Abstraction and Implementation of a Lightweight Distributed Termination Protocol

D.M. Goodeve [a], C.M. Tofts [b] and S.A. Dobson [c]

[a] *Department of Computer Science, University of York, UK*

[b] *School of Computer Studies, University of Leeds, UK*

[c] *Department of Computer Science, Trinity College Dublin, Eire*

Abstract

Termination detection is a significant problem in applications whose state is distributed across a machine, thus making synchronisation on this state costly. Many parallel applications are structured such that termination occurs when some work-containing data structure becomes exhausted. This view of termination gives rise to a novel abstract model for termination detection. This paper investigates this model and its implementation using a novel scalable protocol on a distributed memory parallel system. A process algebraic model of the protocol is developed for which correctness arguments and proofs are given, using mechanical verification techniques. Results are presented to demonstrate the low resource utilisation of the protocol in practical application.

Keywords: Termination detection, termination protocols, distributed data structures, process algebra.

1 Introduction

The distributed termination detection problem has been widely studied[3,8]. The problem is to efficiently detect when the global state across a distributed computation reaches a stable termination state, providing notification of this to the application or system. This state might be explicit in that each process knows when it has finished, allowing the construction of a global conjunction predicate. More usually the problem is that all processes are co-operating in some manner and may transiently be idle before recieving more work. To detect termination therefore it must be ascertained that the entire system of processes are *simultaneously* in the idle state.

Many task-oriented parallel algorithms either terminate or move from one behaviour to another on the exhaustion of the supply of work from some pool. Detecting this

exhaustion in a distributed system is not trivial as it involves the execution state of the application, which may spontaneously generate new work, and the state of an underlying data structure. Ring-based protocols have been a popular means of detecting termination in this type of system, for example[7,11].

The motivation in this work has been the study of algorithms based on shared abstract data-types[5]. This abstraction technique allows the details of data management to be effectively removed from application code. A significant issue in the design of such types is how they can be used to support and expose useful distributed control mechanisms at their interfaces in ways meaningful to application code.

This paper has three specific contributions. Firstly, a characterisation of the termination problem in terms of operations on a task pool data structure is presented. This motivates the design of an elegant embedding technique for this and similar protocols underlying shared abstract data-type implementations. The third contribution is a novel and formally analysed scalable termination protocol.

The layout of this paper is as follows. Section 2 outlines the task pool exhaustion model for termination and compares this with conventional formulations of the termination problem. Section 3 discusses the embedding of the termination mechanism within the implementation of a shared abstract data-type. In section 4 the design and analyis of the protocol is presented. Section 5 presents a simple performance argument for the protocol and practical results obtained from a distributed application running on a network of workstations. Some concluding remarks are given in section 6.

2 Task Pool Exhaustion

An application employing task parallelism typically uses some pool data structure that contains all tasks to be processed. This pool may be centralised or implicitly or explicitly distributed. Processes typically both remove tasks for the pool and add new ones as execution proceeds. For example, a node of some tree may be removed from the pool and expanded to produce its children which are then re-inserted so that another process may access them. Termination should be signalled when the pool becomes exhausted so that either the application may exit, or this mode of process behaviour may be changed.

In a system where there is only one process, detecting that the pool has become empty is straightforward. When there is nothing in the pool, any further *get* operation, attempting to remove a task, should return an exceptional value indicating that the operation cannot be completed. If there is more than one process operating, this semantics is no longer adequate. If the pool is empty, it may just be a transient con-

dition before another process inserts some tasks. Processes cannot therefore rely on an exceptional return value to signal termination. These two situations are depicted in figure 1.

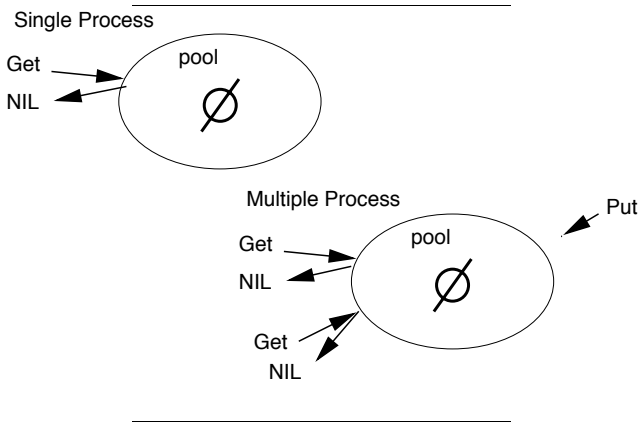

Fig. 1. Single and multiple processes accessing a task pool data structure.

As the return of an exceptional value in the multiple process case merely indicates a transient state, an alternative semantics is that all *get* operations return valid data. If no valid data is available therefore, a *get* operation cannot complete, forcing the calling process to block. The system can be externally observed to have terminated when *all* processes are in the blocked state.

2.1 Solutions

The simplest solution is to argue that termination detection is a system issue, not an algorithmic one. Rather than internal to the algorithm, the supporting system may be left responsible for detecting when all processes have reached the blocked state. After a period of inactivity, detected through some consensus mechanism such as the probing mechanism of [3], it may be concluded that the algorithm has terminated.

This scenario is typical of Linda systems. To enable Linda programs to detect this type of termination internally, new primitives have been proposed, notably *collect* which allows the state of a tuple space (cf. pool) to be examined directly[1].

