Chapter 1

MultiCore Parallelisation for Hume

Abdallah Al Zain[1], Greg Michaelson[2], Kevin Hammond[3]
Category: Research Paper

Abstract: With the emergence of commodity multicore architectures, exploiting tightly-coupled parallelism has become increasingly important. Hume is a novel formally-motivated programming language oriented to developing software where strong assurance of resource use is paramount, in particular embedded architecture. Functional programming languages, such as Hume, are, in principle, well placed to take advantage of this trend, offering the ability to easily identify threads for parallelism in multicore architectures. Unfortunately, obtaining real performance benefits has often proved hard to realise in practice. This paper outlines the design and implementation of Hume for multicore architectures. It presents preliminary results which suggest that there is strong potential for seamless parallel gains in Hume programs. A key aspect of our approach is the use of a new and completely lock-free communication mechanism. Using this mechanism, we can obtain good parallel performance for suitable Hume programs, of up to 6.8 on eight cores.

1.1 INTRODUCTION

Hume [18] is a contemporary programming language based on concurrent finite state automata controlled by transitions expressed in a rich polymorphic functional language. Hume has been designed to encompass different degrees of static analysis precision for different degrees of expressiveness, in particular for determining resource bounds, in particular for time and space. Key to the Hume design is the concept of concurrent *boxes*, interacting through *wires* that link them both to

[1]Heriot-Watt University, Edinburgh, UK; a.d.alzain@hw.ac.uk
[2]Heriot-Watt University, Edinburgh, UK; greg@macs.hw.ac.uk
[3]University of St Andrews, St Andrews, UK; kh@cs.st-and.ac.uk

each other and to the external environment. Boxes offer natural loci for software design, implementation and analysis.

Hume is based on strong semantic underpinnings and is supported by a mature tool-chain centred around the Hume Abstract Machine (HAM). This provides a stable, shared abstraction for both implementation and analysis. The HAM interpreter (HAMi) directly excecutes HAM code, the Hume compiler (HAMc) translates HAM code to native code through C, and the resource analysers relate statically-inferred properties of HAM to precise instrumentation of equivalent native code on designated hardware platforms.

Current Hume implementations and analyses are oriented to programs executed sequentially on a single processor. As part of the UK EPSRC Islay project, we are exploring the deployment of Hume on heterogeneous hardware platforms that offer some combination of multicore, SIMD and/or FPGA support. Our objective is to make optimal use of these parallel processing resources under the guidance of high-quality cost analysis. In this paper, we present some preliminary results based on adapting the HAM interpreter to support multicore systems.

1.2 THE POTENTIAL OF HUME FOR MULTICORE ARCHITECTURES

Almost all CPUs that are used in desktop or server systems now contain multiple cores, and many laptops also have two cores. Although it offers advantages in terms of scaling and power usage, multicore technology is, at heart, just a contemporary realisation of the well-known shared-memory multiprocessor paradigm that has been popular since the 1980s, with similar properties and problems.

Structurally, each core usually has a fairly small, private level 1 cache and shares higher levels of cache and the global memory with other cores. True dual-core CPUs, both realised from the same die, are common (e.g. Intel's Core 2 Duo). Four-core (quad-core) and eight-core CPUs are becoming more widely available, but are still often formed from specially-packaged dual-core chips. They thus contain a hidden memory hierarchy that may have a significant impact on performance (Intel's recent Nehalem, or Core i7, architecture is an exception, with four or eight cores on a single chip, sharing on-chip level 3 cache and memory controller hardware). Future designs are expected to increase the number of cores (perhaps scaling back on the capabilities of individual cores), leading ultimately to *many-core* designs with hundreds or thousands of cores in a single package. An example is Intel's forthcoming Larrabee *general-purpose graphics-processing unit* (GPGPU), which represents a cross-over between multicore and conventional graphics-processing (GPU) designs, and which targets high-performance computing as well as computer graphics. Each Larrabee chip will possess 32 or more small, simple, but hyper-threaded 1GHz cores.

Basic support for multicore programming, e.g. for controlling the placement, execution and memory access rights of processes or threads, is available at a variety of levels. For example, the Intel Thread Library [24] supports multicore specifically on Intel CPUs; the generic Posix Threads (pthreads) Library [21] supports multicore for Posix compliant operating systems; and OpenMP [10] of-

fers high level, language and operating-system independent compiler directives. Nonetheless, as with general parallel programming, multicore programming remains a black art. A deep understanding of algorithm, platform and support software characteristics is required to exploit a multicore system effectively. The key problem, as with all shared-memory multiprocessing, is maximising memory locality and minimising the volume of accesses to shared memory.

