Efficient Termination Detection for Asynchronous Parallel Computations

Nabeel Al-Fayoumi and Eric N. Hanson

Rm. 301 CSE, P.O. Box 116120
CISE Department
University of Florida
Gainesville, FL 32611-6120
nabeel@cis.ufl.edu
hanson@cis.ufl.edu
http://www.cis.ufl.edu/~nabeel/
http://www.cis.ufl.edu/~hanson/

Technical Report 96-036

Key words: *parallel processing, shared-nothing multiprocessors, asynchronous parallel computation, termination detection*

Abstract

A solution is presented to the problem of quickly detecting the end of an asynchronous parallel computation, in which processors exchange task request messages asynchronously, and termination occurs when all processors are idle and no messages are in transit. The proposed termination detection scheme is based on a finite automaton consisting of only two states, and one possible transition between states. This solution is superior in many cases to *ad hoc* solutions such as polling processors periodically to see if they are done, because it is simple, efficient, and has fast response time. Simulation results show that the scheme has low overhead.

1 Introduction

In a shared-nothing parallel computer system [8, 11], each processor has its own memory and hard disk units, and different processors in the system communicate via message passing. In this paper we address the problem of testing for termination of a parallel computation on this type of machine when the computation is *asynchronous*, i.e. when control of the computation is distributed among all the processors, who communicate with each other only by exchanging messages requesting some task to be performed.

The architectural model we assume here is shown in Figure 1. It consists of a number of processors, each of which has its own memory, disk, and task queue. The processors are connected via an interconnect. One of the processors will be designated to bear the task of planning, initiating, and monitoring the parallel execution of some computation, and is usually referred to as the coordinator. Each processor receives requests to perform tasks from other processors via its task queue. Based on this, we will describe the behavior of an asynchronous computation run on top of this platform, and describe the requirements for detecting termination of the computation.

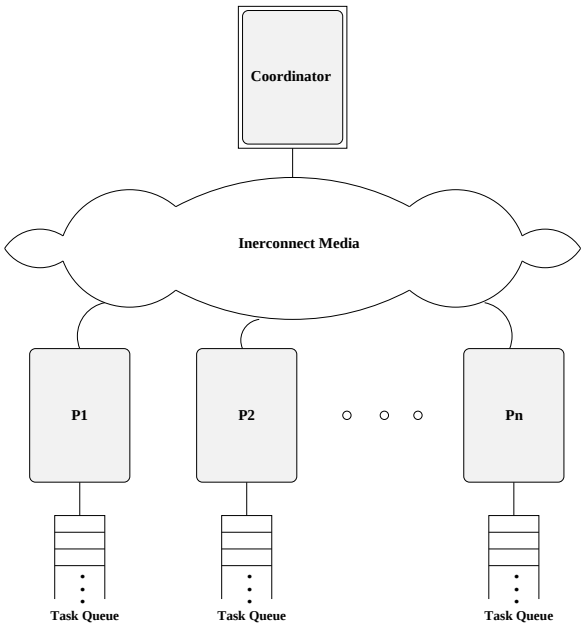

Figure 1: A shared-nothing system

Assuming that the underlying network is fault tolerant, messages sent by one processor to another are guaranteed to reach their destination. Each message will hold necessary information about 1) one or more computation tasks, 2) the processor to receive and carry out those tasks, and 3) the processor that directly originated the task(s), or indirectly caused them to be generated. Since we are concerned with asynchronous parallel computation, we define it as having the processors in the system execute fragments of a parallel computation without any synchronization points enforced by the coordinator. Accordingly, we can order the steps involved in executing an asynchronous parallel computation and the possible resulting events as follows:

1. Task queues are initialized by receiving local tasks assigned by the coordinator.
2. The coordinator will broadcast a "GO" massage to start parallel computation at all sites.
3. Each processor will start executing the tasks in its queue, which might cause the generation of further tasks to be sent to other processor(s) for execution (those tasks will be eventually received and enqueued in that processor's task queue).
4. The system will terminate when all the processors in the system have finished processing all the original and generated tasks, i.e. all the task queues are empty, and no messages are in transit from one processor to another.