An alternative is to attempt to solve the consensus problem directly using a strongly coherent counter[6]. By appropriate manipulation, a counter may keep track of the number of tasks that are *potentially* in the pool, analogously to the counting protocol described in [8]. In a possible protocol, the last remaining unblocked process examines the counter before committing to a *get* operation, allowing it to detect the condition that this pool is and will remain empty. It can therefore signal the

other processes *via the pool* that termination has occurred[4]. Unfortunately this solution leads to significant performance problems due to the costs of maintaining a coherent counter across a loosely-coupled distributed system[5].

2.2 A Hybrid Semantics

The two options explored for the semantics of the pool are:

– Non-blocking. In this case, processes will not be blocked by an empty pool, but cannot conclude anything about the state of the system from an exceptional return value.
– Blocking. In this case, the termination state exists when all processes are blocked waiting for an item to be returned from the pool.

An alternative semantics for the *get* operation offers an elegant integration of the termination mechanism with the pool shared type. The mechanism is depicted in figure 2. The semantics of the *get* operation are identical to the blocking case up

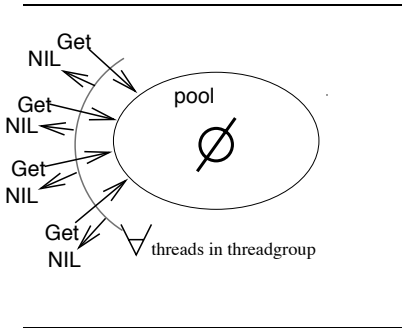

Fig. 2. Multiple processes accessing an empty pool data structure, with consensus termination detection.

to the point that the pool data structure becomes empty. The pool structure *records* which processes have access to it. In an object-oriented environment this can be captured by the notion of threads *binding* on to the object encapsulating the pool data structure. When all bound threads are all simultaneously attempting to remove a task from the pool using *get*, and all are unable to proceed we have the termination state. In this state it is arranged so that an exceptional value is simultaneously returned to all *get* operations, signalling termination to all threads. This allows the threads to behave in an application dependent manner in response to the exhaustion of a supply of work. The termination property as described above can also be re-used a number of times during the execution of an application; there is no requirement implied that the application need terminate.

This typical set of assumptions modelling a distributed system follows that of [12].

(i) Each processing node has two states; idle and active.

(ii) An active processing node can become idle spontaneously.

(iii) Only active processing nodes can send messages.

(iv) On receipt of a message, an idle processing node becomes active.

And optionally:

(v) the sending and receipt of a message occur in the same atomic action.

The conjunction of assumptions (i)-(v) assures that the state where all processing nodes are idle is stable, which is the termination state.

The model presented in section 2.2 can be understood in these terms as the pool distributed shared abstract data-type is itself implemented in terms of a messaging system. Delivery of data to an idle node can move that node to the active state by providing more work. An active node can exhaust its supply of work and therefore become idle. There is thus a direct correspondence between the abstracted model and this underlying messaging model.

3 Embedding mechanism

To allow the construction of operations with collective operational semantics such as termination detection, the concept of a threadgroup is introduced. A threadgroup is a logical collection of threads of control that may for some purposes be treated as a unit. This construction allows predicates over the execution state of parts of a distributed application to be evaluated. A group may all run in the same address space, or in the case of the implementation described here, may run across many address spaces in a distributed application.

Once a threadgroup is established, mechanisms may be employed to allow the combined execution status to be examined. Once such mechanism is that used to detect termination as suggested in section 2. It is suggested that other predicates over group execution state may be useful, motivating the design of a generic mechanism.

A prototype threadgroup mechanism specifically to facilitate termination detection

in applications distributed across networks of workstations has been implemented using the DEC-SRC Modula-3[10] *Threads* module. Inter-thread synchronisation is facilitated using signal variables in this module. The implementation modifies the calls on signal variables such that if a thread belonging to a threadgroup waits on a signal variable, it first examines whether all other threads are similarly waiting. If so, a run-time signal is generated that may be used to trigger other activity, facilitating the protocol mechanism. Within the run-time layer, signal variables can also be accessed to release all threads currently waiting, returning to each blocked thread an *exceptional* value indicating a *group* release event.

This termination detection mechanism is similar to a barrier. However, joining a synchronisation barrier is a committed event, whereas this mechanism permits threads to join and leave the synchronisation point, driven by other normal events on signal variables. The similarity with the barrier is that once all threads in the group arrive, they are simultaneously released. The distinction is that there are two circumstances that lead to a thread leaving the synchronisation point; normal release or group synchronised release. This implementation informs threads of which has occurred, allowing operations that can use this information to be constructed.

4 Termination Protocol

It is known to be difficult to verify the correctness of termination protocols; mistakes have been made in the literature (see [14] for a detected example). This section presents the design and analysis of a lightweight protocol that efficiently detects termination. A process algebraic approach is used to both specify and verify the protocol. Algebraic reasoning is supplemented with the use of the Edinburgh Concurrency Workbench automated analysis tool[2].

The protocol presented underlies our implementation of the threadgroup abstraction for termination detection. The realisation of this protocol using local synchronising operations and RPC for remote synchronising operations is straightforward. The protocol has been used to implement the run-time detection mechanism on both a loosely-coupled distributed system, and on a massively parallel machine[5].