It is our contention that Hume offers considerable potential for multicore exploitation because Hume boxes are *the right size* for current and future multicore designs. Firstly, the Hume execution model is premised on a high degree of locality of memory for individual box execution, offering a high degree of potential parallelism. Secondly, the super step scheduling approach that we use (Section 1.3 guarantees predictable and hence analysable patterns of global memory use: all global memory changes (corresponding to wire modifications) are resolved as part of an atomic *super step*. Finally, boxes are considerably coarser-grained than lightweight threads in typical implicitly parallel functional languages such as GpH [22, 17] or Eden [5], and are easily mapped to operating-system threads. This reduces the number of threads that are produced for a typical Hume program, meaning that there is a straightforward mapping to the small number of cores in current multicore designs.

1.3 HUME SUPER-STEP SCHEDULING

At its simplest, a Hume box consists of a set of *input* wires, a set of *output* wires and a set of *matches*, where each match associates a *pattern* over the inputs with an *expression* over the outputs. As noted, a Hume program then consists of one or more Hume boxes linked to each other, and to the external environment, by wires. The wires thus effectively constitute a shared memory channel between two boxes. Each box may also have additional local memory for inputs, working store, and buffered outputs.

It is important to note that an individual Hume box effectively constitutes an autonomous program that runs continuously, repeatedly matching and consuming inputs to generate new outputs. The Hume semantics specifies a very abstract model of such execution based on a "super-step" scheduling model, which divides program execution into a series of scheduling cycles. Each and every box in the program may be run at most once during each scheduling cycle.

At the start of each super-step scheduling cycle, all boxes are checked to determine whether they are *RUNNABLE*. A box is *RUNNABLE* if it has sufficient inputs to match one of its rules, and it is not currently blocked producing outputs that have not yet been consumed by some other box (in the latter case, it is in a *BLOCKED OUTPUT* state). During the first phase of each scheduling cycle, every *RUNNABLE* box satisfies one of its matches, working on local copies of its input values to generate locally-buffered output values. It thus ends this phase in the *BLOCKED OUTPUT* state. Note that the order in which boxes are executed in a super-step is arbitrary and immaterial, and that boxes are total, meaning that any *RUNNABLE* box must, by definition, satisfy one of its matches, and be executed

during the super-step.

When every box has completed this phase, input wire consumption and output wire instantiation are next resolved globally. First of all, the input values matched by new *BLOCKED OUTPUT* boxes are removed from the corresponding wires. Then, for each *BLOCKED OUTPUT* box, if all its output wires are empty then they are set to the new buffered output values and the box may become *RUNNABLE* in the next cycle. A *BLOCKED OUTPUT* box with one or more non-empty output wire will, however, remain in the *BLOCKED OUTPUT* state.

A useful exception to this tight-stepped scheduling process can be made by distinguishing a *SELF OUTPUT* state, where a box generates outputs solely for its own consumption [16]. So long as a box is *SELF OUTPUT*, it may execute repeatedly without the need for super-step wire resolution. Conversely, not distinguishing such *SELF OUTPUT* boxes may result in other boxes being needlessly scheduled without state change, pending some *SELF OUTPUT* box consuming their inputs or generating their required outputs.

1.4 A MULTICORE REALISATION OF HUME

We have chosen to implement a straightforward threading model for Hume that directly exploits the existing Hume tool chain to map boxes to operating-system threads. In the longer term, parallelism could be introduced through, for example, automatic program analyses, semi-explicit program anotations or multicore-oriented box skeletons

The Hume compiler produces highly-efficient but opaque C code, where the code for individual boxes is combined into a single monolithic C program. Since this would be difficult to modify, we have instead focused on modifying the HAM interpreter, which is cleanly structured with clearly discernable stages corresponding to the generic Hume execution semantics, and which is reasonably efficient (about 4-10 times the speed of the standard JVM on similar code). Finally, for our initial experiments, we decided to deploy the generic OpenMP library to annotate salient features in the HAMi: OpenMP combines a high degree of abstraction, high level of portability and excellent compiler support.