The problem in the previous sequence of events shows up in the last step, where *we need to detect the termination* in an efficient way, with low run time, message, memory, and computation overheads, and a fast response time. In other words, we are simply trying to come up with a way to determine when all the processors in a shared-nothing environment have completed an asynchronous parallel computation. In such setup, the coordinator will have to schedule the parallel computation, start activity, and finally detect the *termination point* as soon as it occurs.

There may be many solutions to this problem [9], but most of them will fail to meet the efficiency, low overhead, and response time requirements. For example, one possible solution is to use polling,

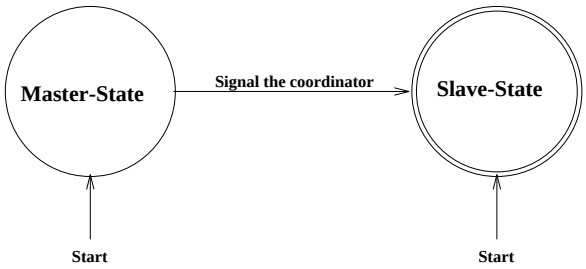

Figure 2: Processor State Diagram

where the coordinator would poll all the processors periodically, and inquire about their status (idle or active), so that when it gets idle responses from all, it can realize that the parallel computation has been completely carried out. The problem in this simple solution is the high message passing rate between the coordinator and other processors, which would cause a bottleneck when there is a large number of processors, which in turn restricts the scalability of the system. Then, there is the issue of response time which might be very sensitive for some application [1], and this solution never guarantees a fast response time, because it might detect the termination up to a full polling interval after it occurred. Hence, we will propose a solution in the following section, which, unlike polling, takes into consideration efficiency, overhead, and response time.

2 The theory behind a simple solution

We propose a simple automaton to control the state transitions of the system processors. The coordinator detects termination using messages sent to it by each of the processors when they make their transition to their final state. We start by explaining the states and transitions in the solution automaton. Afterwards, we suggest suitable data structures to operate side-by-side with the automaton. As shown in Figure 2, any processor in the system can be in one of two states: *Master* or *Slave*. All processors in the system that are assigned work by the coordinator start in the Master state, thus we later refer to those as *masters*. Other processors (if any) will start in Slave state, so we refer to them as *slaves*.

A master may send computation assignments (messages) to any other processor depending on the application. Therefore, we think of this as having the destination processor work on behalf of that master, where the destination processor can be in either state (Master or Slave). The master should eventually receive some kind of acknowledgment from any processor that's working on its behalf once it has finished doing so. A slave processor can be either idle or working on behalf of other processor(s). A key idea here is that *any master is solely responsible of monitoring its activities, both local and nonlocal*. In this sense, when the master finishes all its local and nonlocal activities, it has to make a transition to Slave state. Consequently, it's responsible for reporting that event to the coordinator. Once a master goes into Slave state, it can never go back to Master state again. This implies that each master in the system will report once and only once to the coordinator when it makes the only possible transition (from master to slave). It can be proven that once the last master processor reports its transition event, this scheme guarantees that a parallel computation has been completed, and there are no residual activities in the system at this point. Hence, if the coordinator receives transition messages from the master processors in the system, it can identify the *termination point* (where there are no masters in the system any more) as soon as

the last master sends its transition signal.

Theorem When the last processor in Master state moves to its Slave state, then it and all other processors must be idle.

Proof: Recall that during a single parallel computation, if a processor enters the Slave state after being in the Master state, it can never go back to Master state again. If the processor ever becomes active again, it will be in Slave state. A processor will never acknowledge any request message received from another processor until it is certain that it has completely finished processing that request. Hence, a master processor will never be able to go into Slave state until it has received all acknowledgments for all the requests performed by any other processor on the behalf of that master.

Based on the above discussion, we will prove the theorem by contradiction. Suppose that the coordinator has received transition messages from all processors in the system but the last one, which is still in Master state. At this point, this last processor finishes its work, and sends the coordinator its transition message. Assume that after the coordinator receives that message, there are still some other active processors. These active processors must be working for a processor in Master state. But this is a contradiction, since there are no processors in Master state. *Hence, when the last master processor goes to Slave state, all other processors in the system must be idle, and the coordinator can safely conclude that the parallel computation has terminated.*

3 Direct acknowledgment

Before we start talking about the implementation of the suggested solution, we have to establish an important concept, which will be used in the rest of our paper. In the normal case, when a processor performs a task on behalf of another, it may spill over and cause some task(s) to be sent to another processor. If a master sends tasks to a slave, then processing that task causes further task(s) to be sent to yet another slave, and so on. This could result in a long chain that starts with a master on one end followed by a number of slaves. Each of the slaves will wait for the acknowledgment from the lower level slave in the chain. This chaining effect complicates things, and can cripple some schemes in which this is an undesired behavior. The alternative is to have the slave attach the original master ID to any request due to processing some task on its behalf. Then, the slaves will *directly acknowledge* the master rather than the slave who actually sent the message on behalf of the master. In the following sections, we assume that slaves always directly acknowledge the original master. This avoids passing acknowledgments along a chain of slaves.

