Performance of Fault Tolerant Networks of Workstations

John Morris
Centre for Intelligent Information Processing Systems,
The University of Western Australia,
Nedlands WA 6907, Australia
morris@ee.uwa.edu.au

Abstract

Functional or dataflow models of computation enable a program’s run-time system to determine which portions of it must be repeated when faults occur. Modifications were made to the run-time system of Cilk - a threaded C - to enable a Network of Workstations to tolerate fail-stop faults of individual processors or the network. The overheads needed to provide this fault tolerance are shown to be mainly CPU cycles and memory, with little additional network load being generated in the absence of faults. This makes it feasible to run long computations successfully on NoWs where ownership, control and distribution of the individual processors may be widely distributed.

1. Introduction

There are two key aspects to achieving fault tolerance in processor systems. Replication tolerates the loss of a component or the corruption of information, e.g. in error correcting memory, additional bits guard against the loss of a memory chip or cross-talk on signal lines. Additional processors may also be used to replicate part or all of a computation. However replication is expensive and may be unnecessary when failure rates are low. In such cases, sufficient information about the state of a computation needs to be kept so that it can be restarted. Traditionally, one uses check-pointing - saving the entire state of a computation at intervals and re-starting from the last check-point. The key problem is determining the state that needs to be saved: the simplest approach saves the entire memory contents - at considerable cost if large blocks of data must be transported over a network. Conventional programming languages give scant aid to a programmer who needs to save only the needed state: profligate use of global variables makes determining the required state a laborious and error-prone activity. Although automatic systems for generation of checkpoint code have been developed[8], they must be conservative and will generally identify more state than strictly necessary. On the other hand, functional languages implicitly determine the required state and thus a programmer doesn’t need to consider what must be saved.

Networks of Workstations (NoW) are an economic means of assembling high performance systems: demand for higher performance personal computers has ensured ready availability and low cost of fast processors and network components. NoW systems may also exploit the power that sits idle on many desks. However using this resource has a fundamental problem: most of the processors are not ‘owned’ by those who would exploit their power. The actual owners rightly consider that they may turn off, disconnect, re-configure or otherwise make their machine unavailable without notice. The distributed nature of such a system also makes it susceptible to faults - such as accidents with power cords - against which a centralised machine is more easily protected. A system that will tolerate such faults is thus enormously valuable: it provides vast amounts of ‘free’ cycles for useful work. This paper describes the performance of a low-overhead fault tolerant software system based on Cilk[5, 4]. In this fault-tolerant Cilk (FT-Cilk), the dataflow semantics enable the run-time system (RTS) to determine the active state of the computation at all times: programmers do not need to add code for fault tolerance. It is sufficient that the hardware and communications networks are able to detect errors, e.g. parity memory is used and network messages have CRC words. A processor with a memory error may simply terminate. A corrupted network message is simply dropped. The RTS will detect that a fault has occurred and restart lost computations. This class of fault is often termed ‘fail-stop’.

Use of machines by their owners - who will reasonably demand priority on their machine - may result in irregular and unpredictable response times. The functional nature of FT-Cilk ensures that, while some work may get duplicated, the whole computation proceeds correctly to the final result. To be useful for long and complex calculations, a NoW system must continue towards a correct result in the face of:
individual machine failures - whether from hardware, software or accidental causes,
loss of messages - whether from link breaks or corruption from hardware faults or E-M interference,
removal of a machine by its owner and
unpredictable response times - interactive use of a machine generates bursts of intense CPU demand interspersed by periods of negligible demand.

Some of these problems are not really faults at all but result from the distributed ownership and location of individual processors. The cost of fault tolerance must also be considered: complete replication of every computation thread wastes resources and mitigates the benefit of using multiple processors in the first place. Now systems - generally built from relatively low bandwidth commodity components - are very sensitive to poor communication bandwidth: memory and CPU cycles are ‘cheaper’. FT-Cilk’s overheads are mainly memory and to a lesser extent CPU cycles. The fault tolerant nature of the whole system allows unreliable communications protocols to be used when the system is basically reliable - FT-Cilk uses UDP rather than TCP/IP to achieve some performance gains.