Our first step towards a multicore implementation of Hume was to implement a static scheduler for HAMi based on a classical fork-join model of parallel execution. Using this approach, at the start of each super-step cycle, the Hume scheduler spawns one thread for each box, whether or not it is runnable. Subsequently, at the end of the cycle, these threads all re-join the main thread, as shown in Figure 1.1 (above). Each thread is assigned to a core. In GHC's terminology [**?**], these threads are therefore *capabilities*. Since our static scheduler showed poor performance in practice, we consequently decided to optimise the Hume scheduler so that in any given cycle, it only executed runnable threads, as shown in Figure 1.1 (below). In more detail, the Hume scheduler is declared to be a parallel region using the appropriate OpenMP preprocessor directive. This creates a team of worker threads, one for each core. Runnable boxes are then assigned to

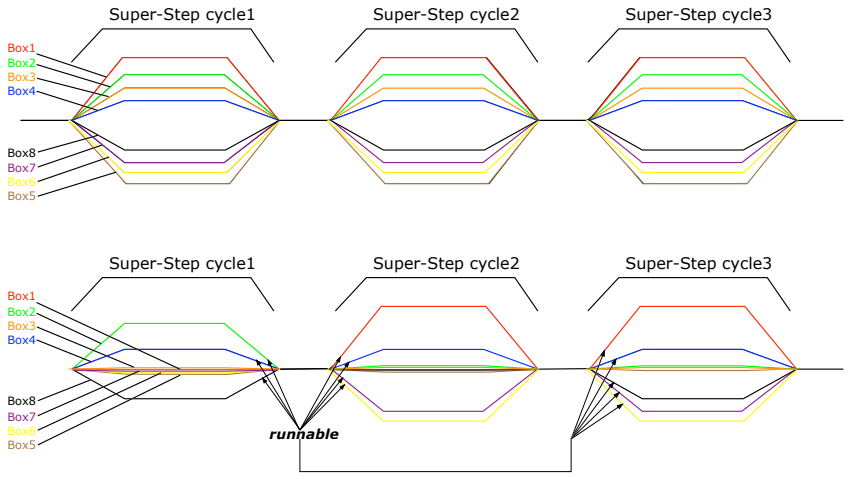

FIGURE 1.1. Static (above) and dynamic (below) Hume schedulers for multicore

one of these threads using the OpenMP `task` construct. Since Hume boxes are independent computational units and share memory only a point-to-point basis through explicit wires, they can be assigned private unshared memory areas using the OpenMP `threadprivate` directive.

Because all communication between boxes is through shared wires that are either all written to or all read from at a specific point in the super-step scheduling cycle, it is possible to achieve *completely lock-free communication* between boxes. There is no need to synchronise two communicating boxes: since any output written during one scheduling cycle will never be read before the subsequent cycle, and since the subsequent cycle will not be scheduled before all output is completed, it follows that a thread can never start to read data before it has been completely output. This design also reduces memory hotspots: each thread needs to synchronise only once in each cycle – with the scheduler – no matter how much communication it performs.

1.5 EXAMPLE

Our testbed example counts `Fibonacci(19)` 20000 times in ten boxes and then merges the result data to `stderr`. The program consists of ten boxes calculating Fibonacci and a single `merge` box that collates the results from each of the ten boxes using a fair merge strategy and directs these results to the `stderr` channel. The `fs`*x* boxes receive an initial value of 20000 and calculate Fibonacci for some fixed constant `N` using the `fib` function. For our experiment, we have replicated the `fib` function to avoid problems of shared-memory access that can happen as a result of code in the original sequential `HAMi` implementation.

