Literature on Threads in Functional Programming Languages – Version0.2

Ananda M. S. Amatya*
Department of Computer Science
University of Warwick

January 8, 1998

Abstract

In this paper we review some recent work done in adding threads to Functional Programming Languages with a view to making them faster and more powerful by allowing parallel processing.

1 Introduction

The purpose of this literature search on the use of light-weight threads in Functional Programming Language implementations is to determine the best strategy for incorporating Java threads in the Ginger parallel implementation. A thread is a sequence of actions as successive statements are executed when a program is run in a processor. For an introduction to threads see [5]. A Functional Programming Language allows constructions of function definitions with a view to evaluating expressions containing them [4]. Threads in functional programming will therefore evaluate part or whole of such expressions. The goal is to improve the speed and performance of functional programs by executing such threads in parallel [8], [12], [15].

2 Threads in SML

Among the first successful implementations of threads in Functional Programming Languages was the work by Cooper and Morrisett [7] in which they describe how threads have been added to Standard ML by providing an SML Module that handles threads, without making any modifications to the SML language syntax.

- A thread module provides *fork* control structures to create threads. Fork is a function that starts an invocation of its argument executing as an independent thread of control. The effect is to have two threads, the parent (the thread that invoked fork) and the child (the thread that executes the argument passed to fork).

*E-mail ananda@dcs.warwick.ac.uk

- No *join* control structure is required as thread termination occurs either implicitly on returning from its call or explicitly when *exit* is used.

- Communication between the threads is via shared mutable objects. Mutual exclusion between threads is managed using *mutex* locks with *acquire* and *release* operations. A busy waiting version of *acquire* uses *try_acquire*. Another operation *with_mutex* acquires mutex during the function execution releasing it before the return from the function.

- Synchronisation and termination of threads is via wrapper functions. Synchronisation between threads is achieved using condition variables which are implemented using *condition*, a function to create a new condition variable, *mutex_of*, a function to return the mutex associated with a condition, *with_condition* function which is just the composition of *with_mutex* and *mutex_of*, *signal* operation to signal the occurrence of an event, *broadcast* operation to signal to awaken all threads waiting for the condition, *wait* operation to wait for a specified condition variable; also provided are *await* to test before wait operation and *yield* to allow scheduling another thread on the current processor.

- *var* type is provided for thread states. This allows independent thread identification in different subsystems.

- It is claimed that higher level abstractions like Recursive Mutual Locks, Futures (as in Multilisp), Channels, CSP guard commands, Ada rendezvous and select, and Reppy's events could be built on top of these basic constructs.

- Two different implementations using continuations are discussed: one uses coroutines for multiplex control, and another uses signals and asynchronous signal handling facility for preemption.

- Modifications of this thread package for multiprocessor implementation such as with Mach operating systems [21] are discussed [9] & [1].

- Comparison with C-coroutines and C/Mach suggest optimisations are required. No asynchronous communication with the threads is available in the current implementations.

- Formally, integration of SML with threads really requires a new definition of SML. It is suggested that a dynamic semantics of SML/Mach using CSP or CCS concurrency models might provide a firm theoretical basis for such a threads extended SML.

3 CML

Another approach to adding threads to SML is due to Reppy [19] in which a running program consists of threads which communicate and synchronise by passing synchronous messages on typed channels.

- *spawn* function takes a function as an argument and dynamically creates a new thread that evaluates the application of the argument function.

- *channel* function dynamically creates a (weakly) polymorphic channel. SML's Lexical scoping is used to share channels between threads, and to hide channels from other threads.

- *send* and *accept* operations deal with synchronous communications. They complement each other. To communicate to a channel a thread must rendezvous with another thread that wants to do complementary communication on the same channel.

- CSP style rendezvous mechanism with selective communication is implemented using *events*. According to Reppy, this achieves a level of abstraction that ensures security and ease of maintainance. *events* are a type of values that represent synchronous operations. *base_events* represent communication operations. *receive* and *transmit* functions build on these *base_events*. Combinators for associating actions to events and to build nondeterministic choices of events are provided. *wrap* combinator associates the action to an event. *choose* combinator builds the non-deterministic choice. A *sync* operator is provided to force synchronisation on an event. This formalism allows a programmer to create first class synchronisation and communication abstractions. Other synchronous operations like thread termination, low-level I/O support and time-outs can be expressed in terms of *events*. They are *wait* for termination, *syncOnOutput* and *syncOnInput* for I/O and *waitUntil* and *timeOut* for time-outs. CML has a uniform mechanism for combining synchronous operations. It provides *guard* and *wrapAbort* combinators to deal with commit points in communications. Concurrent versions of I/O primitives are provided.

