

**Lemma 1** *Let the signature function be an invertible mapping $f$, or sum of such mappings. Then*

$$pred(v) = v \cup f^{-1}v, \quad succ(v) = v \cup fv \Rightarrow succ(pred(v)) = f^{-1}v \cup v \cup fv$$

*Proof. We need to check three cases.*

1. $v' \in v$: $succ(v) \cap succ(v') \neq \emptyset$ *trivially.*
2. $v' \in fv$:

$$succ(v) = v \cup fv, \quad succ(v) = fv \cup ffv$$

   *making the intersection $fv$.*
3. $v' \in f^{-1}v$:

$$succ(v') = f^{-1}v \cup v$$

   *making the intersection $v$.*

### 1.3.2 Inferring task conclusion

In order to analyze this situation, we reduce the above five *communication states* to three *activation states*, called 'A,B,C' for 'Active', 'Broadcasting', 'Concluded'.

**A** A task is 'Active' if it is in states 'accepting' or 'exec'; that is, it is incorporating data dependencies into its local computation.

**B** A task is in state 'Broadcasting' if it is in state 'avail'; that is, it is publishing its data to tasks that depend on it.

**C** A task is in state 'Concluded' if it is in state 'used'; that is, its produced data is successfully dispatched and the task is finished in every sense.

The state 'request' is meaningless in that tasks can be created in it.

The basic laws of task activation states, in terms of kernels and distributions are the following:

1. If a task is Active, its predecessors are Active or Broadcasting:

$$p \in A \wedge q \in pred(p) \Rightarrow q \in B.$$

2. If a task is Broadcasting, its predecessors are Concluded:

$$p \in B \wedge q \in pred(p) \Rightarrow q \in C.$$

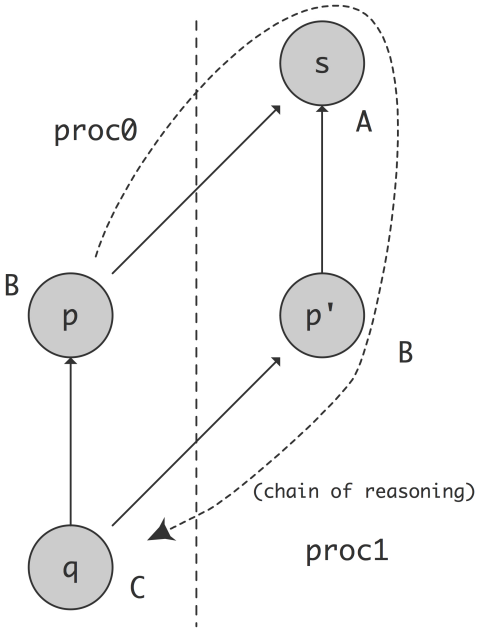


Figure 3: Chain of reasoning in theorem 1.

We now have a relation between the activation states of tasks.

**Theorem 1** *(McCalpin-Eijkhout Garbage Eliminator) For any task $p$ in a looped task graph, if it has an active successor, that is,*

$$\exists_{s \in succ(p)} : A(s)$$

*then all its predecessors are concluded:*

$$\forall_{q \in pred(p)} : C(q).$$

*Proof. Task $p$ detects a active successor by getting a requestToSend.*

$$\begin{aligned}\exists_{s>p}: \mathit{RequestToSend}(s,p) &\Rightarrow \exists_{s>p}: A(s)\\ &\Rightarrow \exists_{s>p}\forall_{p'<s}: \mathit{Sending}(p',s)\\ &\Rightarrow \exists_{s>p}\forall_{p'<s}: B(p')\\ &\Rightarrow \exists_{s>p}\forall_{p'<s}\forall_{q<p'}: \mathit{AcknowledgeReceipt}(p',q)\\ &\Rightarrow \exists_{s>p}\forall_{p'<s}\forall_{q<p'}: C(q)\end{aligned}$$

*Since $p$ is one of these $p'$, we have proved that its predecessors are concluded.*

From theorem 1 we can prove

**Theorem 2** *If every processor $P$ contains a 'start-to-finish' sequence of tasks, that is,*

$$P \supset \langle t_0,\ldots,t_k\rangle, \qquad \text{where} \quad \begin{cases} t_0 \in T_0 \\ t_k \in T_\infty \\ t_i < t_{i+1} \end{cases}$$

*then no garbage collection is needed.*

*Proof. Every task has a successor on the same processor, which can perform the act of freeing the send buffer.*

### 1.4 Task failure detection

The above 'loopy' reasoning can also be used for a different purpose, namely in the service of resilience.

