# Threading Ginger

Ananda Amatya and Mike Joy

Department of Computer Science, University of Warwick,
Coventry, CV4 7AL, U.K.
**Email:** {Ananda.Amatya, M.S.Joy}@dcs.warwick.ac.uk

**Abstract.** This is a report on an ongoing study of the parallel evaluation of lazy Functional Programming Languages using Java Threads and the language Ginger. Issues arising from the use of threads in a shared-memory graph reduction machine are discussed. Also described are some lessons learned about the application and debugging of threaded code. A GUI to assist in the development of quality parallel programs is proposed.
**Key-words:** Functional Programming, Graph Reduction, Java Threads, Lazy Evaluation, Non-determinacy, Parallel Evaluation, Software Quality.

## 1 Introduction

*Ginger*[11] is a lazy functional language that supports parallel programming. The aim of the current Ginger project is to experiment with a small lazy functional language with a view to exploring the main issues of implementing lazy functional languages. It avoids unnecessary diversions into the specifics of handling many *desirable* features of a richer language (e.g. the complex type system of *Haskell* [10] ). On the other hand, it incorporates parallel programming (unlike other simple languages like $Miranda^{TM}$ [21]). With the advent of *Java* [2] it was natural to use it in place of the original C as the implementation language for Ginger. Not only has Java its well-publicised virtues [6], but more importantly, Java already provides the necessary APIs [7] for using threads in the study of parallelism. Some previous work in Functional Programming Languages that used threads were by Cooper and Morrisett [5] in *SML*, Reppy [18] in *CML*, Maranget [14] in *GAML*, Augustsson and Johnsson [3] in the $< \nu, G >$ machine, Hammond and Peyton Jones [8] in the *GRIP* machine, and Trinder et.al. [20] in the *GUM* parallel implementation of Haskell. Historically, Ginger was designed to be of pedagogic value to students learning functional programming as a component of parallel and distributed programming courses in Warwick. It was, and still is, our belief that the elegance of functional programming can be an inspiration to newcomers in the art of writing programs.

## 2 Original Ginger Model

The original Ginger implementation [11] in the C language followed the classical stack based graph reduction method of translating a Ginger program into a $\lambda$-

expression, converting the latter into a combinator graph and then reducing the graph, as described in [16, 17]. The following summarises the basic mechanism used to simulate parallelism.

The Ginger language primitive, *parallel* subdivides the program into several sub-graphs to be evaluated in parallel. Each of these sub-graphs is reduced by a virtual processor on a single processor machine (or by a real processor in a shared memory multi-processor machine). Each such processor has a *process* allocated to it. A process is an entity that reduces the graph by rewriting a sequence of applications. It may be suspended when the reduction cannot be continued, due to branch nodes being not available yet, by virtue of lazy evaluation.

A process can have the status *Running* (being processed), *Waiting* (ready and waiting for a processor), *Suspended* (waiting for the results needed to continue) or *Dead* (finished processing).

A process has priority *Needed* when one or more processes require the results of its evaluation. Otherwise it has the priority *Possible*. Thus a needed process (i.e. one with priority *Needed*) is one whose result is known to be required.

The process data structure includes two stacks: *RootStack* for the graph of nodes on the spine of the graph being reduced by it, and the other, *ArgStack* to hold the right-branches of those nodes.

At the start of a Ginger program, a new process is allocated to a processor, with no process running on it, and with, as yet, empty *Waiting* and *Suspended* process queues. Such a process, in its turn, may spawn new processes which join the *Waiting* queue, or because of lazy evaluation get suspended thereby joining the *Suspended* queue.

A processor running a part of a program will, normally, have a running process as well as two queues, one of *Waiting* processes, and another of *Suspended* processes. For the *Waiting* queue, the processes with priority *Needed* are kept at the head and those with priority *Possible* at the tail. These queues manage the scheduling as described below.

A running process continues until it completes, thereby attaining the status *Dead* and making the processor free, or it gets suspended (due to the arguments required by it being not available), thereby being moved to the *Suspended* queue, and the processor becoming free, or is preempted because its priority is *Possible* and a process with priority *Needed* has joined the *Waiting* queue. When the processor becomes free, the head of the *Waiting* queue is moved to the processor. Also when a *Suspended* process becomes able to resume because of the arguments on which it is suspended becoming available, it is moved from the *Suspended* queue to the tail of the *Waiting* queue, if priority *Possible*, otherwise to the end of the *Needed* part of the *Waiting* queue.

Parallelism is implemented for a shared memory architecture with the user specifying the number of processors to be used. Ginger uses the construct *parallel* $m$ $x$ $y$ to evaluate $x$ in parallel on processor $m$ returning $y$.