Cilk is described briefly in Section 2. Section 3 details the system used. The fault tolerance mechanisms of FT-Cilk are elaborated in Section 4. Experimental results showing the cost of fault tolerance in FT-Cilk are in section 5. Section 6 reviews some related work and section 7 summarises the benefits of this approach to fault tolerance.

2. Cilk

Cilk enhances C with a dataflow computation model[5, 4]. Cilk or thread keywords declare threads and spawn activates a thread. A pre-processor converts a Cilk program to C. FT-Cilk uses the Cilk 2.0 pre-processor - modifications were made to the RTS only. When a thread is spawned, a closure is created, which contains the complete context of a computation. Cilk’s parallelism is based on this: once a closure is full, it may execute on any processor. The principal components of a closure are (cf. Figure 2): (a) a pointer to the code for the thread, (b) argument slots which hold data used by this thread, (c) continuations pointing to argument slots in other closures where this thread will post results and (d) a join counter - a count of data items which must be supplied before the closure becomes full and ready to run. When a thread is spawned, the argument slots may be either full or empty. When all argument slots are full (the join counter is 0), the thread is ready to run and is moved to a ready queue. Continuations point to empty argument slots: other closures use them to send arguments to a partially empty closure.

2.1. Parallelism

An idle processor (a thief’s) RTS starts sending steal requests to randomly chosen target processors - ‘idle-initiated’ work-stealing[3]. If a target processor has a full closure, it will migrate it to the thief.

```c
#include "Game.h"
cilk int EvalMove( Game g, Move m, int depth) {
    /* Call a standard C function */
    return MoveScore( g, m, depth );
}
cilk Move BestMove( Game g, Move ml[], int n_moves, int depth) {
    int k, score[n_moves];
    for(k=0;k<n_moves;k++)
        /* Spawn threads for each move */
        score[k] = spawn EvalMove( g, ml[k], depth )
        /* Wait for thread return values */
        sync;
    k = MaxScore( score, n_moves );
    return ml[k];
}

void cilk_main(int argc, char **argv) {
    Move *ml, best_move;
    int n_moves;
    g = ReadGame( argv[1] );
    depth = atoi( argv[2] );
    ml = GenerateMoves( g, &n_moves );
    best_move = spawn BestMove( g, ml, n_moves, depth );
    sync;
    PrintMove("Best Move: ", best_move );
}
```

Figure 1. Cilk Game Playing Program

Fig 1 shows an outline of a simple Cilk program for game playing. After pre-processing, it is compiled and linked with the RTS. The resultant image is sent to each PE. Figure 2 shows the general structure of a Cilk program with closures and the continuations which enable data transfer between threads.

2.2. Cilk Runtime System

The Cilk RTS manages the system startup, the running of threads and orderly termination. Its principal component is the scheduler loop which
- invokes the handlers for incoming messages,
- examines the ready queue for full closures,
- runs threads and
- initiates work stealing when there is nothing else to do.
Messages sent to remote processors invoke a handler on that processor. The whole RTS comprises ~2000 lines of C and is relatively easy to port. In the southern hemisphere, an early version of FT-Cilk was ported to Macintosches[7].

---

1Cilk 2.0 was used in this study: later versions provide shared memory capabilities which depart from the functional model which is the key to the fault tolerance capability.
3. The Myrmidons

Experiments were run on a dedicated NoW - the Myrmidons\(^2\) - consisting of 6 ‘slave’ 150 MHz Pentium PCs connected by a private 100MHz ethernet to a "host" 200MHz Pentium Pro PC. The Cilk RTS uses PThreads under Linux. A Cilk program running on one of the Myrmidons actually consists of two threads: a computation one and an input/output one.