Let us define a failing task as one that has received its input, but never sends its output. In figure 4 task $p$ fails in this sense. Now if task $s$ goes into state $A$, at least one of its predecessors $p'$ has to be in state $B$, so its predecessors $q$ in turn have to be in state $C$. Thus, if $q$ is in state $C$, $p$ has to be sending at some point. If we have a timing model for the program, we can stipulate a time-out that lets $s$ decide that $p$ is lost to us.

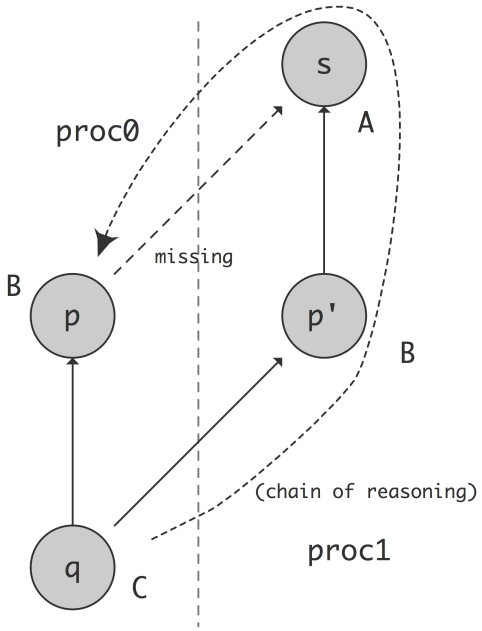


Figure 4: Chain of reasoning in theorem **??**.

## 2 Task synchronization in practice

In section 1 we described the abstract concepts of tasks synchronization.

***MPI*** We describe MPI task interactions in figure 5.
We use the following convections:

1. An arrow originating at a statement

   Stmt $\longrightarrow$

   indicates that the signal is caused by the completion of the statement.
2. An arrow pointing to a statement

   $\longrightarrow$ Stmt

   indicates that the completion of the statement is caused by the arrival of the signal. Example: a 'Wait' call can be made at any time, but it only completes when the actual data signal arrives.
3. An 'X' indicates that no user level call corresponds to the originating or arrival of a signal.

In this figure we see for instance how the usual three-way handshake only corresponds to a single user action on the sender; the receiver has two user-level calls, one to indicate the willingness to receiver, and one to signal to the user that the receive is completed.

| MPI calls, $p$ | control messages | MPI calls, $q$ |
|---|---|---|
| | | exec |
| X | $\xleftarrow[\forall q < p]{\textit{notifyReadyToSend}}$ | Isend |
| Irecv | $\xrightarrow{\textit{requestToSend}}$ | X |
| Wait | $\xleftarrow{\text{sendData}}$ | X |
| X | $\xrightarrow[\forall q < p]{\textit{acknowledgeReceipt}}$ | Wait |
| exec | $\xrightarrow{\textit{notifyReadyToSend}}$ | et cetera |

Figure 5: MPI state diagram of a task $p$, related to predecessor tasks $q < p$

***Task graph*** In a shared memory DAG environment this is all a lot simpler; see figure 6. The *notifyReadToSend* message is not an actual message: the task sets a flag that can be read by other tasks. Likewise there is no data transfer: the receiving task can simply read the data from shared memory.