## 3 Java Implementation

### 3.1 A Java Version

As mentioned before implementation in Java was considered for an overall improvement in design, and with the express purpose of exploiting Java threads in dealing with parallel evaluation. However using Java with its ever evolving language specification turned out to be a rather demanding project. Initially, we followed the original C model closely. Towards the end of 1997 a Java implementation of this was completed in time for use by M.Sc. students in Warwick. This provided a valuable testing environment for debugging, and for assessing the performance and robustness of the code. On the whole, this was successful. Java with its higher level features made the conversion relatively easy, and its language directed documentation was also of use. The better structuring of Java made the design more elegant, and the students found the implementation easier to understand.

This then encouraged us to pursue our other objective of dealing with parallel evaluation more cleanly using Java Threads. This involved replacing the parallel process *simulation* of the previous model with Java Language APIs for thread control. We have taken advantage of the recent work on Java Threads [4, 9, 12, 13, 15]. Our threaded solution is described below. As yet the implementation is still incomplete, though almost all the features are in place. In this version, the issue of suspending and unsuspending of the threads (for lazy evaluation) has not been fully resolved. Because of this, it has not yet been possible to deal with the issue of performance profiling. All the development efforts so far have gone into ensuring the correctness of the implementation model, so that its reliability and stability have been our main concern. The question of performance quality will be considered in due course.

### 3.2 A Threaded Solution

We now describe our threaded implementation of Ginger. Instead of the original model which had several processors each with its running process, and queues of *Waiting* and *Suspended* processes, we use a simpler model. For each *parallel* construct a new thread is spawned. A list of threads is maintained. The scheduling of *Waiting* threads is left to the operating system of the machine. Because of this we do not distinguish between *Waiting* and *Running* threads. The currently running thread either terminates, or suspends, or finishes its turn. If it terminates, it notifies the *Suspended* threads. If a *Suspended* thread can now continue, its status is set to *Running*, and it will be scheduled by the operating system.

In our model new threads are spawned as they are needed, using the Java Thread *start()* method. This invokes the *run()* method of the thread class and begins the graph evaluation process. It is possible to terminate a thread using *stop()*, however, it is deprecated in JDK1.2 as it is unsafe. According to JDK1.2

documentation, this is because stopping a thread causes unlocking of all monitors previously locked by it, so that any objects previously protected by these monitors, if in an inconsistent state and damaged, become now visible to other threads, potentially leading to arbitrary behaviour. We have not used it, but have used *join()* to make one thread wait for another thread to terminate. Also we use *setPriority()* to set the priority of a thread when it is started so that it will have the priority needed to run. We ensure that all threads are started with the same high value of 6 (chosen arbitrarily in the range 1 (MIN_PRIORITY) to 10 (MAX_PRIORITY), but above the default value of 5 (NORM_PRIORITY)). Apart from this we have not used it to alter the priority after a thread has been started (see below for a discussion of this). To simplify matters, we use *yield()* to allow the currently running thread to pause and be rescheduled by the operating system, thus allowing another *Waiting* thread with the same priority to run. No other mechanism for prioritising the unsuspending of *Suspended* threads is imposed. Threads evaluating operators whose arguments are yet unknown are suspended using the Java Language Object synchronisation method *wait()*. There is a version of *wait()*, which takes a *long-integer* time argument (in milliseconds) so that indefinite postponement will not occur, the thread unsuspending not later than the time specified by the argument. We have not used it in order to avoid non-determinacy introduced by it. Once the required arguments are evaluated, the *Suspended* thread is unsuspended using the Java *notifyAll()* method, thereby making the operating system schedule it for evaluation. We have not used the alternative version *notify()* of this. This is because *notifyAll()* is safer [13], even though *notify()* will be more efficient. A further discussion of this issue is given below. To allow delays for context switching Java *sleep()* is used. This will allow the effect of another thread's action to take place before the current thread uses the outcome of that action. We have used this very sparingly, again, to minimise the non-determinacy that it is likely to introduce. We have handled lazy evaluation by using these along with Java's synchronisation constructs: *synchronized* for methods and *synchronized()* for blocks.

### 3.3 Some Threading Problems

At any instant there may be zero or more *Suspended* threads, which are unsuspended when they become ready for evaluation. Unfortunately, we have found it difficult to manage the control of such independently running threads by using the above synchronisation constructs. While our solution has passed normal testing procedures, verifying the absence of logical errors in the synchronisation constructs used has proved difficult. Whether Java synchronisation constructs lend themselves to a simplification of the formal verification process is an interesting problem.

The precise moment when a thread starts, stops, changes priority, waits, or awakes cannot be known. A common solution is to allow some time delay using random *long-integer* parameter (milliseconds) for the Thread class method *sleep()*. Some care is needed to avoid *race conditions* [15].

