Asynchronous Termination Detection Module User's Guide

William McLendon III
Sandia National Laboratories[*]
wcmclen@sandia.gov

1 Introduction

Interprocessor communications are performed through point-to-point messages between processors. An application written in this manner can run with a higher efficiency than an equivalent application that uses synchronizations. One added complexity that exists with an asynchronous application that does not exist with a synchronous application is the termination detection problem.

The termination problem is the problem of how exactly will a particular processor know when the global computation is finished and therefore can exit without allowing a global synchronization. For some applications, we might know how much work a particular process will perform a-priori, in which case termination can be determined using local data such as a counter. In other situations, such as one in which we do not know exactly how much work will be done a-priori, termination occurs after the following two conditions are met.

1. There are no unreceived messages. (i.e., Every message that has been sent by a process has been received by some process)
2. No process is doing any work. That is, all possible work has been exhausted.

These two conditions must be met because unreceived messages might create more work, and if a process is still working then clearly the task is not finished and we can't exit.

As shown in Baker et al. [1], the processors can be mapped into a tree structure. Tokens are passed from a node up to its parent or down to its children. An example of a set of processors mapped into a tree is shown in **Figure 1**. This allows an asynchronous communication scheme between processors and also guarantees

[*]Sandia is a multiprogram laboratory operated by Sandia Corporation, a Lockheed Martin Company, for the United States Department of Energy's National Nuclear Security Administration under Contract DE-AC04-94-AL85000.

[0]Sandia Document #: SAND 2005-3692 P.

$O(log\ P)$ time for all processors to detect when the global termination condition is met.

This document is presented as a user guide for a code library written in C with MPI that implements a scalable parallel asynchronous termination detection. The technology is based on the work presented in [1].

2 How it works

This routine allows an $O(log\ P)$ termination detection by organizing all of the processors involved in the computation into a binary tree. **Figure 1** shows how a grouping of 7 processors are organized into a tree with the three processor classifications shown. There is one *root* processor which is the only processor that will have no parent node. In this case processor 0 is the root. There are *leaf* processors which have one parent and no children; these are processors 3, 4, 5, and 6. Finally, there are *internal* processors, which have both a parent processor and one or two children processors; these are processors 1 and 2 in our little example.

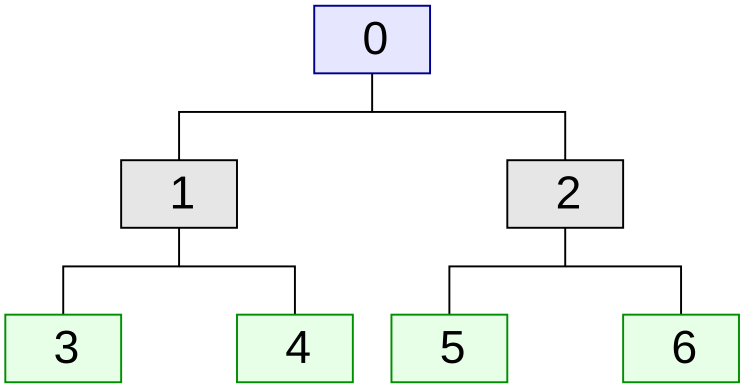

Figure 1: Example processor tree for 7 processors. Processor 0 is the root processor, processors 3, 4, 5, and 6 are the leaves, and processors 1 and 2 are the internal nodes.

A processor will perform local work until it has nothing left to do, only then will it enter the termination detection routine. This routine will determine whether or not the global termination conditions have been satisfied without using collective communications. Point-to-point messages are passed up and down a binary tree mapping of the processors involved in the computation to track certain information relevant to the status of the ongoing computation. The sum of all messages sent, received, and work performed on each subtree is passed up to the root. Once the root receives this information it determines if the global number of sends and receives are matched. If these do match, it sends a query down the tree. Once this query reaches the leaf nodes, they push their latest sum of sends, receives, and work back up. Once the root node receives the sum from both subtrees it checks to make sure that the number of sends and receives match and that no additional work was performed since the previous iteration. If everything matches we know the termination requirements are satisfied and the root processor sends a message to its children notifying them that they can terminate and then sets its state to TERMINATE so it can exit. Each intermediate processor receives this message, sets its state to TERMINATE and passes the message on to its children. When leaf nodes receive a terminate message they simply set their state to TERMINATE and exit.