4 A High Overhead, Fast Response Time scheme

To efficiently implement the previously proposed solution to the termination problem, simple data structures are needed to record essential information. To keep track of the number of master processors in the system, the coordinator will maintain a special counter called the *master processors counter* (MPC). Otherwise, all processors will have a *slave activity counter* (SAC), which will keep track of the number of task requests generated directly or indirectly by the processor. A processor will receive messages which hold necessary information such as: origin processor ID, number of tasks, task code, and task body, and maybe some other fields depending on the application. In this scheme, as soon as the processor executes the task of a dequeued message, it will acknowledge its

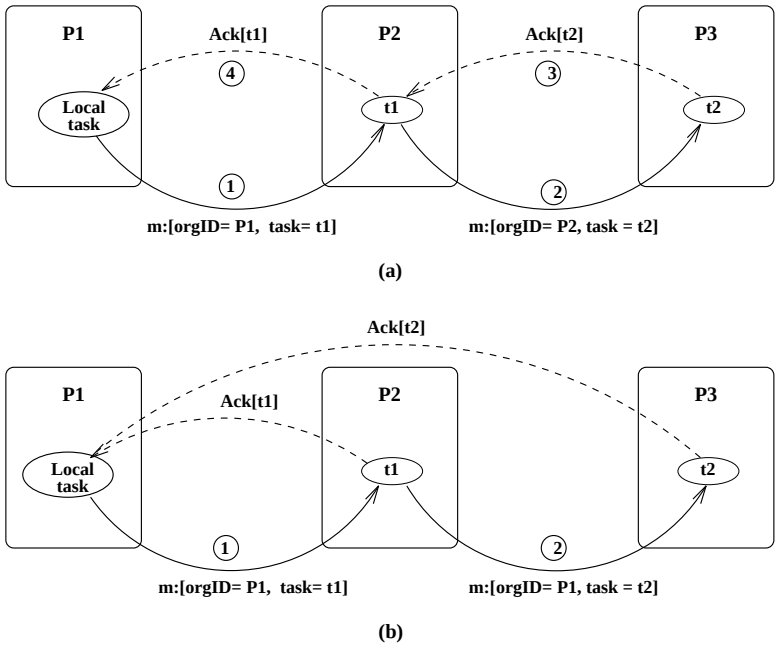

Figure 3: (a) Chained acknowledgment (b) Direct acknowledgment

origin processor. As the origin receives an acknowledgment, it will decrement its SAC. When the SAC becomes zero, the origin processor can detect the end of its foreign activities.

An undesired side effect of this scheme is what we call *acknowledgments blizzards*, which will occur because the different processors in the system will acknowledge individual tasks. If those are fine grained, there could be a huge number of exchanged tasks, consequently producing an equal number of acknowledgments. This might cause contention and high overhead. Hence, we need to reduce the intensity of messages exchanged between processors in the system. We will overlook the details of this scheme, because we are primarily interested in the scheme discussed in the next section, which avoids acknowledgment blizzards and results in low message overhead.

5 A Low Overhead, Fast Response Time scheme

The following solution to the termination detection problem uses an acknowledgment message batching technique to greatly reduce the total number of acknowledgment messages, eliminating the problem of acknowledgment blizzards. In this solution, an important data structure called the *slave activity table* (SAT) is added at each processor. This table has an entry for each processor in the system, and each of these entries consists of the following fields:

- An *Is-Master* field which holds a *true* value if the processor corresponding to this entry is a master of the local processor. Otherwise it will contain the value *false*. Initially, all the entries of the SAT will have their *IS-Master* field set to false.
- A *Count* field which holds the number of task requests received from that processor yet to be processed on its behalf.