merging count Fibonacci

```
type Int = int 32;

fib0 :: Int -> Int;
fib0 0 = 1;
fib0 1 = 1;
fib0 n =
   fib0(n-2) +
   fib0(n-1);


fib1 :: Int -> Int;
fib1 0 = 1;
fib1 1 = 1;
fib1 n =
   fib1(n-2) +
   fib1(n-1);
  ...
  ...
fib9 :: Int -> Int;
fib9 0 = 1;
fib9 1 = 1;
fib9 n =
   fib9(n-2) +
   fib9(n-1);

N = 19;

box fs0
in (n:: Int)
out(n', r:: Int)
match
  0 -> (*,*)
| n -> (n-1, fib0 N);

box fs1
in (n:: Int)
out(n', r:: Int)
match
  0 -> (*,*)
| n -> (n-1, fib1 N);
   ...
   ...
box fs8
in (n:: Int)
out(n', r:: Int)
match
  0 -> (*,*)
| n -> (n-1, fib8 N);

box fs9
in (n:: Int)
out(n', r:: Int)
match
  0 -> (*,*)
| n -> (n-1, fib9 N);

box merge
in (x0,x1,x2,x3,x4,x5,x6,
  x7,x8,x9:: Int)
out (z :: Int)
fair
    (x,*,*,*,*,*,*,*,*,*) -> x
|   (*,x,*,*,*,*,*,*,*,*) -> x
|   (*,*,x,*,*,*,*,*,*,*) -> x
|   (*,*,*,x,*,*,*,*,*,*) -> x
|   (*,*,*,*,x,*,*,*,*,*) -> x
|   (*,*,*,*,*,x,*,*,*,*) -> x
|   (*,*,*,*,*,*,x,*,*,*) -> x
|   (*,*,*,*,*,*,*,x,*,*) -> x
|   (*,*,*,*,*,*,*,*,x,*) -> x
|   (*,*,*,*,*,*,*,*,*,x) -> x;


wire merge(fs0.r, fs1.r, fs2.r,
 fs3.r, fs4.r, fs5.r, fs6.r,
 fs7.r, fs8.r, fs9.r)(output);
wire fs0(fs0.n'
  initially 20000)
  (fs0.n,merge.x0);
wire fs1(fs1.n'
  initially 20000)
  (fs1.n,merge.x1);
    ...
    ...
wire fs8(fs8.n'
  initially 20000)
  (fs8.n,merge.x8);
wire fs9(fs9.n'
  initially 20000)
  (fs9.n,merge.x9);
```

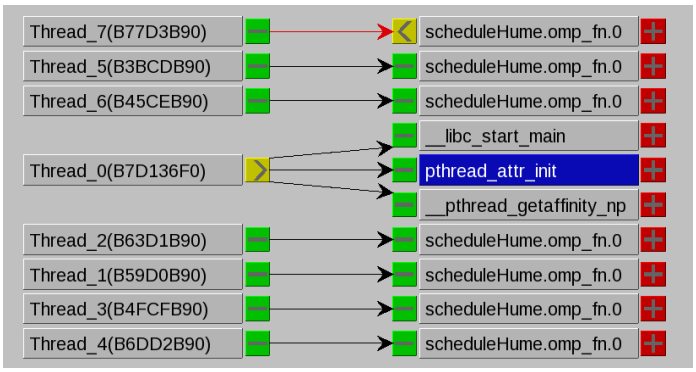

FIGURE 1.2. Per Thread Call Graph

1.6 RESULTS

Our measurements have been performed on an eight-core Dell PowerEdge 2950 machine located at Heriot-Watt University (*lxpara3*). This machine is constructed from two quad-core Intel Xeon 5410 processors running at 2.33GHz (as discussed earlier, each quad-core is itself packaged from two dual-core chips). *lxpara3* has a 1333MHz front-side bus, and 33GB of fully-buffered 667MHz DIMMs (of which the Hume implementation uses only 1-2MB). It runs Ubuntu Linux 4.3 (kernel version 2.6.27-7). Our code has been compiled using the Intel C Compiler (`icc`) version 11.0. Figure 1.2 is a visual presentation of the program execution, which gives a profile for the computation as a whole, and also displays the critical functions and call sequences. The root node in each chain of nodes represents a thread, and the remaining nodes show the dynamically-changing call graph. Our computation uses eight threads. The first thread (`Thread_0`) runs posix threads and starts up C libraries. The remaining seven threads run the Hume scheduler (the main parallel part of the implementation). The colour version of the profile shows the *critical path* (the path with the maximum edge time) in red. Here, the critical path represents the single threaded scheduler for the non-runnable boxes in the Hume computation.