3 How to Use the Asych. Termination Detection Module

The termination detection module should be fairly straightforward in its usage. The basic outline of a program that would use this termination detection routine is shown in **Algorithm 1**. The function calls are defined in later sections within this document.

```
begin
    termdetect_init(&zz, 1, 7193, MPI_COMM_WORLD);
    repeat
        while There is local work do
            do $n_w$ units of work;
            termdetect_update_workcount(&zz, $n_w$);
            send $n_s$ messages to other procs;
            termdetect_update_sendcount(&zz, $n_s$);
            recv $n_r$ messages from other procs;
            termdetect_update_recvcount(&zz, $n_r$);
        end
        termdetect_process(&zz);
    until termdetect_isterminated(&zz);
end
```

Algorithm 1: Example structure of an asynchronous application using termination detection.

In **Algorithm 1** we first initialize the termination detection data structure. Then we enter the main body of the computation where we will update the appropriate counters for *work completed*, *messages sent*, and *messages received*. After the process has exhausted its local work, the inner work–loop exits and we drop into the outer loop, which is controlled by the termination status. There, a call is made to the `process()` function where one step of the termination detection process is performed. If this operation notifies the process that the global termination conditions have been met then we can exit the routine since we know that all other processors have also completed their work.

Since the processors are mapped into a binary tree and uses point-to-point messages we expect the extra time added to the end of an asynchronous computation to be $O(log\ P)$.

```
begin
    switch node type do
        case root
            if $State = Up$ then
                iReceive messages from children;
                if recv'd f/m all children AND $work_{old} \neq work_{new}$ then
                    Send message down to children to do tree sum again;
                else if recv'd f/m all children AND $work_{old} = work_{new}$ then
                    Send TERMINATE message to children;
                    State $\leftarrow$ TERMINATE;

            end
            break;
        case leaf
            if $State = UP$ then
                Send {nSend, nRecv, nWork} to parent;
                State $\leftarrow$ DOWN;
            else if $State = DOWN$ then
                iReceive message from parent;
                if message is requesting data then
                    Send {nSend, nRecv, nWork} to parent;
                else if message is TERMINATE then
                    State $\leftarrow$ TERMINATE;

            break;
        case internal
            if $State = UP$ then
                iReceive messages from children;
                if received from all children then
                    update global nSend, nRecv, and nWork with subtrees;
                    send message to parent with global data;
                    State $\leftarrow$ DOWN;

            else if $State = Down$ then
                iReceive message from parent;
                if message is request for data then
                    Send message to children requesting new subtree data;
                    State $\leftarrow$ UP;
                else if message is TERMINATE then
                    Send TERMINATE message to children;
                    State $\leftarrow$ TERMINATE;

            break;

    end
end
```

Algorithm 2: termdetect_process(): This is the general pseudo-code for the termdetect_process() function call. This function is the "brain" of the termination detection code.

4 List of Files in the Module

The termination detection routines are stored in the files `termdetect.c` and `termdetect.h`.

5 Data Structures

The core data structure is `struct termdetect_type`. It contains all the internal data for the asynchronous termination detection routine. **Figure 5** gives a listing of the data members within `struct termdetect_type` as well as a brief description of the function of each member.