Once a message arrives at a processor, it first checks if it has received any messages prior to this one from the same origin processor. If true, then the origin already knows that this processor is

one of its slaves, and the slave will just queue the message. Otherwise, the origin doesn't know that this processor is going to be its slave. Therefore, this processor will send a "new-slave" message to notify the origin processor that it has a new slave.

If a processor sends a task message to another, then the sender has to block until it receives an acknowledgment from the task receiver. On the other side, when a processor receives a task message from another, it will send it a "continue" acknowledgment only when it has finished queuing the task request, and if necessary, has informed the origin processor it is now doing a task for it. This is essential to avoid a possible race condition, in which a master could be deceived that it was done when it was not. The following method can be used to coordinate the blocking protocol along with the collaboration of the task message receiving processor:

```
processor::SendRequest(m,p) // m is the message to be sent, p is the destination processor of m
{
  SendMessage(to: p, header: "task-message", body: m);

  // Now block until receiving a continue message from p:
  ReceiveMessage(from: p, header: "continue");
}
```

A processor invokes the following method to handle any received request messages:

```
processor::ReceivesRequest(m) // m is the received message
{
  if (SAT[m.origin].IsMaster == false) {
      SendMessage(to: m.origin, header: "new-slave");
      SAT[m.origin].IsMaster = true;
  }
  SAT[m.origin].count++;
  enqueue(m); // put the message in local work queue for this processor

  // Ack the request sender to let it continue its work:
  SendMessage(to: m.sender, header: "continue");
}
```

The receiver will use the queue to process messages in the order they where received. Whenever a master processor receives an acknowledgment from one of its slaves, it will invoke the following method:

```
processor::ReceivesAck(m)
{
  --SPC;
  if (SPC == 0)
     {
       state = slave; // move to slave state
       SendMessage(to: Coordinator, header: "transition-to-slave");
     }
}
```

When a master processor finishes all its local work and its SPC contains zero, i.e. it has finished all the work assigned to it both locally and non-locally, it will make the transition to Slave state. Afterwards, it will also send the coordinator a "transition" message.

When the coordinator receives a transition message from a master processor (which has become a slave), it will decrement its MPC to record the fact that master processors have decreased by one. When the last master processor sends the coordinator its transition message, the MPC will be decremented to zero. Then, the coordinator will immediately realize that all processors have made transitions to Slave state, which implies that the parallel computation is done.

Furthermore, processors will use grouping, by holding the acknowledgments to their masters until their queues are empty (see the ProcessMessages() method below), and they've finished processing all the received requests. Only then will they acknowledge their masters. This will tremendously reduce the number of acknowledgments flowing from the slaves to the masters in the system (it is a type of batching technique). This avoids the problem of acknowledgment blizzards. The method slaves use to acknowledge masters as needed can be written as follows:

```
processor::SlaveIsDone() // called when slave has completed work on the last entry in its input queue
{
  for(i=0; i<= # of system processors; i++)
     {
       if(SAT[i].Is-Master == true)
          SendMessage(to: master, header: "ack");
     }
  Reset(SAT); // set all Is-Master fields in the SAT to false
}
```

Each processor can use the following method to handle the messages in its task queue. It's assumed that the operation to be carried out by the processor is embedded in the message, and is done by applying the "process(m)" method:

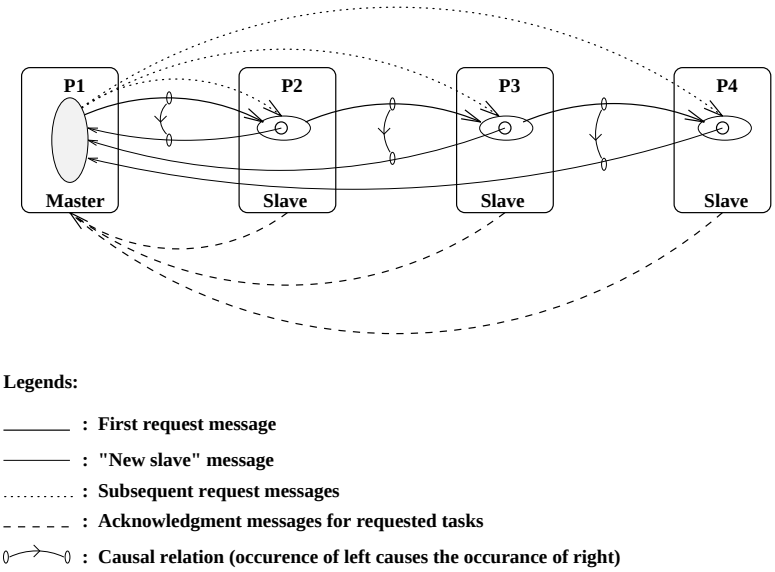

Figure 4: Different message types, and their causal relations

```
processor::ProcessMessages()
{
  while( task-queue not empty )
        {
          m = task-queue.dequeue();
          process(m);
          SAT[m.originID].count--;
        }
  SendAcks();
}
```

Figure 4 illustrates the different message types that can be exchanged between a master and its slaves under this scheme. It also depicts the causality relation between the first request message from a master to a slave, and the corresponding "new slave" message that should be returned immediately from the slave to that master. Any two messages are independent, unless they have a causality relation between them.

6 Simulation Results

We developed a simulator to test our scheme, and tried to mimic the real parallel environment as closely as possible. The simulated system consists of a number of processors, one of which is assigned the role of coordination. The simulation starts by picking a number of master processors. Then a random exchange of messages is primed at each master. Once a slave processor receives a message, it follows the termination scheme, and may generate more messages to be sent out on behalf of the sender of the message. The stopping condition we have is the number of hops, which is the maximum number of activities a message from a master can linearly cause on other processors. To simulate a real environment, the simulator allows only a time slice for each process in a round robin fashion, in order to try to detect any race condition effects if present. The variable parameters

Variables	*Message Stats*	
Msg Fan Out	Overhead Msgs	(Overhead/ Task msgs)%
4	3358	0.07516
16	3306	0.06612
64	3262	0.06524
256	3222	0.06444
1024	3186	0.06372

Figure 5: Simulation of 16 master processors out of 1024 total processors for 5 million task messages.

in our simulator are: number of masters, number of slaves, maximum number of generated tasks due to executing one task, time slice, and some other stopping parameters that control the number of total generated tasks throughout the simulation. The following table shows the overhead of the scheme under the variation of the message fan out (indicated in the first column), which is the maximum number of messages generated as a result of executing a task. Variations of other parameters either have the same effect, or have a negligible effect. Therefore, we didn't bother to list statistics for them here.

To test for proper operation of the scheme, we had the coordinator detect termination, then ran a verification pass on the data structures to ensure they all contained the expected correct data at that point, and finally printed all message statistics. Throughout the extensive experiments we attempted, the simulator never failed to detect termination, indicating that the scheme works fine in our experiments, and should do so in a real parallel environment.

The tables below summarize the effects of varying the message fan out on the overhead of the scheme, and the corresponding ratio between that overhead and the number of other messages exchanged between processors in the system for different master processor assignments (figures 5, 6, and 7). In all the figures, the results given are for a simulation involving 1024 total system processors. Throughout our preliminary experiments, we could clearly see that the growth of the overhead messages is independent of the increase in the number of task messages. Hence, we fixed the number of task messages generated in the rest of the experiments for which results are documented here. The different overhead messages (ack, new slave, and transition messages) are dependent on the number of masters and their slaves in the system. Thus if we fix the number of master processors, we neutralize its effect, and the overhead becomes solely dependent on the number of slave processors. The number of processors in a shared-nothing system is normally limited to at most a few thousand, consequently limiting the maximum number of overhead messages when the scheme presented here is used. Therefore, when the number of task messages is large, the ratio of overhead messages to the total number of task messages is guaranteed to always be reasonably low.

Variables	*Message Stats*	
Msg Fan Out	Overhead Msgs	(Overhead/ Task msgs)%
4	68050	1.36083
16	54456	1.08912
64	52122	1.04244
256	48900	0.97025
1024	51016	1.02032

Figure 6: Simulation of 256 master processors out of 1024 total processors for 5 million task messages.

Variables	*Message Stats*	
Msg Fan Out	Overhead Msgs	(Overhead/ Task msgs)%
4	272590	5.44777
16	218080	4.34874
64	208370	4.11869
256	190884	3.64523
1024	204614	3.44110

Figure 7: Simulation of 1024 master processors out of 1024 total processors for 5 million task messages.