The protocol describes both the messaging structure within a shared abstract datatype facilitating dynamic load balancing, and the underlying run-time mechanism for detecting termination. There is subtle interaction between these two mechanisms, requiring that both of them are exposed for analysis.

Despite the apparent simplicity of the protocol, its correct derivation has proved to be difficult. This has motivated the detailed definition of the protocol in this paper and the presentation of the validation methodology applied.

The process algebra CCS[9] is used to describe both the threads acting on the data structure and the internal components comprising the data structure and implementing the termination protocol. It is assumed in the following that the reader is reasonably familiar with this notation. The correctness arguments presented below are given both in English and in a form suitable for automated validation. The protocol has been validated using the Edinburgh Concurrency Workbench[2] using propositions expressed in the Hennessy Milner modal logic[13].

4.1 Principle of operation

A Shared Abstract Data-type (SADT), providing an abstract interface to a distributed data structure, is accessed by a number of threads spread across the address spaces of a distributed machine. A *store* SADT supports *put* and *get* operations, allowing threads to deposit and obtain items of work. This can model a wide variety of data structures. The only assumption about the semantics of operations on such a structure in the following is that it is possible for get operations to block due to lack of data.

The SADT is implemented by a set of *representatives*, one per address space in a system[5]. Each of these representatives tracks the number of threads accessing it and their state. When a representative becomes empty, and all its client processes are stalled during a *get* operation, it sends a signal to a central controller stating that it is stalled. After this it can either tell the controller that it has become active again as new data has arrived from elsewhere, or it is informed by the controller that all other representatives have signalled that they have also become stalled, so signalling that termination has been detected. A model of the system corresponding with the CCS formulation is shown in figure 3. In the CCS formulation below it is assumed that the number of threads in the group is set during some initialisation phase before normal operation commences.

4.2 Local system

The protocol is analysed firstly at a local level before extending it to a distributed implementation spanning multiple address spaces. The local model consists of a number of $Thread$ agents and an SADT representative comprising a data structure agent and a control agent.

4.2.1 Thread agent

The *Thread* agent models the behaviour of client threads of the SADT. Conventionally, outputs are denoted through the over-barred co-action of an action name

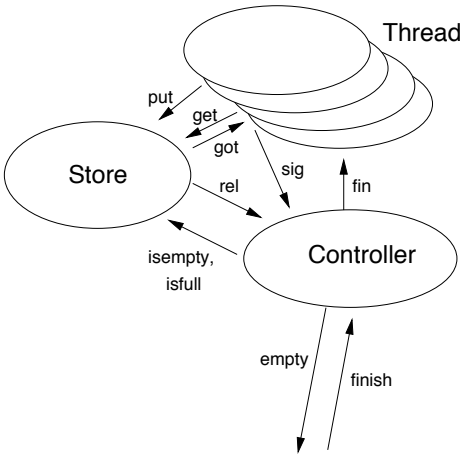

Fig. 3. System of agents comprising the distributed termination protocol mechanism.

pair.

$$Thread \stackrel{def}{=} \overline{sig}.(\overline{get}.got.Thread + fin.0) + \overline{put}.Thread \quad (1)$$

The $Thread$ agent repeatedly accesses the data structure, either putting or getting an element at each step. Putting an element is accomplished by the simple action $\overline{put}$. Getting an element involves a compound synchronisation. Firstly the thread signals ($\overline{sig}$) that it is attempting a get operation, equivalent to issuing a wait on a signal variable. Either the thread will then be allowed to complete the sequence $\overline{get}.got$ to obtain an item, or will synchronise of a fin action, terminating its behaviour.

The design of the synchronisation mechanism should guarantee that all threads are only allowed to terminate via their fin action when the global data structure is empty and all threads are similarly blocked, unable to synchronise via the $\overline{get}$ action.

4.2.2 Representative agents

The local SADT representative consists of two agents; $Store_E$ and C_0. The Store agent simulates the underlying data structure.

$$\begin{aligned} Store_E &\stackrel{def}{=} isempty.Store_E + put.Store_F \\ Store_F &\stackrel{def}{=} isfull.Store_F + put.Store_F + \\ &\quad get.\overline{rel}.\overline{got}.Store_F + get.\overline{rel}.\overline{got}.Store_E \end{aligned} \quad (2)$$

A complete model of the data structure would have an unbounded number of states, representing the number of stored elements. This representation only distinguishes two major states; full and empty. In the empty state $Store_E$ nothing can be removed from the store. In the full state, removing an item through the sequence $get.\overline{rel}.\overline{got}$ either does not empty the structure, hence it remains in state $Store_F$ on completion, or it becomes empty. In both states, actions are available to test the state externally.

The control agent, beginning as C_0, implements a counter tracking the number of threads currently attempting to get an element from the store agent. The number of threads is expressed by the parameter T.

$$\begin{aligned} C_0 &\stackrel{def}{=} sig.C_1 \\ C_i &\stackrel{def}{=} sig.C_{i+1} + rel.C_{i-1} \\ C_T &\stackrel{def}{=} \overline{isempty}.\overline{empty}.C_W + rel.C_{T-1} \\ C_W &\stackrel{def}{=} finish.(\overline{fin})^T.\overline{ok}.0 \end{aligned} \tag{3}$$

If all threads are blocked (C_T) and it is possible to observe that the store is empty ($\overline{isempty}$), then the controller indicates that it is preparing to finish via the $\overline{empty}$ action. The $finish$ action is provided as a means for an external agent to give the controller *permission* to terminate. Subsequently the control agent offers $\overline{fin}$ actions to synchronise with and terminate each of the threads. Finally it offers an $\overline{ok}$, signalling that the termination sequence has occurred, action before becoming the null agent.