Figure 1.3 is an activity profile showing all the Hume threads. The thread and process names, plus event samples, are shown on the left; the right is shows a horizontal bar chart showing each thread's activity. In our computation, there is only one active running process (`hami`, the Hume abstract machine interpreter). However, we have eight threads within this process. The event samples report the total count of events for each thread. In `hami` these events represent the activities that each box performs during the computation. As discussed earlier, only runnable boxes are evaluated in parallel while the main thread performs book-keeping for all other threads. The main thread (`thread4`) therefore counts more events than any other thread. The horizontal axis depicts the total number of events monitored over a period of time. The colour of the bar for each thread reflects the number of

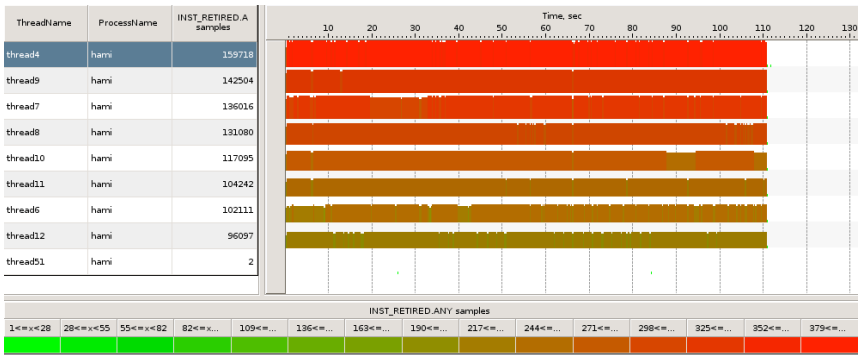

FIGURE 1.3. per-core Sampling Activity Profile

no of Threads	Runtime (Sec)	Speedup
1	782	1
2	460	1.7
3	300	2.6
4	205	3.8
5	166	4.7
6	147	5.3
7	130	6.0
8	115	6.8

TABLE 1.1. Parallel performance of merging Fibonacci

boxes that are under evaluation at each point in time within the specific thread.

Table 1.1 shows the performance of the merging Fibonacci function described above. The first column shows the number of threads that are involved in the computation; the second and third columns show the runtime and speedup. We can see that we achieve good speedup of up to a factor of 6.8 on eight cores. However, there is clear tail-off in performance above 5 cores. We have not investigated this in detail, but it may reflect either the costs of synchronisation of each worker thread with the main scheduler thread, or simply the costs of I/O to the standard error channel.

1.7 RELATED WORK

The recent trend towards multicore architectures has sparked a significant amount of new work that is aimed at exploring novel programming models and runtime systems for such architectures (e.g. [19, 8, 7, 14, 15, 20, 2, 9, **?**]). This work has exploited a number of different approaches. These include:

- *Parallel libraries*, such as *Pthreads* [21] and Phoenix [23], provide the pro-

grammer with the ability to express parallelism directly. In fact, Phoenix is an implementation of Google's MapReduce skeleton for shared-memory systems. The Phoenix runtime system is implemented on top of the Pthreads library and automatically manages thread creation, dynamic task scheduling, and data partitioning.

More advanced libraries, such as Cilk [14] or OpenMP [10], provide higher level parallelisation primitives, including, for example, support for nested parallelism [11]. However, according to Bridges [6], typical library approaches provide little support to help achieve correct or effective parallelism. For instance, the Cilk `inlet` directive is similar to its `Commutative` directive, in ensuring correct execution of code in a non-deterministic fashion. However, `inlet` is meant to serially update state upon return from a spawned function, while `Commutative` is meant to facilitate parallelism by removing serialisation. Clearly, it would be easy to use the wrong construct.

- *Message-passing approaches*, such as our GUM implementation of Haskell for multicore machines [4], or Erlang [3, 12]. Our approach to programming multicore systems in Haskell uses algorithmic skeletons to introduce parallelism that is then mapped to multicore threads executing sequential program components. The Erlang approach is based on creating explicit threads which are mapped directly to operating system threads. In GUM, as with the Hume implementation we have described here, all communication and thread synchronisation is implicit, whereas Erlang requires the programmer to explicitly use message-passing primitives. The key difference between our GUM work and that described in this paper is that the latter makes direct use of shared memory mechanisms. Moreover our approach of treating Hume boxes as units of execution makes it straightforward to derive a parallel solution that can easily be mapped to multiple cores.

- *Explicit memory transactions* [8, 20, 1] attempt to reduce locking by exposing parallel operations as transactions. Some such approaches require an explicit step to make locations or objects part of a transaction, while other approaches make the memory operation behaviour implicit. Implicit transactions require either compiler or hardware support. Both of those techniques have been proposed to help the programmer express parallelism in an easier manner.