- CML has been used in interactive systems to build a multi-threaded interface to the *X* protocol, called *eXene*, providing similar functions to the *Xlib*. Another area of application of CML is distributed systems programming. Especially interesting is the use of CML in *Nuprl* (an interactive proof system for ML) to distribute the parts of a proof among the pool of proof servers across a network of workstations.

- Implementation languages used is SML and its extensions (first-class continuations (for threads) and asynchronous signals (for preemptive scheduling)). Heap based implementations are particularly suited for light-weight threads (though stack-based ones are also possible).

- A CSP based formal semantics for CML has been given [20]. Some recent work in this area are reported in [6] and [3].

4 GAML

GAML is a parallel implementation of lazy ML [17] on a shared-memory computer.

- Several G-machines cooperate in reducing a graph concurrently with the intention of speeding up the reduction.

- Annotations in the source program determine partitions to be evaluated in parallel. Annotation consists of '#' (usually, called *fork* or *spark*).

- Automatic parallelization (with information of what the compiler is doing) is possible. Locking and unlocking of tags are used to ensure that when sharing takes place, a graph being run on one G-machine cannot be run by another G-machine. For performance improvement, the G-machines must cooperate. A scheduling of *fork pools* is necessary.

- Heap management divides the heap space into *from-space* and *to-space* pages, a *to-space* page being allocated by a global manager to each processor initially. A processor that runs out of its heap page can request for more heap *to-space* page. If this request cannot be fulfilled, there is a page fault, and the processor will enter a collector state. All the other processors will end up in the collector state naturally when they have used up their allocated heap page. Stop and Copy garbage collection is performed after swapping the *to-space* and *from-space* pages of the heap. Parallel garbage collection of all running processors is done, but program execution is halted.

- The stacks for unwinding the spine, for intermediate results of the arithmetic computations, and for remembering the G-machine states, do not require constant checks for overflow due to parallel implementation.

- Parallelism is seen to be beneficial. However, as the number of processors is increased, garbage collection time also increases. This is due to more objects to be copied during garbage collection. Furthermore, this also increases the number of stacks and hence the cost due to stack overflow checks.

5 $< \nu, G >$ machine

$< \nu, G >$ machine [2] is a modification of the G-machine in order to allow parallel graph reduction of lazy ML on a shared memory multiprocessor (a 15 processor Sequent Symmetry machine). Both sequential and parallel versions are possible.

- It uses stack frames (one small stack for each single evaluation) in the heap.

- Only stack frames need locking as there is less synchronisation during evaluation of a node.

- Heap allocation simply increments the heap pointer. Each processor allocates a chunk of the global heap space when it needs more heap space. When a processor cannot allocate more heap space, garbage collection is performed.

- Garbage collection requires all the processors to be synchronised first. Sequential garbage collection is used (possible system deterioration due to the garbage collector being called too often).

- A frame node is used to represent a suspended function application; it holds the arguments of the application, plus a pointer to the code for the function, and some space for temporary variables for intermediate results during reduction. It also has a dynamic link field which holds a pointer to the frame node from which evaluation was called. Recursive evaluation links together individual frame nodes to form a stack frame of nodes. Several such frames correspond to the evaluation by parallel processes. Together these stack frames form a *cactus* stack.

- Spark model is used for parallelism, with one point of evaluation for each evaluating process. Sparks are advisory only. If for some reason, evaluation has not started the process needing the value does the evaluation itself.

- The $< \nu, G >$ machine behaves like a stack machine. As in G-machine, compilation rules follow compilation schemes. Instruction transition rules deal with the instructions emitted by the compilation schemes. The main difference between the G-machine and the $< \nu, G >$ machine is the use of individual stack frames to evaluate the graph. In the parallel version of the $< \nu, G >$ machine there are many reduction points where reductions take place concurrently. There is only one reduction point in the sequential case. Also, in the parallel case there is a *spark pool* of nodes waiting to be evaluated.