4.2.3 Correctness of local system

A system consisting of the $Thread$, $Store_E$ and C_0 agents illustrates the basic principle of operation of the protocol. Before extending this from a single address space model to a distributed model, this simple local system is analysed for correctness.

A local system may be described in its initial state by the $Local_E$ agent below:

$$\begin{aligned} Local_E &\stackrel{def}{=} ((\prod_T Thread) | Store_E | C_0) \backslash R \\ &\quad \text{where} \quad R = \{sig, rel, get, got, put, fin, isempty\} \end{aligned} \tag{4}$$

For a termination to be considered correct the following conditions must apply:
(i) All threads must be waiting to get an item from the data structure.

(ii) The data structure must be empty.

Condition (i) implies that the thread agents are all in the waiting state:

$$\overline{get}.got.Thread + fin.0$$

Condition (ii) implies that the data structure is in the state $Store_E$, where it cannot synchronise on a get action. Taken together therefore, this implies that the only actions on which the thread agent can synchronise is fin.

For the thread agents to have entered the waiting state as implied by (i), the control agent must have entered state C_T. As no thread can offer a $\overline{rel}$ action, progress is only possible through synchronising with the store agent on $\overline{isempty}$. This confirms that the store is empty, therefore that the $\overline{get}$ actions of the threads cannot progress, implying that the rel synchronisation cannot occur.

Once the termination conditions arise therefore, the controller agent will enter state C_W after synchronising with the external system on an $\overline{empty}$ action. Once the external system offers a $\overline{finish}$ action, the controller can only progress by offering the sequence $(\overline{fin})^T.\overline{ok}$. From (i), the $\overline{fin}$ actions can thus synchronise with the thread agents, leading them to terminate (null agent). Once this has been accomplished the control agent offers the $\overline{ok}$ action and itself becomes the null agent.

To verify that the system does indeed follow these behaviours, the Edinburgh Concurrency Workbench has been employed. The first test of correctness is that the system is *only* able to terminate correctly. The termination state defined above allows no further actions. In a correct system, there should be no other states in which no further actions are possible. Enumerating the deadlocks of the agent $Local_E$ returns only one state, the correct termination state.

From the argument given above it follows that a correct termination consists of the following sequence of external observations: $\overline{empty}.finish.\overline{ok}$. In particular, it is not possible to observe the action $\overline{ok}$ unless the threads have correctly terminated. To check correctness, two temporal propositions expressed through the Hennessy Milner Logic are used:

$$\begin{aligned} P1 &= \Box(\Diamond(< \overline{ok} > T \wedge \neg < isfull > T) \ \mid \ [-]F) \\ P2 &= \Box(\neg < -\overline{ok} > [-]F) \\ Local_E &\models (P1 \wedge P2) \end{aligned} \tag{5}$$

Where $\Box$ is the *always* temporal operator; its argument must be true for all derived states reachable from this state, and $\Diamond$ is the *eventually* operator, that in at least one derived state, the argument is true.

Proposition *P1* expresses the condition that from all states (except deadlock) it is possible to terminate correctly, observing an $\overline{ok}$ action and *without* the possibility of observing the data structure to be in the full state via the $isfull$ action. Proposition $P2$ expresses that from all states, it is not possible to deadlock (terminate) *except* via the $\overline{ok}$ action. The conjunction of these two therefore implies that correct termination is always possible (liveness) and that the only deadlock is the correct termination sequence (correctness).

The $\Box$ and $\Diamond$ operators are straightforward to express using the minimum and maximum fix-point propositions supported by the Concurrency Workbench. Checking that the agent $Local_E$ satisfies both $P1$ and $P2$ is accomplished using the model checker of the Concurrency Workbench. A limitation of a proof made in this way is its limitation on the size of the system which must be boundedly finite to complete an analysis. Towards a general proof, the following inductive argument is offered.

If a system with T threads correctly terminates; consider a system with $(T + 1)$ threads. The additional thread can either be in the active state, or waiting to synchronise on one of $\overline{get}$, got or fin. The controller is also expanded by one state to accept the additional $\overline{sig}$ and $\overline{rel}$ actions due to the additional thread.

If the thread is active, then the controller will be in the same state as it would in a system without this thread present. If the thread is waiting, having synchronised on sig, then the controller will be elevated by one state from where it would be in the equivalent T thread system. This expansion of the controller states will therefore still permit correct behaviour. If the added thread is active, then the controller will not be able to reach the C_W state. If the added thread is waiting on $\overline{get}$ or fin, then the controller will only reach the C_W state if all other threads are similarly waiting.

Informally therefore, the addition of an extra thread will not affect the correctness of the local system. Correctness has been mechanically verified for systems with up to 10 threads, giving confidence in the existence of a general proof.

4.3 Global model

To extend the model to a distributed system it is necessary to include definitions for a global controller and a mechanism for migrating elements between representatives at each node in the system. This is achieved through the following additions to the local model, shown pictorially in figure 4.