Although the computation thread could handle all messages since the dataflow semantics make it immaterial in which order messages are processed, the separate I/O thread permits immediate response to steal requests - on which the thief is blocked. Communication uses UDP - the standard connectionless, unreliable datagram protocol. Using a less reliable, protocol necessitates invoking FT-Cilk’s error recovery mechanisms when messages are lost or corrupted. To demonstrate the effectiveness of FT-Cilk, three versions of the RTS - differing in the communications style and overheads - were used;

- FT-Cilk
- an ‘Unsafe’ version using UDP without the fault tolerance overheads and
- a version in which messages are ACK’ed and senders retry messages. A sliding window protocol tracks un-ACK’ed messages. This emulates a higher overhead, reliable protocol like TCP/IP.

Due to the dual-thread implementation of FT-Cilk, a program invoking a thread on another processor waits several ms for multiple context switches to occur. Thus, for reasonable efficiency, one must ensure that threads run for several hundred milliseconds. No attempt has been made to improve the ethernet drivers (as in UNet[11]) or PThreads implementation because a parallel project has developed a higher bandwidth switching device, Achilles[9], with ‘zero-copy’ drivers. To run FT-Cilk programs, a server is started on each Myrmidon. A master program on the host broadcasts the FT-Cilk program to the slaves, master and slaves then fork processes which run it.

4. Fault Tolerance

The mechanisms used by FT-Cilk to tolerate faults are enumerated here: in each case, the additional overhead required is elaborated.

4.1. Steal requests

Whenever it is idle, a PE (the thief) randomly targets another PE and sends it a steal request. If the target PE has no full closures which can be migrated, it calls the steal_failure handler on the thief. However, if work is available, a closure migrates to the thief. Firstly one can note that it is unnecessary to ensure the receipt of the steal_request message: if the target PE had failed, the thief selects another target. It is possible that the initial target had not actually failed but is busy or the time-out value was set too optimistically (due, for example, to a temporary heavy network load). In this case, at some later time, the target will send a steal_failure, which can simply be ignored, or a closure to be executed. In the meantime, it’s likely that the thief has stolen work from another target: it may thus become temporarily overloaded, but the computation will still proceed correctly. Thus, our robust implementation can not only use an unreliable protocol for the request transmission, it can use an aggressive time-out parameter - helping to avoid processors idling for extended periods.

Fault tolerance has not been achieved without some cost: a too-anxious thief could have obtained two critical closures (ones on the critical path of the computation) and thus have blocked the whole computation for a while. However, it is assumed that the cost of restarting the whole task from scratch is high, so that it is preferable to have computation continue, albeit somewhat handicapped by an illusory fault, than to have it abort entirely from a real (and probable) one.

4.2. Closure Migration

Closure migration - in response to a steal_request or programmer-generated work distribution - is another operation which can be made sufficiently fault-tolerant that lightweight protocols can be used. To ensure fault-tolerance, the ‘home’ of each closure (the PE on which it was originally spawned) retains a ghost copy of a migrant closure tagged with the index of the PE to which the closure migrated. In the absence of faults, when the migrant closure is executed,
it is freed and a message is sent to the home to enable the ghost to be freed. A number of faults can occur:

1. the migrant is never received,
2. the migrant is received, but the receiving PE dies before executing it,
3. the migrant is received and it is partially executed, sending some results via continuations in the migrated closure, before failing or
4. the migrant is executed, but the message to free the ghost is never received by the home.

In all these cases, when it runs out of other work, the home PE will examine the list of ghosts which it is maintaining and query the PEs to which they were sent to determine the status of the ghosts. If the receiving PE responds that the migrant is still ‘alive’, then the ghost is marked alive. However, if the recipient fails to respond or responds with no knowledge of the ghost, then the idle PE executes it. The side-effect free nature of Cilk thread execution ensures that a repeat execution of any thread will produce identical results. Even in case 3, where some results have already been despatched to waiting closures, the results despatched from any execution of the same closure will be identical. Thus, at first glance the repeat execution produces no problems. However, there are two potential problems. To avoid the join counter being decremented prematurely to zero by a second arrival of the same data, the join counter was implemented as a set of bits - one for each slot in the closure. Bits are cleared when data is sent to a slot. Duplicate data is discarded. A second problem can arise if the closure receiving the original data became full, was executed and then freed. The continuation from the duplicate copy of the sending closure is now ‘dangling’ - the closure to which it pointed no longer exists. A unique identifier is added to each closure so that the address in the continuation can be recognized as no longer representing a valid closure. The identifier is formed from the PE index and a closure serial number for the PE, so that unique identifiers can be formed without any communication. The assignment of identifiers and their inclusion in continuations is handled by the RTS transparently to the programmer.