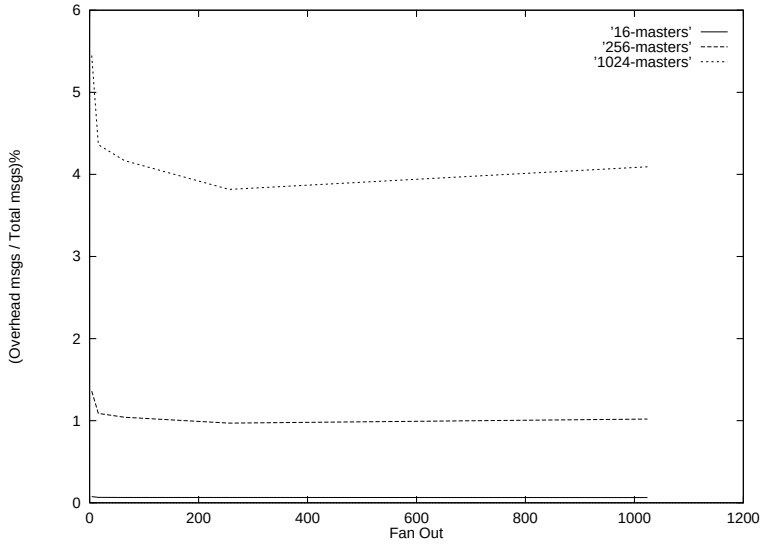

Figure 8: Overhead percentage for three master processors assignments, with 1024 total processors.

7 Applications

The first application of our proposed scheme is detecting the termination of the asynchronous parallel trigger action execution in a parallel active database system. In a parallel active database system, like any other database system, updates are submitted by users to alter the state of the database. The main characteristic of active databases is the provision of a trigger (or rule) system [12, 6]. The main function of a trigger system is to provide a tool to define triggers, test any update that can alter the state of the database against the triggers, and execute the actions of the matched triggers. Triggers are defined using a rule language that is usually an extension of the query language. A trigger consists of three basic components: An event under which the trigger is tested, a condition part that defines the predicate used for matching data, and finally the action part, which is the action(s) to be executed when the trigger matches the data of an update. A database update transaction can contain a number of updates to the database. When a transaction is submitted, all of its updates are tested by the trigger system prior to making the updates permanent. The point at which the updates are made permanent is the point at which the transaction is said to *commit*. After an update is processed by the query processor, the update is passed to the trigger system which checks if that update matches the condition of any predefined triggers. If a match is detected the data is recorded in special data structure for the trigger action processing phase.

After all the updates of a transaction have been processed, and before the transaction commits, the actions of all the matched triggers have to be executed. When trigger actions execute, they can in turn cause other triggers to fire. This is the source of asynchronous execution in this application – it is not possible to structure trigger execution as a synchronous sequence of parallel steps. An asynchronous parallel computation is primed to run the actions of matched triggers, and thus, a termination algorithm has to be utilized to detect the end of trigger execution for the transaction to commit. In our case, we designed a parallel trigger system to be integrated with a parallel database system called Paradise [2]. In our trigger system, we use discrimination networks [5] similar to those used in production systems [4, 7] for trigger condition matching.

From a user's point of view, the result of a submitted transaction is supposed to be reported as soon as it commits with the least possible response time. In a parallel database system, the coordinator is the processor responsible for the client program interface, parallel activity scheduling, etc. To reduce response time, the coordinator needs to detect the end of an asynchronous parallel computation, i.e. the trigger action execution at the end of an update transaction, as soon as possible.

Polling is a possible solution for this same problem, where the system coordinator can periodically poll all the other active processors, and when everybody indicates that they have finished execution, the coordinator can detect the end of that asynchronous activity and commit the corresponding transaction. The problem with this approach is the high overhead and slow response time. When using polling, the coordinator will have to send polling messages every period of time (polling cycle) to all processors in the system, and receive corresponding answers. Even when the asynchronous computation is done, the coordinator will have to wait for the beginning of the polling cycle to start polling and detect the termination of that parallel computation, which most likely will result in higher response time than if the scheme proposed in this paper is used. We surveyed other parallel termination algorithms [3, 9], but they had disadvantages for the parallel trigger termination problem. Hence, we thought that a novel and simple scheme was needed to solve our termination problem, which is a special case of the asynchronous parallel computation termination problem. This motivated the solution presented in this paper.