- One point of reduction for each evaluating process. Time slicing is used to guarantee that all evaluation points make progress.

- Flag bit on a tag in the node for exclusive access using test and set and busy wait until lock acquired.

- A process suspension mechanism so that a process can be waiting but the processor can continue doing something else is implemented.

- Real speed ups (5 to 11) on standard G-machine evaluations on LML ares reported.

6 GRIP

GRIP (Graph Reduction In Parallel) is a shared-memory parallel machine (i.e., multi-processor) to support implementation of Haskell [11] for effective performance of programs by executing them in parallel using compiled supercombinator reduction [10]. It uses the abstract STG machine [14] for code generation which concerns mainly with thread creation and scheduling.

6.1 Thread Creation and Scheduling

- An expression to be evaluated is represented by a graph of *closures*, each of which is held in a heap. Each closure consists of a pointer to its *code*, together with zero or more *free-variable* fields. Some closures are in *head normal form*, in which case their code is usually just a return instruction. Other closures represent unevaluated expressions, in which case their code will perform the evaluation. A closure is evaluated by jumping to the code it points to, leaving a pointer to the closure in a register so that the code can access the free variables; this is called *entering* the closure. When evaluation is complete, the closure is *updated* with (an indirection to) a closure representing its head normal form.

- A *thread* is a sequential computation whose purpose is to reduce a particular sub-graph to normal form. At any moment, there may be many threads available for execution in a parallel graph reduction machine; this collection of threads is called the *thread pool*. A processor in search of work fetches a thread from the thread pool and executes it. A particular physical processor may execute a single thread at a time, or may split its time between a number of threads. During execution, a thread may encounter a closure whose value will be required in the future. In this case it has the option of placing a pointer to the closure in the thread pool, where it is available for execution by other processes - this is referred to as *sparking* a child thread.

- Inter-thread communication and synchronisation occurs only when a parent thread at some stage requires the result of a sparked closure which has started execution. In that case, the parent becomes *blocked*, until the sparked closure completes evaluation. On receiving the evaluated result, the blocked parent thread resumes processing. An evaluated expression (in Normal Form) may be used by any thread simultaneously without any contention. This is referred to as *evaluate-and-die* model for blocking/resumption. A parent gets blocked only when the sparked thread has already started but has not completed yet. A sparked thread is ignored, and the parent itself evaluates the graph if it reaches the point when it needs the result and the sparked thread has not yet been allocated to a processor for execution. Such an *orphaned* sparked thread is removed at the next garbage collection.

- The global address space holds the graph to be evaluated as variable sized graph objects - pools of executable threads. Local address space on each Processing Element (PE) is used for evaluating one or more threads acquired from the pool. Heap objects in the global address space are grouped into evaluated and unevaluated ones. An unevaluated object has a bit which is set when fetched by a PE. This locks the object and makes it unavailable by any other PE. When evaluation is completed the object is overwritten with its result and becomes an evaluated object. All evaluated objects are available for simultaneous access by all the PEs without contention. Global heap is accessible to each PE. Local heap is held in local memory of a PE and is private to it. Caches of closures are held there, while the code for currently evaluated closure is held in local memory. Local heap can be garbage collected independently and in parallel with graph reductions by other PEs. Global garbage collection will affect all the PEs, but is rarer. Closures are copied from the global heap to the local cache held by a PE when needed. An evaluated closure from a PE is 'flushed' out to the global heap if it is a copy of a closure in the latter. The only other occasion when this 'flushing' out from a PE to Global heap occurs is when the PE is overloaded or when the system load is low and othe PEs are idle. This ensures that the performance cost due to 'flushing' is minimised.

- GRIP results are mixed, and the approach is prototypical.

7 GUM

7.1 Summary

GUM (Graph reduction for a Unified Machine model) is a portable Haskell parallel implementation [22], The basic design concepts are more fully described in [13], [14].