4.3.1 Work migration mechanism

To facilitate migration of data elements between nodes, a sender-initiated migration protocol is modelled. Arrivals of new elements from elsewhere occur via the ***Rput*** agent. These elements are generated by the ***Migrator*** agent at a remote node. The ***Migrator*** agent is simply a client of the store, similar to the ***Thread*** agent, and will similarly block if the store is empty. The thread count T reflects the addition of this extra client into the local agent. The sending of an item of data across the network is accomplished via the sequence $\overline{rputo}.racki$, modelling a remote procedure call.

The ***Rput*** agent accepts an $rputi$ action from a remote ***Migrator*** agent, subsequently putting the received item into the local store. It is possible that this arrival will

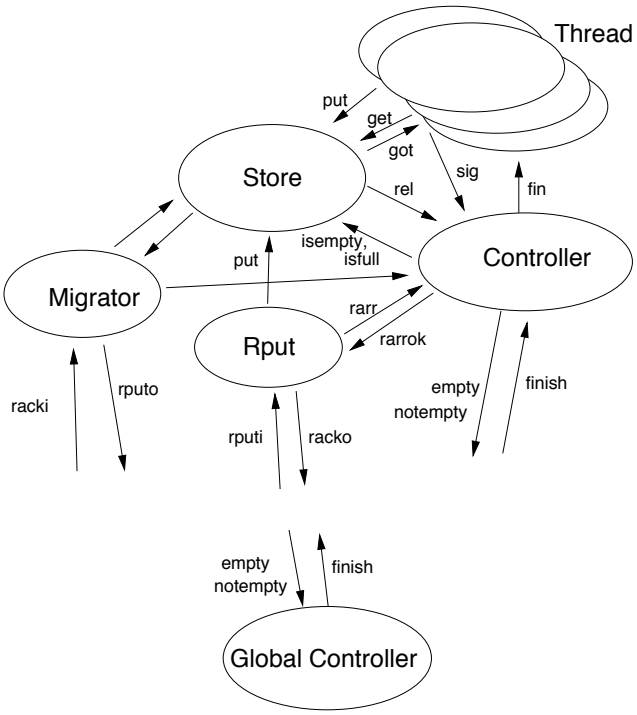

Fig. 4. Global system of agents, extending the local system to facilitate termination detection across multiple address spaces.

enable this node to perform work again after it has previously generated an $\overline{empty}$ action. This requires synchronisation with the control agent, accomplished by the sequence $\overline{rarr}.rarrok$.

$$\begin{aligned} Migrator &\stackrel{def}{=} \overline{sig}.(\overline{get}.got.\overline{rputo}.\overline{racki}.Migrator + fin.0) \\ Rput &\stackrel{def}{=} rputi.\overline{put}.\overline{rarr}.rarrok.\overline{racko}.Rput \end{aligned} \tag{6}$$

If all client threads have become idle waiting on an empty store, then the control agent will have performed $\overline{empty}$ action, notifying some global controller. If new work arrives from another node in the system, this notification must be revoked. This occurs through the $\overline{notempty}$ action, triggered when the $Rput$ agent inserts a new item into the store.

$$\begin{aligned} C_0 &\stackrel{def}{=} sig.C_1 + rarr.\overline{rarrok}.C_0 \\ C_i &\stackrel{def}{=} sig.C_{i+1} + rel.C_{i-1} + rarr.\overline{rarrok}.C_i \\ C_T &\stackrel{def}{=} \overline{isempty}.\overline{empty}.CW + rel.C_{T-1} \\ CW &\stackrel{def}{=} finish.(\overline{fin})^T.\overline{ok}.0 + rarr.\overline{notempty}.\overline{rarrok}.C_T \end{aligned} \tag{7}$$

A race exists where a remote node migrates data to this node rendering itself empty. If that node were to generate the action $\overline{empty}$ *before* this node generates the action $\overline{notempty}$, revoking its previous action, then a global controller might erroneously detect that all nodes are idle.

The remote inserter agent $Rput$, through the $\overline{rarr}.rarrok$ sequence, ensures that the local controller agent performs the $\overline{notempty}$ action, if required, before acknowledging the arrival of the new data to the sending *Migrator* agent. This sending *Migrator* agent can therefore only proceed to its blocking state, allowing $\overline{empty}$ to be performed by its local controller once the receiving node has performed $\overline{notempty}$. This ensures that global termination can only be detected when *all* nodes have achieved the idle state. It may be observed that the system is stable after the time that all nodes have signalled $\overline{empty}$.

4.3.2 Local correctness

The new local node agent is defined as the parallel composition:

$$Local'_T = ((\prod_{T-1} Thread)|Migrator|RPut|StoreE|C_0)\backslash R \tag{8}$$

To assure that the properties of the local node agent are still preserved, *Local'* is checked against the propositions *G1*, *G2* and *G3*.

$$\begin{aligned}
G1 &= \Box(\neg < -\overline{ok} > [-]F) \\
&\quad Local' \not\models G1 \quad Local'\backslash\{rputi\} \models G1 \\
G2 &= \Box(\neg < \overline{empty} > \rightarrow \Box(\neg < isfull > T)) \\
&\quad Local' \not\models G2 \quad Local'\backslash\{rputi\} \models G2 \\
G3 &= \Box(\neg < \overline{empty} > T \rightarrow \neg\Diamond(< \overline{rputo} > T)) \\
&\quad Local' \not\models G3 \quad Local'\backslash\{rputi\} \models G3
\end{aligned} \tag{9}$$