Another possible application of our scheme could be in the area of parallel production systems, where rule actions are executed under an asynchronous parallel computation [10]. A termination algorithm is necessary in this application to detect the end of that computation. We believe that there are many computational domains in which our scheme can serve as a helpful tool to detect the termination of a parallel asynchronous computation. Actually, in any asynchronous parallel computation where processors are exchanging information, and data can be in transition at any point of time, some kind of termination algorithm is needed to detect the complete end of such an activity. We are currently studying the possibility of applying our algorithm to computational domains where asynchronous parallel computations play a major role.

8 Discussion and Conclusions

It is clear from the tables shown earlier that the overhead tends to decrease as the number of master processors is reduced in the different simulation experiments. This is due to the fact that fewer master processors will cause more batching of messages. In other words, the tasks in the case of fewer masters are produced by fewer processors, requiring less acknowledgments when they are processed. Hence, the smaller the number of master processors, the less overhead is incurred. This might give the illusion of having one master processor would give the best results. But this is not necessarily true because if the system has a large number of processors, the single master will be swamped with messages, and can become a bottleneck. The optimal number of masters must be determined empirically, and depends largely on the application rather than the scheme itself.

As for the intra-experiment results, we noticed that as we increased the number of slaves to serve a master processor (the fan out), less overhead was incurred. We believe that when the number of slaves increases, the load will be more distributed. Thus, most slaves will be assigned work each cycle, causing their task queues to be occupied most of the time. Since no slave will acknowledge any of its masters until all messages in its task queue have been completely processed, avoiding premature emptying of the task queue will result in sending less acknowledgments, consequently reducing the overhead. This is exactly what happens when the number of slaves is increased, having the load be distributed among more processors, causing them to receive messages more frequently, and keeping their task queues nonempty as late as possible.

Experimental results show that increasing the number of masters in the system will not degrade its performance significantly. Moreover, increasing the number of slave processors allowed per master processor would enhance or at least maintain the same performance level. Therefore, we can conclude that this scheme is highly scalable, it works well independent of the fraction of processors that start as masters, and is suitable for a wide spectrum of shared-nothing system environments with anywhere from a few processors to thousands of processors.

References

[1] N. Al-Fayoumi and Eric Hanson. A solution to the parallel trigger termination problem. Technical Report TR96-009, University of Florida, CISE Dept., February 1996. http://www.cis.ufl.edu/cis/tech-reports/.

[2] David J. DeWitt, Navin Kabra, Jun Luo, Jignesh M. Patel, and Jie-Bing Yu. Client-server Paradise. In *Proceedings of the 20th VLDB Conference*, 1994.

[3] E. W. Dijkstra, W.H. Seijen, and A. J. M. V. Gasteren. Derivation of a termination detection algorithm for a distributed computation. *Information processing letters*, pages 16–5:217–219, 1983.

[4] C. L. Forgy. Rete: A fast algorithm for the many pattern/many object pattern match problem. *Artificial Intelligence*, 19:17–37, 1982.

[5] Eric N. Hanson. Gator: A generalized discrimination network for production rule matching. In *Proceedings of the IJCAI Workshop on Production Systems and Their Innovative Applications*, August 1993.

[6] Eric N. Hanson. The design and implementation of the Ariel active database rule system. *IEEE Transactions on Knowledge and Data Engineering*, 8(1):157–172, February 1996.

[7] Daniel P. Miranker. *TREAT: A New and Efficient Match Algorithm for AI Production Systems.* Pitman Publishing, 1990.

[8] M. G. Norman. Much ado about shared-nothing. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, volume 25, pages 16–22, Sept. 1996.

[9] K. Rokusawa, N. Ichiyoshi, and T. Chikayama. An efficient algorithm for distributed processing systems. In *Proceedings of the 1988 International conference on parallel processing*, pages 18–22, 1988.

[10] James G. Schmolze and Suraj Goel. A parallel asynchronous distributed production system. In *Proceedings of the Eighth National Conference on Artificial Intelligence*, July 1990.

[11] M. Stonebraker. The case for shared nothing. *IEEE Transactions on Knowledge and Data Engineering*, 9(1):4–9, 1986.

[12] Jennifer Widom and Stefano Ceri, editors. *Active Database Systems: Triggers and Rules for Advanced Database Processing.* Morgan Kaufmann, 1996.