- It is a runtime system for GHC. [16]
- It is portable.
- It is message based and uses PVM.[18]
- It requires networked processor-memory units (PE).
- It provides tools for monitoring and visualising PE activities as well as thread activities over time.
- Two-level addressing scheme: one for local heap and another for the global heap is used. Global virtual heap is formed from all the local heaps.
- It supports independent garbage collection on each PE's local heap. Garbage Collections for the individual PE local heaps within a single global virtual heap without any need for their synchronisation. Each PE implements part of the global heap in its local heap.
- Threads distribution is performed lazily: no thread movement between PEs for load balancing; thread moved to another PE only if the latter is idle.
- Data distribution is performed eagerly: data copied between PEs as and when requested.
- All messages are asynchronous: this has the effect of favouring long-latency machines in the network.
- GUM bases: Sun Sparcs, DEC Alphas, Sun's symmetric multiprocessors SPARC server.
- Significant speedups (for problems with large-grained parallelism) is claimed.

7.2 Thread management

- A thread is a virtual processor with Heap allocated TSO (Thread State Object with slots for registers) which has pointers to heap allocated SO (Stack Object). As the SO grows more SOs are added and chained to the latest SO.
- Each PE has a pool of runnable threads (TSOs): runnable pool.
- Once a thread has been started on a PE, it cannot be moved to another PE. A thread runs non-preemptively until it completes, or space runs out, or it blocks (accessing remote data, or waiting for result of another thread). This reduces space usages and run-time.

7.2.1 Limitation of current implementation

- No thread migration: a thread once executed on a PE cannot be moved to another PE, leading to uneven loads on PEs.
- No thread preemption (reduces space usage and overall run time): makes concurrent and speculative execution harder.

7.2.2 Spark

- *par* - a parallelism construct in Haskell program.
- A combinator that (explicitly) initiates or sparks a *thunk* for (possible) parallel evaluation.
- Sparking a thunk amounts to adding a pointer to the thunk to the PE's spark pool.
- A sparked thunk may be (but need not be) evaluated in parallel; sparks may be discarded if there are too many of them.
- A PE keeps its waiting thunks in its spark pool, which may migrate to an idle PE.

7.2.3 Synchronisation

- Need to ensure that a thunk is evaluated only once, and no two threads evaluates the same thunk simultaneously.
- This is achieved by requiring a thread entering a thunk to overwrite the thunk with a *black-hole*. A thread entering a black-hole saves its state in its TSO, attaches its TSO to the queue of threads blocked on the black-hole and enters the scheduler.
- A thread completing the evaluation of a thunk overwrites it with its value and moves its TSO to the runnable pool.
- Synchronisation costs only when two threads collide.

7.2.4 Load distribution

- An idle PE tries to schedule a spark from its spark pool. If that spark is under evaluation by another PE or has already been evaluated (orphan) then the PE tries another spark (in FIFO). A useful spark is turned into a thread by allocating a fresh TSO and SO and starting to execute it.
- If no useful local sparks are available, the PE seeks work from other PEs. No PE is left idle while there are sparks waiting to be evaluated in other PE spark pools. A message passing mechanism using a sequence of *FISH*, *SCHEDULE* and *ACK* messages migrates unevaluated sparks between requesting and supplying PEs. A *FISH* message sent out by an idle PE looking for work, goes from one PE to another until work is found. To avoid swamping the system with too many circulating *FISH* messages, some kind of *ageing* mechanism that increments a count field in the message for each PE visited unsuccessfully is used. This is used to determine whether a previously set limit on the number of PEs that a *FISH* message is allowed to visit is reached. In that case, the last visited PE returns the message back to the PE that originated the message. Retransmission of the *FISH* message is allowed only after a certain delay interval. Also, one PE may have at any time at most one outstanding *FISH* message. If a PE that receives a *FISH* message has a useful spark, it sends a *SCHEDULE* message back to the originator of the *FISH* message. The *SCHEDULE*

message contains the spark thunk packaged with "nearby" graph. The originating PE unpacks the graph and adds the newly acquired thunk into its local spark pool and sends *ACK* message to the sender of *SCHEDULE*. The spark may not be evaluated straightaway as the PE may no longer be idle, because it might have received a message unblocking some thread, in the meantime.