In the absence of remote insertion ($rputi$ action) all the propositions hold. *G1* guarantees that deadlock (termination) is only possible via the $\overline{ok}$ action, thus no additional deadlocks have been introduced. Proposition *G2* guarantees that once the node agent has signalled $\overline{empty}$, then it cannot be observed to become full. Proposition *G3* guarantees that once the node agent has signalled $\overline{empty}$, then it cannot subsequently generate a remote insertion ($\overline{rputo}$). For a node in isolation, not receiving any remote insertions, these propositions taken together guarantee that the node agent will behave correctly; only terminating via the $\overline{ok}$ action and once having signalled that it is empty, remaining empty.

That these propositions are not true for the agent when remote insertions are enabled means that a collection of nodes as a whole must co-ordinate to support the necessary guarantees. In a system of local agents which does permit inter-node

work migration, if the system is operating correctly, the only achievable deadlock must be preceded by an $\overline{empty}$ action. This is asserted by *G4* in an analogous manner to *G1*. The *Local'* agent is also re-labelled to enable multiple agents to intercommunicate. Note that the semantics of re-labelling do not permit *Local'* agents to synchronise with themselves on the msg and ack actions.

$$\begin{aligned} RLocal'_T &\stackrel{def}{=} Local'_T[\overline{msg}/\overline{rputo}, msg/rputi, \overline{ack}/\overline{racko}, ack/racki] \\ A &= \{msg, ack, finish\} \\ Nodes_{(N,T)} &\stackrel{def}{=} (\prod_N RLocal')\backslash A \\ G4 &= \Box(\neg < -\overline{empty} > [-]F) \\ Nodes_{(2,T)}\backslash A &\models G4 \end{aligned} \tag{10}$$

The satisfaction guarantees this property for a system of up to 2 nodes closed under the $finish$, msg and ack actions, demonstrating that no other deadlocks exist apart from that induced by the non-availability of the finish synchronisation with an external controller. This satisfaction has been mechanically checked for $T = 0, 1, 2, 3$. Extending this satisfaction to larger N is discussed in section 4.4.

4.3.3 Global controller

It has been demonstrated that the new local system under the restriction of no inter address-space communication behaves correctly. Once message passing is allowed, then a distributed system of local agents behaves correctly up to the point of delivering the $\overline{empty}$ notifications.

An global termination agent must therefore guarantee that $finish$ synchronisations are only possible once all local nodes have reached an impasse via their $\overline{empty}$ actions. A global controller that facilitates this is simply a counter, analogous to the local controller.

$$\begin{aligned} GC_0 &\stackrel{def}{=} empty.GC_1 \\ GC_i &\stackrel{def}{=} empty.GC_{i+1} + notempty.GC_{i-1} \\ GC_N &\stackrel{def}{=} \overline{gok}.(\overline{finish})^N.0; \end{aligned} \tag{11}$$

The complete system can therefore be expressed as the agent:

$$System_{(N,T)} \stackrel{def}{=} (Nodes_{(N,T)} | GC_0)\backslash\{empty, notempty, finish\} \tag{12}$$

4.3.4 Global correctness

To test correctness for the complete system, three propositions *G5-G7* are used. Proposition *G5* guarantees that the system can always terminate correctly from any state except deadlock. Proposition *G6* guarantees that the only deadlocks in the system proceed via $\overline{ok}$ as the final action (a node agent terminating). Finally *G7* guarantees that once the $\overline{gok}$ global termination signal has been seen, then it is not possible to observe any node to have a store which is not empty.

$$\begin{aligned} G5 &= \Box(\Diamond(< \overline{ok} > [-]F) \mid [-]F) \\ G6 &= \Box(\neg < -\overline{ok} > [-]F) \\ G7 &= \Box(< \overline{gok} > T \rightarrow \Box(\neg < isfull > T)) \end{aligned} \tag{13}$$

The final property of the system required to prove correctness is that all terminations (deadlocks) can only occur following the $\overline{gok}$ action. This can be concluded by inspection of the agents concerned. The local control agent can only signal local termination via $\overline{ok}$ by completing the requisite set of $\overline{fin}$ actions with its client $Thread$ and $Migrator$ agents. This in turn can only be triggered via a $finish$ action, which can only arise subsequent to a $\overline{gok}$ action. Therefore all terminations can only proceed via a $\overline{gok}$ action, implying correct global termination.

From these satisfactions it is concluded that the system can only terminate correctly, closing down all operations on all local nodes once all local stores are empty and all client threads are blocking on get operations from those stores.

It has been mechanically checked that these satisfactions apply to various instances of $System_{(N,T)}$:

$$System_{(N,T)} \models (G5 \wedge G6 \wedge G7) \tag{14}$$

This has been mechanically checked for $N = 1, 2$ and $T = 1, 2, 3$.

4.4 Extensions

The preceding reasoning and model checking has assured that the protocol described is correct for systems containing up to two local nodes each supporting three client threads.