  While such approaches appear promising on paper, they have so far generally failed deliver good performance. For example, Harris, Marlow and Peyton Jones [20] report poor and highly variable parallel performance, using memory transaction techniques on shared-memory machines. . In contrast, we have demonstrated a speedup of 6.8 on our eight-core testbed machine.

- *Data-parallel* approaches where parallelism is exposed by evaluating elements of bulk data structures in parallel. For example, Data-Parallel Haskell [9] provides parallel arrays and special parallel operations to handle them. Good results are reported for typical data-parallel problems (such as sparse matrix multiplication). A similar approach is taken by Fluet *et al.* [13] who embed

nested data-parallel constructs into an explicitly parallel Concurrent ML setting.

While these approaches can deliver good performance for appropriate applications, they are, however, primarily suited to static, regular, data-driven parallelism rather than the more dynamic, irregular forms that can be programmed using our Hume box approach.

In contrast to most of the approaches described above, we focus on exploiting parallelism in multicore architecture by introducing boxes as the *right-size construct* for mapping to cores.

1.8 CONCLUSIONS

We have shown that by treating boxes as loci of parallelism in Hume, we can gain valuable and consistent speedup in a multicore environment for one suitable testbed application. Furthermore, this was achieved at very little software engineering effort, with the addition of just four **OpenMP** directives into the existing C source code for the sequential HAMi abstract machine interpreter. A key novelty of our approach is the use of completely lock-free communication through a novel super-step scheduling mechanism. This has contributed to the good performance result that we have achieved.

It is important, however, to appreciate that, while this result is encouraging, it is still preliminary, since it is based on a perfectly-balanced and very simple example. For arbitrary realistic Hume programs it is entirely possible that we will achieve poorer results: different boxes will have widely different individual processing characteristics but the synchronised wire resolution in the super-step semantics means that overall program performance depends on that of the slowest box.

Nonetheless, we have now increased our confidence that Hume boxes are good places to seek performance improvements through parallelism. In future research we intend to:

- modify the run-time scheme to prioritise *RUNNABLE* boxes;
- analyse programs to identify and hence prioritise *SELF OUTPUT* boxes;
- deploy the Hume worst-case execution time analysis to identify groups of boxes that might be excuted together on individual cores in order to optimise overall load balance.

In the longer term, we will also explore building thread box skeletons which can be explicitly called in Hume programs, with the aim of eventually modifying the native code compiler to generate threaded code for nominated or analysis identified boxes.

ACKNOWLEDGEMENTS

This research is supported by the UK EPSRC Islay project (EP/F030592, EP/F030657, EP/F03072X, and EP/F031017) "Adaptive Hardware Systems with Novel Algorithmic Design and Guaranteed Resource Bounds".

REFERENCES

[1] A-R Adl-Tabatabai, B.T. Lewis, V. Menon, B.R. Murphy, B. Saha, and T. Shpeisman. Compiler and Runtime Support for Efficient Software Transactional Memory. In *PLDI '06: Proceedings of the 2006 ACM SIGPLAN conference on Programming language design and implementation*, pages 26–37, New York, NY, USA, 2006. ACM.

[2] C.K. Anand and W. Kahl. A Domain-Specific Language for the Generation of Optimized SIMD-Parallel Assembly Code. SQRL Report 43, Software Quality Research Laboratory, McMaster University, May 2007. available from http://sqrl.mcmaster.ca/sqrl_reports.html.

[3] J. Armstrong, R. Virding, C. Wikström, and M. Williams. *Concurrent Programming in* ERLANG. Prentice Hall, 2nd edition, 1996.

[4] J. Berthold, S. Marlow, A. Al Zain, and K. Hammond. Comparing and Optimising Parallel Haskell Implementation. In Sven-Bodo Scholz, editor, *IFL'08 — Implementation and Application of Functional Languages 20th International Symposium*, Draft Proceedings, pages 223–240, Hetfield, Hertfordshire, UK, September 2008. Technical Report No. 474.

[5] S. Breitinger, R. Loogen, Y. Ortega Malln, and R. Pea Marí. Eden — The Paradise of Functional Concurrent Programming. In *EuroPar'96 — European Conf. on Parallel Processing*, LNCS 1123, pages 710–713, Lyon, France, 1996. Springer.

[6] M. Bridges, N. Vachharajani, Y. Zhang, T. Jablin, and D. August. Revisiting the Sequential Programming Model for Multi-Core. In *MICRO '07: Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 69–84, Washington, DC, USA, 2007. IEEE Computer Society.