7.3 Memory management

GUM needs to manage the virtual shared memory in which the graph to be reduced resides taking into account,

- Local addresses: Refer to addresses in a PE's local heap. New heap objects formed from the local free heap space in the PE. Local addresses are pointers to local heap objects. Local garbage collection affects only local addresses of a PE without affecting Local addresses of other PEs. A heap object moved from one PE to another (due to *FISH* message) will have its original thunk overwritten by a *FetchMe* object, containing the global address of the new copy of the thunk on the receiving PE, to ensure that the thunk gets evaluated only by the receiving PE.

- Global address: Points between local heaps of different PEs. In order to allow local garbage collection, the global address consists of a pair (PE address, local identifier), where each globally visible object is given an immutable local identifier. Each PE maintains a Global Indirection Table (GIT): maps local identifiers to the local address of the corresponding heap object. GIT source of roots for local garbage collection. Updated after garbage collection. A heap object is globalised by allocating an unused local identifier and augmenting the GIT to map the local identifier to the object's address. If the object already has a global address no globalising is required. In order to allow this a LA→GA table is maintained for each table that matches local address to a global address. This table is referred to before a heap object is globalised. The LA→GA table has to be reconstructed following a garbage collection. as the local addresses alter during garbage collection. A PE may hold copies of gloabally-visible heap objects owned by another PE. To save having to fetch again, such copies are given the same Global Addresses as the original ones, so that they can be shared. A GA→LA table (that maps foreign global addresses to their local counterparts) is maintained for each PE, a new GA fetched to a local copy LA is added to the table. The three tables: GIT, GA→LA and LA→GA are implemented as a single data structure (Hash).

- Garbage collection: Local address garbage collection need not be synchronised with other PEs; A heap closure with an entry in GIT cannot be garbage collected, also its local identifier cannot be re-used. Weighted reference count garbage collection is used to recover the identifiers and corresponding heap closures. This requires the GIT table and the GA→LA table to deliver a weight along with the local address.

- Lazy thread distribution: Threads exported from one PE to another only when the latter is idle; Sending PE retains a copy (to maintain locality).

- Eager data distribution: "Neighbouring" data packed along with requested data being sent (for locality).
- Asynchronous message passing between PEs: No waiting for replies to messages sent.

7.4 Packaging and unpackaging graph

Packing and unpacking of graphs is done to reduce network latency costs in passing them between PEs. Algorithms and heuristics for packaging must preserve sharing and cycles within the packet.

7.5 PVM environment and GUM

GUM starts off by creating a PVM manager task that controls startup and termination. The PVM manager spawns required number of tasks as logical PEs and maps them to the actual PEs. Each PE task then initialises itself processing runtime arguments, allocating a local heap etc. After all the PEs have been initialised, one PE is regarded as the main and evaluates the Main thread of the Haskell program. On completion of the Haskell program the main PE sends FINISH message to the manager task, which broadcasts it to all the PE tasks. When all the PE tasks have responded, the manager terminates the program. Each thread follows the schedule:

- Perform local garbage collection (if necessary);
- Process any incoming messages from other PEs (sending replies if necessary);
- If there are runnable threads, run one of them, otherwise look for work.

7.6 Results

Significant speed-ups are possible for programs with significant large-grain parallelism. However, no conclusive evidence yet. Tests on 512 Sparc processors at Los Alamos National Laboratory are planned. Applications that use GUM: Natural language processing (LOLITA), database queries. A uni-processor simulation of GUM is GranSim.

8 Conclusion

The GUM (and GranSim) approach seems particularly promising and needs further study. It is hoped that the implementation details of these will map easily to the parallel Ginger implementation using threads. It is hoped that the conversion into Java threads should not be too difficult. The real problem could be disentangling the essential thread model from the intricacies of the GUM implementation. A possible outline of steps to follow:

1. Familiarise with GUM and GranSim by experimenting with some example programs.
2. Read the papers relating to their design and implimentation.
3. Look at the source codes for relevant modules, if necessary.

4. Determine the algorithms in the relevant functions.

5. Design and implement in Java the corresponding code for ginger.

A longer term goal could be to look at Reppy's implementation of CML and see whether any improvements of the thread model are possible.

References

[1] Accetta, M.J., Baron, R.V., Bolosky, W., Golub, D.B., Rashid, R.F., Tevanian, A. and Young, M.W., *Mach: A new kernel foundation for UNIX development* in Proceedings of the Summer 1986 USENIX Conference, Pages 93-113, July 1986.