A problem with the system construction as presented above concerns the non-uniqueness of the msg and ack synchronisations in the system. Agents $Nodes_{(3,N)}$ and above do not satisfy *G4* which in turn leads to the failure for systems including a global controller to satisfy *G5-7*. This is found to be due to the non-unique pairing

of msg and ack communications between nodes, leading to the possibility that the wrong ack will acknowledge a communication. This can lead to a deadlock arising where one $RLocal'$ is left waiting for an acknowledgement that has been consumed by one of the other $RLocal'$ agents.

To avoid this, and thus extend the satisfiability of *G4-7*, a guarantee of unique pairing of messages and acknowledgements is required. This can be achieved by introducing two unique names for each communication from an $RLocal'$ agent, re-labelling these to uniquely address one other $RLocal'$ instance in the system. The *Migrator* and *Rput* agents are thus re-defined as follows:

$$\begin{aligned} Migrator &\stackrel{def}{=} \overline{sig}.(\overline{get}.got.Mig' + fin.0) \\ Mig' &\stackrel{def}{=} \sum_{i=0}^{N-2} \overline{rputo_i}.racki_i.Migrator \\ Rput &\stackrel{def}{=} \sum_{i=0}^{N-2} rputi_i.\overline{put}.\overline{rarr}.rarrok.\overline{racko_i}.Rput \end{aligned} \tag{15}$$

The construction of $Nodes_N$ now requires the re-labelling of the $rputo_i$, $rputi_i$, $racko_i$ and $racki_i$ actions to form unique communications paths across the set of nodes, for example the zeroth node agent in a three node system ($N = 3$) could be assigned the relabelling:

$$\begin{aligned} RLocal_0 \stackrel{def}{=}\ & RLocal'[\overline{m01}/\overline{rputo_0}, m10/rputi_0, \overline{a01}/\overline{racko_0}, a10/racki_0, \\ & \overline{m02}/\overline{rputo_1}, m20/rputi_1, \overline{a02}/\overline{racko_1}, a20/racki_1] \end{aligned} \tag{16}$$

Constructing the system as before with this relabelling applied and restricting over the extended resulting action set allows systems with the unique name pairing property to be specified and verified. Systems so formulated have been mechanically verified correct under $G5-7$ up to the complexity of $T, N = 3, 3$.

4.4.1 Re-entrant behaviour

In the protocol as described, all thread and control agents deadlock on correct termination. Instead of this behaviour, all these agents can continue operating from their initialisation state, allowing the termination mechanism to operate again. Rather than being just purely for application termination therefore, this mechanism can be used to manage the phases of a distributed computation.

To express this change in the process algebraic model is straightforward, however it makes the system cyclic and thus expressing the correctness of termination is more involved. It is reasonable to argue however that the extension of processes behaviour beyond the deadlock point should not change the form of the system state space as we will just be returning to a starting state again, thus the correctness

arguments and checks remain valid. The implementations of the protocol assume the correctness of this re-entrant formulation.

4.5 Summary

A correctness argument and automatically verifiable propositions over the behaviour of the termination protocol have been presented. On the basis of these, the correctness of the distributed termination protocol has been demonstrated for restricted system sizes. The complexity of the system has precluded the discovery of a simple analytical proof of correctness for arbitrary system size. The experience gained with automated model checking does however inspire confidence that such a general proof does exist.

5 Performance

It has been claimed that the proposed protocol is lightweight. By this it is implied that the protocol uses little in the way of system resources to complete its task. In this section this issue is briefly studied, presenting a paper analysis and practical results from the use of the protocol in a network of workstations based distributed application.

5.1 Message counting

A node will generate a termination message on becoming idle, or on after being idle, becoming active again. Typically this might happen several times for each node in a network before the stable system-wide termination state exists. The number of messages generated by the protocol is therefore $O(N)$ where N is the size of the network. On a scalable network therefore, the protocol should scale ideally. A limitation to the scalability of the protocol will be the bandwidth sustainable by the node responsible for maintaining the global control thread. It is assumed that this will not be a limitation on relatively small systems. For larger numbers of process this could lead to a hierarchical design for the global controller, alleviating this bottleneck.

The constant factor in the time complexity of the protocol is due to the number of times a node enters into and exits from the idle state. In the distributed data structure application investigated below, this is a function of the quality of the load balancing in the system. Specifically, if a node becomes regularly starved of work, then the constant factor of the number of termination messages will increase.

A set of executions of a distributed application were performed with the code instrumented to measure the network traffic generated. The distributed system used was a set of Silicon Graphics Indy workstations connected via a switched 10BaseT Ethernet using TCP/IP protocols to support message passing. The distributed termination protocol was embedded in the implementation of a weakly-coherent distributed priority queue, in which transparent dynamic load balancing was performed. For this shared abstract data-type, a record was made of the number of messages generated in total by all the representatives, and the messages specifically concerning termination; *empty* and *notempty*.