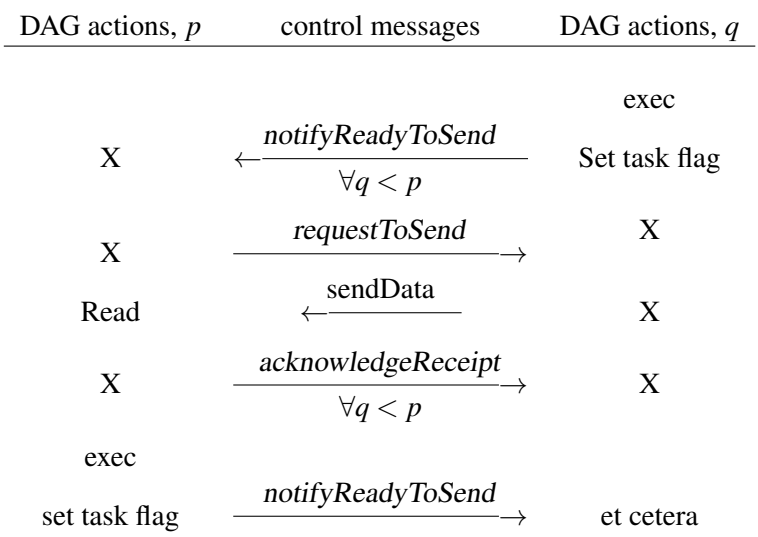


Figure 6: DAG state diagram of a task $p$, related to predecessor tasks $q < p$

## 3 Implementation

After the discussion in section 1.2, this is what task execution in the IMP code looks like. The 'post' and 'xpct' messages are the actions of a successor task. SPELL THIS OUT!

```
%% imp_base.cxx
void task::execute() {
  auto d = get_domain(); auto outvector = get_out_object();

  try { outvector->allocate();
  } catch (std::string c) {
    throw(fmt::format("Error during outvector allocate: {}",c)); }

  if (get_has_been_executed()) {
    fmt::print("Not executing {}\n",this->get_name());
    if (get_exec_trace_level()>=exec_trace_level::EXEC)
      fmt::print("Task bypassed <<{}>>\n",this->get_name());
    return;
  }
  if (!outvector->lives_on(get_domain()))
    return;

  bool trace = get_exec_trace_level()>=exec_trace_level::EXEC;
  if (trace)
    fmt::print("Task execute <<{}>>\n",this->get_name());

  if (!get_has_been_optimized()) {
```

```
    try { notifyReadyToSend(get_send_messages(),requests);
    } catch (std::string c) { fmt::print("Error <<{}>> in notifyReadyToSend\n",c);
      throw(fmt::format("Send posting failed for <<{}>>",get_name()));
    }
    try { acceptReadyToSend(get_receive_messages(),requests);
    } catch (std::string c) { fmt::print("Error <<{}>> in acceptReadyToSend\n",c);
      throw(fmt::format("Recv posting failed for <<{}>>",get_name()));
    }
  }

  auto objs = this->get_beta_objects();
  if (!has_type_origin() && objs.size()==0)
    throw(fmt::format("Non-origin task <<{}>> has zero inputs",get_name()));
  if (all_betas_live_on(d)) {
    if (outvector->get_split_execution())
      try { // the local part that can be done without communication.
local_execute(objs,outvector,localexecutectx,unsynctest);
      } catch (std::string c) { fmt::print("Error <<{}>> in split exec1\n",c);
throw(fmt::format("Task local execution failed before sync")); }
  } else fmt::print("skip pre-exec for missing halo\n");

  if (trace) fmt::print("Waiting for {} requests\n",requests->size());
  try { requests->wait(); }
  catch (std::string c) { fmt::print("Error <<{}>>\n",c);
    throw(fmt::format("Task <<{}>> request wait failed",this->get_name())); }
  catch (...) { fmt::print("Unknown error waiting for requests\n");
    throw(fmt::format("Task <<{}>> request wait failed",this->get_name())); }
  record_nmessages_sent( requests->size() );

  if (all_betas_live_on(get_domain()) && all_betas_filled_on(d)) {
    try { // for product this executes the embedded queue; otherwise just local stuff
      if (trace) fmt::print("local execute\n");
      if (!outvector->get_split_execution())
local_execute(objs,outvector,localexecutectx);
      else
local_execute(objs,outvector,localexecutectx,synctest);
    } catch (std::string c) { fmt::print("Error <<{}>> in split exec2\n",c);
      throw(fmt::format("Task local execution failed after sync")); }
  } // else fmt::print("skip pre-exec on {} for missing halo\n",d);

  notifyReadyToSend(get_post_messages(),pre_requests);
  acceptReadyToSend(get_xpct_messages(),pre_requests);

  result_monitor(this->shared_from_this());
  set_has_been_executed();
```

```
}
```

This connection between tasks is made in `algorithm::optimize`:

```
%% imp_base.cxx
void algorithm::optimize() {
  for (auto t=tasks.begin(); t!=tasks.end(); ++t) {
    if (!(*t)->has_type_origin()) {
      auto domain = (*t)->get_domain();
      auto deps = (*t)->get_dependencies();
      for (auto d=deps.begin(); d!=deps.end(); ++d) {
int originkernel = (*d)->get_in_object()->get_object_number();
auto othertsk = find_task_by_coordinates(originkernel,domain);
{ auto lift = (*t)->lift_send_messages();
  othertsk->add_post_messages(lift); }
{ auto lift = (*t)->lift_recv_messages();
  othertsk->add_xpct_messages(lift); }
othertsk->set_pre_requests( (*t)->get_requests() );
(*t)->set_has_been_optimized();
      }
    }
  }
}
```

- The 'lift' commands return the message vector from a task, and zero out the container for it on the task. Thus, these lines effect a transfer of the message vectors between two tasks.
- The `othertsk` is a task on the same processor, belonging to the kernel of the predecessor task. **We need to prove explicitly that this has the right properties.**

## References

[1] Victor Eijkhout. IMP distribution theory. Technical Report IMP-01, Integrative Programming Lab, Texas Advanced Computing Center, The University of Texas at Austin, 2014. (Included in source code repository.).

[2] Victor Eijkhout. Processors in the integrative model. Technical Report IMP-05, Integrative Programming Lab, Texas Advanced Computing Center, The University of Texas at Austin, 2014.