The problem with programming delays using *wait()* or *sleep()* is in not knowing what the exact delay is. In most examples this does not seem to matter, and for optimal performance one might require this delay to be reduced optimally. However, a range of values seems to be possible, and this also depends to a large extent on the application domain. There is no way to determine this range precisely because of system load uncertainty, and again *deadlock*, *indefinite postponement*, and *live-lock* can occur, requiring the implementation programmer to worry about these important issues without any guidelines, or useful debugging aids for handling parallel threads. There seems to be a general dearth of good practice guides, and higher level support in logic and debugging in this application domain of parallel programming. While there exist many good discourses on the general design principles for concurrent programming logic (see for example, [1]), extending such principles to parallel evaluation of graphs has turned out to be far from easy (see for example, [12]). As for general debugging aids for parallel programming, not much has come to our notice. A commonly available debugging tool is Java WorkShop 2.0 (*jws*) [23]. Unfortunately, we have found its threads viewing facility not very helpful for debugging multi-threaded programs. We regard *jws* as a useful sequential program debugging tool. However, its Threads/Stack debug option is rather difficult to use when many interleaved threads are involved. We think it is more of a profiling feature which could be of help in the performance enhancement tuning phase (see below). We really need more help for debugging multi-threading, possibly with graphical displays, and tables that can chart the threads as they progress.

The *yield()* function is a convenient mechanism to make one thread allow another thread to run. However, it is not easy to know which thread is the next one that runs, when there are more than one thread with same priority. We have left the intrinsic operating system in charge of all the scheduling issues, apart from suspending, unsuspending, delays, and occasionally making the current thread to *yield()*.

The current status of Java Threads is rather fluid. Many of the Thread functions of JDK1.1 have been deprecated in the JDK1.2 versions. Thus *suspend()*, *resume()*, which seem rather appropriate for handling lazy evaluation, and many others are no longer recommended for use, requiring the programmer to worry about the problems associated with creating such constructs themselves. While this may make Java more reliable, it also makes it less easy to use. We effectively created our own versions of these methods.

So stability and reliability of the program are difficult to guarantee, with implications on the demands of the domain of the application impinging on the appropriateness of the parallel algorithms being used.

A range of implementations is possible. The implications on the performance and stability of the programs vary over this range. The question of an optimum implementation yielding best performance, reliability and stability is difficult to answer. It may very well be that, no such answer exists. It could be that an implementation may be optimal for a group of applications while far from being so for another.

Java synchronized methods prevent two threads entering a method at the same time, thereby ensuring mutual exclusion criterion of monitors. Correct synchronisation is ensured by locking and unlocking automatically. However it is difficult to design synchronized methods which call other synchronized methods. This gets even more difficult when handling synchronized methods that have to be recursive. Software *patterns* libraries may be of help here [12]. Most of the examples in the current literature of Java Threads applications use very many small synchronized methods of two or three lines. We have found this fine grained structuring rather laborious, and have not managed to construct synchronisation constructs in our programs using exclusively synchronized methods.

Because of the difficulties mentioned above many examples use both *synchronized* methods and *synchronized()* blocks. We have followed this strategy. *Synchronized()* blocks allow a free hand, and are easier to use. However, our experience is that they are also more prone to errors. Unfortunately, in large examples (as in ours), there do not seem to exist a simple way of avoiding them. Some domain independent design heuristics are urgently needed.

Java does not provide *mutex* and *condition-variables* constructs, though they can be easily added as in [13]. Because of this it is not possible to select one of a group of *Suspended* processes to wake up using *notify()* (unless one provides extra scheduling overheads of managing queues of *Suspended* threads and scheduling activation interface for the operating system scheduler). Consequently *notifyAll()* is more commonly used [13], and we have followed this practice. *NotifyAll()* looks at all the *Suspended* processes eventually waking up one of them, while *notify()* just picks one and wakes it up. Because of this, the overheads for parallel evaluation can be very high, reducing performance advantages of parallelism. This cost, should hopefully, be offset by the ever increasing processing power of multiprocessors.

## 4 Results

Test cases were designed to handle the *parallel* constructs in the functions. One interesting feature is that the number of threads used is increased automatically as the number of *Suspended* threads grow. However, the number of reductions per thread remains constant.

So far our test results have been encouraging, even though we have concentrated mainly on standard functions for sorting and sequences using small arguments. Work is continuing to ensure that the implementation passes such tests for all reasonable arguments, and also succeeds in other interesting examples.

Currently, the speed of evaluation is too slow for comfort, but should become faster, as we become more familiar with how the threading mechanism behaves.

While our Ginger interpreter was developed exclusively in a Solaris environment, it was found to work also on a Windows95 machine. However discrepancies showed up on the threaded implementations, which we believe this is due to the differences in the native threads in the two environments [19]. We suspect that this rather disappointing violation of Java portability criterion for threads is a limitation of JDK.

## 5 The Future

All features are already there. However, much testing and debugging still needs to be done. This should improve the quality of the code (e.g. better class structures and Java directed documentation).