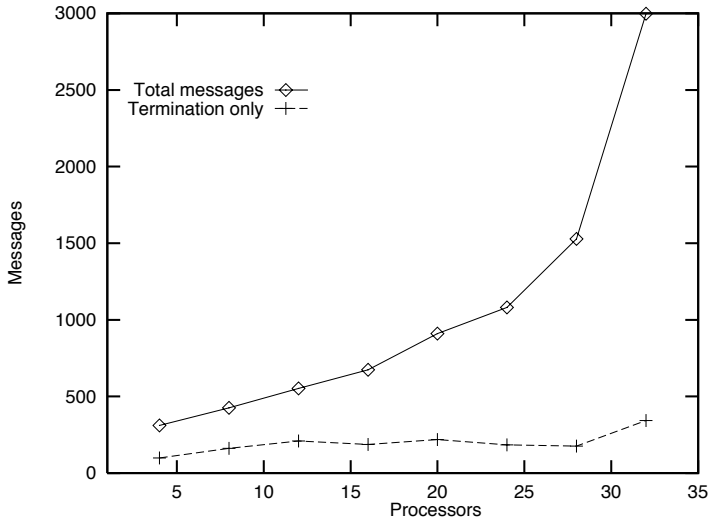

Fig. 5. Termination message count against total message count within a distributed implementation of a weakly coherent priority queue. Figures obtained from usage inside a branch-and-bound parallel solver averaged over several executions of a set of benchmark problems.

Figure 5 shows the number of messages generated by the SADT during execution. The data shown is averaged over several executions on different sizes of network solving a set of benchmark problems. The average amount of work done by each network during each execution is roughly constant.

The total number of messages generated within the SADT steadily climbs with the size of system, showing a marked increase above 25 processors. This is increase is due to the interaction of the periodic dynamic load balancing scheme and the available network performance across the larger distributed machine. The number of these messages attributable to the termination mechanism appears below this. Not only is the number of messages small, but as the machine size increases the proportion of message traffic concerned with termination decreases progressively.

In conjunction with the dynamic load balancing scheme used therefore, the termination mechanism scales well.

An additional factor to be considered is the size of messages. Termination messages are very short status messages. The load balancing messages typically consist of the order of 1KByte of data. Taking this into account, the claim is justified that this termination mechanism is indeed lightweight in practical application.

6 Concluding remarks

The protocol mechanism and its abstraction presented in this paper have been proved effective in practice. The protocol used, whilst simple to formulate did cause implementation difficulties which were hard to fully understand without the aid of automated checking. The discovery of general proofs for this type of protocol system, based on induction as a constructive principle is viewed as a difficult problem.

Our automated property checking approach inspires confidence in the correctness of this protocol, but it is limited by the size of system that can be analysed using current generation property checking systems. It is intended that approaches to the generalisation of this kind of proof are pursued, with the intention of developing proofs that can scale with the implementation of a system.

Acknowledgements

The authors would like to acknowledge the inputs of their colleagues into the development of this work; notably Graham Birtwhistle and Chris Wadsworth. Also thanks to Perdita Stevens for offering a pre-release version of CWB7.1, allowing the use of a Silicon Graphics Origin2000 for model checking.

References

[1] Paul Butcher, Alan Wood, and Martin Atkins. Global synchronisation in Linda. *Concurrency: Practice and Experience*, 6(6):505–516, 1994.

[2] R. Cleaveland, J. Parrow, and B. Steffen. The Concurrency Workbench: A Semantics-based Verification Tool for Finite-State Systems. Technical Report ECS-LFCS-89-83, Laboratory for the Foundations of Computer Science (LFCS), University of Edinburgh., 1989.

[3] Edsger W. Dijkstra, W.H.J. Feijen, and A.J.M van Gasteren. Derivation of a Termination Detection Algorithm for Distributed Computations. *Information*

Processing Letters, 16:217–219, 1983.

[4] D. Goodeve, R. Briggs, and J. Davy. Capturing Branch and Bound using Shared Abstract Data-types. In C.R. Jesshope and A.V. Shafarenko, editors, *UK Parallel '96 (BCS/PPSG)*, pages 119–134. Springer Verlag, July 1996.

[5] D.M. Goodeve, S.A. Dobson, J.M. Nash, J.R. Davy, P.M. Dew, M. Kara, and C.P. Wadsworth. Toward a Model for Shared Data Abstraction with Performance. *Journal of Parallel and Distributed Computing*, 49(1):156–167, February 1998.

[6] Maurice Herlihy. Wait-Free Synchronisation. *ACM Transactions on Programming Languages and Systems*, 11(1):124–149, January 1991.

[7] Nobert Kuck, Martin Middendorf, and Hartmut Schmeck. Generic Branch and Bound on Transputers. In R. Grebe et al., editor, *Transputer Applications and Systems '93*, pages 521–535. IOS Press, 1993.

[8] Friedemann Mattern. Algorithms for distributed termination detection. *Distributed Computing*, 2(3):161–175, 1987.

[9] Robin Milner. *Communications and Concurrency*. Prentice Hall, 1989.

[10] Greg Nelson. *Systems Programming with Modula-3*. Prentice-Hall series in Innovative Computer Science. Prentice-Hall, 1991.

[11] V.J. Rayward-Smith, S.A. Rush, and G.P. McKeown. Efficiency considerations in the implementation of parallel branch-and- bound. *Annals of Operations Research*, 43:123–145, 1993.

[12] Steffan Rönn and Heikki Saikkonen. Distributed termination detection with counters. *Information Processing Letters*, 34(5):223–227, May 1990.

[13] C. Stirling. An Introduction to Modal and Termporal Logics for CCS. In *Proceedings of the 1989 Joint UK/Japan workshop on Concurrency*, volume 491 of *LNCS*, pages 2–20. Springer-Verlag, 1991.

[14] R.B. Tan, G. Tel, and J. Van Leeuwen. Comments on: Distributed Termination Detection Algorithm for Distributed Computations. *Information Processing Letters*, 23(3):163, October 1986.