[7] Broadcom Corp. BCM1250 Multiprocessor. Technical report, Broadcom Corporation, April 2002.

[8] B. Carlstrom, A. McDonald, H. Chafi, J. Chung, C. Cao Minh, C. Kozyrakis, and K. Olukotun. The atomos transactional programming language. In *ACM SIGPLAN 2006 Conference on Programming Language Design and Implementation*. Jun 2006.

[9] M.M.T. Chakravarty, R. Leshchinskiy, S.L. Peyton Jones, G. Keller, and S. Marlow. Data Parallel Haskell: a Status Report. In *DAMP'07: Workshop on Declarative Aspects of Multicore Programming)*, Nice, France, 2007. ACM Press.

[10] B. Chapman, G. Jost, and R. van der Pas. *Using OpenMP: Portable Shared Memory Parallel Programming (Scientific and Engineering Computation)*. The MIT Press, 2007.

[11] A. Duran, M. Gonzàlez, and J. Corbalán. Automatic Thread Distribution for Nested Parallelism in OpenMP. In *19th Annual International Conference on Supercomputing (ICS '05)*, pages 121–130, New York, NY, USA, 2005. ACM.

[12] Ericsson Utvecklings AB. *Erlang Home Page*.

[13] M. Fluet, M. Rainey, J. Reppy, A. Shaw, and Y. Xiao. Manticore: A Heterogeneous Parallel Language. In *DAMP'07: Workshop on Declarative Aspects of Multicore Programming*, Nice, France, 2007.

[14] Matteo Frigo, Charles E. Leiserson, and Keith H. Randall. The Implementation of the Cilk-5 Multithreaded Language. In *PLDI'98 — Conf. on Programming Language, Design and Implementation*, volume 33 of *ACM SIGPLAN Notices*, pages 212–223. ACM Press, 1998.

[15] M.I. Gordon, W. Thies, M. Karczmarek, J. Lin, A.S. Meli, A.A. Lamb, C. Leger, J. Wong, H. Hoffmann, D. Maze, and S. Amarasinghe. A Stream Compiler for Communication-Exposed Architectures. In *ASPLOS-X: 10th international Conference on Architectural Support for Programming Languages and Operating Systems*, pages 291–303, New York, NY, USA, 2002. ACM.

[16] G. Grov, G. Michaelson, and A. Ireland. Formal Verification of Concurrent Scheduling Strategies using TLA. In *3rd IEEE International Workshop on Scheduling and Resource Management for Parallel and Distributed Systems*, number CFP07036-USB in IEEE Catalog Number. IEEE, 2007.

[17] K. Hammond, J.S. Mattson Jr., A.S Partridge, S.L. Peyton Jones, and P.W. Trinder. GUM: a Portable Parallel Implementation of Haskell. In *IFL'95 — Intl Workshop on the Parallel Implementation of Functional Languages*, September 1995.

[18] K. Hammond and G. Michaelson. Hume: a Domain-Specific Language for Real-Time Embedded Systems. In *In Proc. of GPCE'03*, pages 37–56. LNCS 2830, Sep. 2003.

[19] T. Harris and S. Singh. Feedback Directed Implicit Parallelism. In *ICFP '07: 2007 ACM SIGPLAN International Conference on Functional Programming*, pages 251–264, New York, NY, USA, 2007. ACM.

[20] Tim Harris, Simon Marlow, and Simon Peyton Jones. Haskell on a Shared-Memory Multiprocessor. In *Haskell '05: Proceedings of the 2005 ACM SIGPLAN workshop on Haskell*, pages 49–61. ACM Press, September 2005.

[21] B. Lewis and D.J. Berg. *Multithreaded Programming with Pthreads*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1998.

[22] H-W. Loidl. Glasgow Parallel Haskell. WWW page, January 2001. <URL:`http://www.macs.hw.ac.uk/~dsg/gph/`>.

[23] C Ranger, R. Raghuraman, A. Penmetsa, G.R. Bradski, and C. Kozyrakis. Evaluating MapReduce for Multi-core and Multiprocessor Systems. In *HPCA '07: 2007 IEEE 13th International Symposium on High Performance Computer Architecture*, pages 13–24. IEEE Computer Society, 2007.

[24] J. Reinders. *Intel Threading Building Blocks: Outfitting C++ for Multi-core Processor Parallelism*. O'REILLY, 2007.