Also to be investigated is the impact on the portability of our Ginger interpreter of the differences between Solaris and Windows-32 versions of JDK Java Threads. It is hoped that avoiding the use of specific Thread methods will overcome such differences. This however, would inevitably mean relinquishing the advantages of Solaris Threads [19]. Also the difference in thread scheduling on these platforms needs to be hidden by the software.

While the speed seems comparable with other implementations, much optimisation needs to be done, making use of JDK optimising features. Performance profiles are to be produced for well-established bench-mark test-cases [22] and compared with other similar implementations. One possible help for this could be Solaris TNF (Trace Normal Format) utilities [19].This should improve code quality. and lead to inclusion of enhancements, such as performance monitoring options.

An interesting area of investigation would be to find ways of overcoming non-determinism in the use of Java Threads. This should also prove invaluable in designing efficient algorithms for parallel functional programs.

We would also like to incorporate a GUI front end to Ginger with a view to building a tool for designing and debugging parallel functional programs. A GUI should provide more control because of its extra dimensions in space and texture.

## 6 Conclusion

Our results suggest that using Java Threads for parallel functional programming language implementation is feasible and has the advantage of being clean, thus providing a good insight into the mechanisms involved. Our approach lends itself to an in depth study of the question of improving performance of functional language solutions by introducing parallel processing. This should be of high pedagogic value for students of functional programming.

A major difficulty experienced in implementing parallel features of Ginger was the lack of control of the threads. Debugging using Java WorkShop 2.0 turned out rather ineffective. This required the logic of the concurrency control algorithms used to be sound. While some books on Java Threads have appeared recently, we have found them to be of only limited use. Tools supporting Java Threads are urgently needed, preferably integrated with JDK.

## References


1. Andrews, G., *Concurrent Programming: Principles and Practice*, Benjamin/Cummings, 1991.
2. Arnold, K. and Gosling, J., *The Java Programming Language*, Addison-Wesley, 1996.
3. Augustsson, L. and Johnsson, T., *Parallel graph reduction with the* $< \nu, G >$*-machine* in Functional Programming and Computer Architecture, pages 202–214, ACM, 1989.
4. Chan, P. and Lee, R., *The Java Class Libraries, An Annotated Reference*, Addison-Wesley, 1997.
5. 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.
6. Gosling, J., McGilton, H., *The Java Language Environment – A white Paper*, (`http://java.sun.com/docs/white/`), SunSoft, May 1996.
7. Gosling, J., Yelling, F. and The Java Team, *The Java Application Programming Interface*, Addison-Wesley, 1996.
8. Hammond K., and Peyton Jones S.L., *Profiling Scheduling Strategies on the GRIP Multiprocessor*, in Proceedings of the Fourth International Workshop on Parallel Implementation of Functional Languages, Kuchen H (ed.), Aachen University (September 1992).
9. Hartley, S. J., *Concurrent Programming, The Java Programming Language*, Oxford University Press, 1998.
10. Hudak, P., Peyton-Jones, S. and Wadler, P., eds. Report on the Programming Language Haskell, version 1.2. *ACM SIGPLAN Notices,* **27**(5),1992. For Haskell 1.4, see `http://haskell.cs.yale.edu/report/`
11. Joy, M., *Ginger - A Simple Functional Language*, Research Report 235, Department of Computer Science, University of Warwick, U.K., 1992. (`http://www.dcs.warwick.ac.uk/pub/reports/rr/235.html`)
12. Lea, D., *Concurrent Programming in Java, Design Principles and Patterns*, Addison-Wesley, 1997.
13. Lewis, B., *MultiThreaded Programming*, Lambda Computer Science (`http://www.LambdaCS.com/`), 1998.
14. 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.
15. Oaks, S. and Wong, H., *JAVA Threads*, O'Reilly, 1997.
16. Peyton-Jones, S., *The Implementation of Functional Programming Languages.* Prentice-Hall, 1987.
17. Peyton-Jones, S. and Lester, D., *Implementing Functional Languages.* Prentice-Hall, 1992.
18. 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, 1991.
19. *Java*$^{TM}$ *On Solaris*$^{TM}$ *2.6 – A White Paper*, SunSoft, Part No.: 805-3084-01, `http://www.sun.com/solaris/java/wp-java/`, September 1997.
20. 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).
21. Turner, D., Miranda: a Non-strict Functional Language with Polymorphic Types. In Jouannaud J.P., ed., *Functional Programming Languages and Computer Architecture.* Springer-Verlag, 1985.
22. Wakeling, D., VSD: A Haskell to Java Virtual Machine Code Compiler. In *Proceedings of the 9th International Workshop on the Implementation of Functional Languages*, pages 71-84, September 1997.
23. Weaver, L., *Inside JAVA WORKSHOP 2.0*, Prentice-Hall, 1998.