4.3. Ghost closures

The ghost closure list is a list of all work despatched by a PE to logical children of any closures executed on that PE. Once the ghost is freed, this is a signal that all this closure’s children have been executed. Retaining the ghost closures has the potential to cause resource problems in personal computer systems with relatively small physical memories and small (or no) virtual memory spaces. It is possible to retain in memory only a small part of the full closure and to write the bulk of the closure to secondary storage. The closure allocator could perform this back-out to disc when closure space was exhausted, i.e. only when required. If the system is generally reliable, the probability that the data of a closure will be read back for re-start is small. However, as virtual memory would generally be more efficient for this purpose, this capability will not be implemented immediately.

4.4. Data posted to closures

Closures with empty slots become ready when PEs post data into them. As a closure waiting for data can be on the critical path for a computation, FT-Cilk requires acknowledgement for these messages. A failure to receive an acknowledgement from the receiving PE’s input/output thread after a number of retries initiates a trace-back through the tree of spawned closures to locate the parent of the lost closure. Each closure must thus keep track of its parent and active children. Final freeing of ghosts is deferred until all children have signalled that they have completed successfully.

5. Results

The experiments reported here were principally designed to measure the overheads needed for FT-Cilk and to demonstrate that use of unreliable communications protocols can enhance performance when the underlying hardware is generally reliable. Two programs were used in the experiments: (a) a recursive fibonacci number calculator and (b) a chess playing program.

5.1. Fibonacci numbers

The simplest version of the standard (yet inefficient) recursive program has a trivial computational load for each thread and measures overheads in a parallel system. The program used here was modified to spawn threads only for numbers above some threshold. For numbers below this threshold, Cilk’s call mechanism invokes the same function without spawning a new thread. This allowed the granularity of each thread to be adjusted to the communication overhead of the whole system. Numbers below fib(31) were calculated sequentially - a workload of about 0.6s for the processors used.

5.2. Chess

This program implements a parallel version of the alpha-beta algorithm described by Akl[1]: it was chosen for these experiments because of the irregular workload generated. Initially, threads are spawned to calculate the left branch of
the game tree (a major computation) and the left sub-trees of right branches (a large number of smaller computations). The same strategy is repeated when right branches need to be evaluated fully. A middle-game problem with a losing ‘obvious’ first move (a queen capture) [2] was evaluated to a depth of 6 plies. Threads with running times between 10ms and 10s were generated.

![Figure 3. Speedup: Fibonacci numbers](image)

The data presented in Figures 3 and 4 show the speedups for three versions of the run-time system:

- FT-Cilk with the fault tolerance overhead,
- ‘Unsafe’ - no message acknowledgement or re-try and
- ‘ACK’ - messages are acknowledged and re-tried.

Each experiment was run at least 5 times: the error bars show the range of values measured. Due to the inability to synchronise thread start up with a ‘standard’ operating system and the random nature of the work-stealing, the number of threads run on each PE varies. Speedups were measured relative to the average time for the same Cilk program running on a single processor. As expected, the ‘Unsafe’ version shows consistently higher speedups - it lacks the overhead of FT-Cilk and, optimistically, assumes that messages will never be lost or corrupted. There is very little difference between FT-Cilk’s RTS and the ‘ACK’ RTS: FT-Cilk would appear to have slightly greater speed-up potential, but more trials are clearly needed to confirm this. The key observation from this data is that any overhead of FT-Cilk is small, i.e. the aim of achieving fault tolerance with negligible affect on overall execution time in the absence of faults has been met.