Type	**Name**	**Description**
MPI_Comm	comm	MPI communicator group
int	Active	1 if involved in termination otherwise 0.
int	nActiveProc	# of active processors in termination
int	ActiveRank	Processor rank in active list.
int	isRoot	1 if this proc is the root of the tree, 0 otherwise.
int	PID	processor id
int	nProcs	Number of processors in the tree
int	PID_Parent	Process ID of this node's parent.
int	NumChildren[2]	Processor IDs of children.
int	UpMessagesLeft	Number of up messages to wait for.
proc_type	pType	Processor node type in the tree.
state_type	State	Processor termination state.
int	localMsgSendCount	Sum of local messages sent.
int	localMsgRecvCount	Sum of local messages received.
int	localWorkCount	Sum of local work performed.
int	subTreeMsgSendCount	Sum of subtree's sent messages.
int	subTreeMsgRecvCount	Sum of subtree's recv'd messages.
int	subTreeWorkCount	Sum of subtree's work performed.
int	workCountLast	for root, work count at previous iteration
int	termTag	Message ID tag used for termination messages in MPI

Figure 2: Data members for `struct termdetect_type`

Under normal operation, the user should not need to access any of the data structure members directly; the user functions provide the main interface. We have included the data structure information here as a quick reference for debugging purposes.

6 Core Functions

The primary function calls used by termdetect are shown here. These should be the only calls needed to run the termination detection routine.

6.1 termdetect_init

```
Returns:        void
Function Name:  termdetect_init
Parameters:     struct termdetect_type *ZZ
                int active_flag
                int termTag
                MPI_Comm comm
```

Initializes the termination object and determines which processors will be actively taking part in the termination routine. A processor sets active_flag to 1 if it is participating in the termination detection scheme. The processors that are participating must all be contained within `comm`, but do not have to be exactly `comm`. The `termTag` argument specifies the MPI Tag number that will be used for the termination messages.

6.2 termdetect_update_sendcount

```
Returns:        void
Function Name:  termdetect_update_sendcount
Parameters:     struct termdetect_type *ZZ
                int _count
```

Adds `_count` to the counter for local messages sent from this process during the last cycle.

6.3 termdetect_update_recvcount

```
Returns:        void
Function Name:  termdetect_update_recvcount
Parameters:     struct termdetect_type *ZZ
                int _count
```

Adds `_count` to the counter for local messages received by this processor during the last cycle.

6.4 termdetect_update_workcount

```
Returns:        void
Function Name:  termdetect_update_workcount
Parameters:     struct termdetect_type *ZZ
                int _count
```

Adds `_count` to the counter for local work units completed by this processor during this cycle.

6.5 termdetect_process

```
Returns:        void
Function Name:  termdetect_process
Parameters      struct termdetect_type *ZZ
```

This function is the brain of the termination detection routine. Its function is to process one iteration of the termination tree and pass the appropriate tokens up or down with updated counts. Updates the values in ZZ. The basic pseudo-code of this routine is shown in **Algorithm 2**.

6.6 termdetect_isterminated

```
Returns:        1 or 0
Function Name:  termdetect_isterminated
Parameters      struct termdetect_type *ZZ
```

Calling this after termdetect_process() will return true (1) or false (0) to indicate if the code has met the appropriate conditions for termination and may safely be exited. Recall the conditions for termination are (1) There are no unreceived messages and (2) no work is being performed.

7 Extra Functions

These are some extra functions that provide some debugging and other functionality to the termdetect code.

7.1 termdetect_getstate

```
Returns:        state_type
Function Name:  termdetect_getstate
Parameters      struct termdetect_type *ZZ
```

Returns the current state of the node. A node is in one of four states: UP (0), DOWN (1), TERMINATE (2), or ERROR (3). These states indicate what that node is doing. It is either set to pass information up to its parent, down to its children, it is in termination, or an error occurred with the termination system.

7.2 termdetect_printinfo

```
Returns:        void
Function Name:  termdetect_printinfo
Parameters      struct termdetect_type *ZZ
```

Prints out the current state of a node's termination data in a more easily readable format. This is used for debugging.

7.3 termdetect_printstatus

```
Returns:        void
Function Name:  termdetect_printstatus
Parameters      struct termdetect_type *ZZ
```

Prints out some status information for the local node. This is mainly used for debugging.

8 Acknowledgments

Thanks to Bruce Hendrickson, Steve Plimpton, and Lawrence Rauchwerger for their assistance and advice to me while developing this module.

References

[1] A. H. Baker, S. Crivelli, and E. R. Jessup. An efficient parallel termination detection algorithm. Technical Report CU-CS-915-01, University of Colorado, 2001.