[2] Augustsson, L. and Johnsson, T., *Parallel graph reduction with the* $< \nu, G >$*-machine* in Functional Programming and Computer Architecture, pages 202–214, ACM, 89.

[3] Biagioni E., Cline, K., Lee, P., Okasaki, C. and Stone, C., *Safe-for-Space Threads in Standard ML*, Carnegie Mellon University, 1997.

[4] Bird R. and Wadler, P., *An Introduction to Functional Programming*, Prentice-Hall, 1988.

[5] Birell, A.D., *An Introduction to Programming with Threads*, Digital Research Center, 1989.

[6] Blelloch, G.E. and Greiner, J., *Parallelism in Sequential Functional Languages* in Functional Programming and Computer Architecture, pages 226–237, June 1995.

[7] Cooper, E. and Morrisett, G., *Adding Threads to Standard ML* ,Technical report CMU-CS-90-186, Carnegie Mellon University, Department of Computer Science, December 1990.

[8] Davie, A.J.T., *An introduction to Functional Programming System Using Haskell*, Chapter 11, Cambridge University Press, 1992.

[9] Golub, D., Dean, R., Forim, A. and Rashid, R., *Unix as an application program.* in Proceedings of the USENIX Summer Conference, pages 87-95, Anaheim, CA, June 1990.

[10] Hammond K., and Peyton Jones S.L., *Profiling Scheduling Strategies on the GRIP Multiprocessor*, in Proceedings of the Fourth International Workshop on Parallel Implementaiton of Functional Languages, Kuchen H (ed.), Aachen University (September 1992).

[11] Hudak, P., Peyton Jones, S.L., Wadler, P.L., Arvind, Boutel, B., Fairbairn, J., Fasel, J., Guzman, M., Hammond, K., Hughes, J., Johnsson, T, Kieburtz, R., Nikhil, R.S., Partain, W. and Peterson, J., *Report on the functional programming language, Haskell, Version 1.2*, SIGPLAN Notices 27.

[12] Peyton Jones, S.L., *The Implementations of functional programming languages*, Chapter 24, Prentice-Hall, 1987.

[13] Peyton Jones, S.L., Clack, C. and Salkild, J., *High performance parallel graph reduction* in Proceeding of Parallel Architectures and Languages Europe (PARLE), Odijk, E., Rem, M. and Syre, J-C. eds., LNCS 365, Springer Verlag, 193-296.

[14] Peyton Jones, S.L., *Implementing lazy functional languages on stock hardware: the Spineless Tagless G-machine* in Journal of Functional Programming, 2(2):127-202, 1992.

[15] Peyton Jones, S.L., *Implementing Functional Languages – A Tutorial*, Chapter 5, Prentice-Hall, 1992.

[16] Peyton Jones S.L., Hall, C.V., Hammond K., Partain W.D. and Wadler P.L., *The Glasgow Haskell Compiler: A Technical Overview* in Proc. JFIT '93, Keele, (March 1993).

[17] Maranget, L., *GAML: a Parallel Implementation of Lazy ML* in Hughes, J., editor, Functional Programming and Computer Architecture, pages 102-123, Springer-Verlag, August 1991.

[18] Oak Ridge National Laboratory, University of Tennessee, *Parallel Virtual Machine Reference Manual, Version 3.2.* August 1993.

[19] Reppy, J.H., *CML: A Higher-order Concurrent Languages.* In Proceedings of the ACM SIGPLAN '91 Conference on Programming Langugage Design and Implementation. Pages 293 - 305

[20] Reppy, J.H., *Concurrent ML: Design, Application and Semantics*, Cornell University, 1992.

[21] Tanenbaum, A.S., *Distributed Operating Systems*, Chapter 8, Prentice-Hall, 1995.

[22] Trinder, P.W., Hammond, K., Mattson Jr, J.S., Partridge, A.S., Peyton Jones, S.L., *GUM: a portable parallel implementation of Haskell* in the Proceeding of Programming Language Design and Implementation, Philadelphia, May 1996; also an earlier (somewhat more detailed) version of the paper with the same title in Proceedings of the 7th International Workshop on Implementation of Functional Languages, Bastad, Sweden (September 1995).