With chess, speed-ups taper off more rapidly as the number of processors increases - a consequence of additional work as parallelism is increased. Variations due to inability to synchronise and random workstealing are as large as differences between the three RTS’s: the ‘Unsafe’ RTS appears to perform slightly better. A small overhead for FT-Cilk relative to the ‘ACK’ RTS may be apparent: this probably reflects the increased CPU overhead. In the ‘ACK’ RTS, acknowledgements are sent asynchronously and do not stall the computation so are not affecting the overall time with the large grain sizes and thus low total numbers of messages. However, again the key observation is that the effect on performance of the fault tolerant capabilities built into FT-Cilk is small if it exists at all.

Despite the retention of significant numbers of ‘ghost’ closures in FT-Cilk, total memory overheads in these two examples are insignificant: the closures themselves are small. Some factors naturally mitigate this memory overhead: communications overheads mean that to show effective speed-up, programs must be tuned to computation/communication cost ratios - ethernet’s low bandwidth necessitates large granularity threads. This means smaller numbers of threads and smaller numbers of ghosts. FT-Cilk memory overhead can be large when large amounts of data and long-running programs are involved. The OS will eventually move all the ghosts to disc - from which they may never need to be read again. Once they are freed, the next closure to use the same virtual address will write new data without reading the old data. (To fully exploit this consequence of the dataflow semantics may require some OS adjustments though!) If necessary, most ghosts could be ‘archived’ at relatively low cost to backup store and a smaller directory of pointers to the archived ghosts maintained in fast memory. This represents a form of automatic continuous distributed checkpointing. In the absence of faults, the ghosts will never be read back into memory again.

![Figure 4. Speedup: Chess program](image)
6. Related work

This study has focussed on one aspect of fault-tolerance: PEs are assumed to either function reliably (but may have unpredictable response times) or fail completely. Incorrect results are detected and simply abandoned. Thus the Byzantine Generals [6] and similar problems have been ignored. FT-Cilk is optimistic: PEs generally complete their tasks, so it only duplicates work when it believes that a failure may have occurred. Tridandapani et al. [10], on the other hand, are more pessimistic: they use idle power for speculative task duplication. Their system also relies on the ability to simply compare the output from two tasks to detect faults. FT-Cilk could use idle PEs for duplicate ghost execution: closure lists form a convenient data structure for monitoring progress. Duplicate transmissions of data items to slots in closures are also readily checked for consistency.

A centrally controlled fault-tolerant Cilk system has been built by Blumofe [3]. I believe that the distributed control described here is more efficient: there is no central monitor process to become a bottleneck, either for routine messages which make fault recovery possible or for recovery messages when faults are detected.

7. Conclusion

A fault-tolerant RTS for NoWs that only requires a programmer to conform to a functional programming model has been implemented. To provide fault-tolerance, the Cilk 2.0 RTS has been augmented with:
- Bit per slot join counters
- Unique identifiers for each closure
- Unique closure identifiers in continuations
- Parent links in each closure
- List of children in each closure
- Ghost closure list
- Closure status handlers
- Free ghost handlers
- Handlers to locate lost closures

These additional items present very little overhead when closures are spawned. The largest dynamic overhead in normal operation is the need to invoke the free_ghost handler when a migrated closure frees itself. Since FT-Cilk dispenses with message acknowledgements, it substitutes the free_ghost message for the acknowledgement with negligible measured overheads for fault tolerance. FT-Cilk provides distributed recovery mechanisms - rather than centralised control [3]. If a PE dies, it ceases work stealing and eventually PEs which sent it work will discover the failure and spawn new closures to replace lost ones.

FT-Cilk derives its ability to provide fault tolerance from its underlying functional or dataflow semantics: this study has demonstrated that it is possible to exploit this capability to tolerate a wide range of potential faults - without additional programmer effort - at low cost to the performance of the system. This approach could be applied to any functional language, but Cilk has a particular advantage - its C parentage would seem to make it less daunting to those with large computational problems. Extensions to FT-Cilk’s current capabilities which are currently planned are a virtual PE labelling system - a Cilk programmer may explicitly direct work to a PE - so that closures may be re-directed to a live PEs as soon as a failed PE is detected. An implementation which uses the high bandwidth and ability to move some simple handler code to the network interface of the Achilles switch [9] has also been developed: this reduce communication costs by orders of magnitude and increase the range of problems which can be run effectively.

References