Empirically Investigating Parallel Programming Paradigms: A Null Result

Meredydd Luff
University of Cambridge
Meredydd.Luff@cl.cam.ac.uk

Abstract

The dominant paradigm of concurrent programming has well-publicized usability problems, but the alternatives have not been well analyzed from a usability perspective. I attempted an empirical comparison of programmer productivity using the Actor model, transactional memory, and traditional lock-based concurrency paradigms. The results were inconclusive. I discuss my experiment, present its results, and discuss possible reasons why such experiments are a blunt tool with which to investigate programming language usability.

Keywords Empirical study, Parallel programming

1. Introduction

With the widespread arrival of multi-core processors, it is becoming necessary to write parallel programs to fully exploit nearly any modern computer.

However, parallel programming has proven extremely difficult. In particular, the dominant paradigm – that of multiple threads sharing writable memory, and controlling access to it with mutual-exclusion locks – has come in for much criticism for its usability failings ([Lee 2006], among many others).

The design of concurrency mechanisms for programming languages is a serious and painfully unresolved problem in HCI. The adoption of multi-core processors is leading us to put ever more weight on an interface widely regarded as inadequate. Arguably, then, this one of the most important problems in the usability of programming systems.

PLATEAU '09 Workshop at Onward! 26 October 2009. Orlando, Florida, USA.
ACM .

1.1 Contributions

- I describe an experiment to directly measure and compare programmer performance using different parallel paradigms to solve a problem. I lay out some background in Section 2, then describe and justify my methods in Section 3.
- I present the results of this experiment in Section 4, and show them to be inconclusive.
- I discuss possible causes of this inconclusive result, and the difficulty of using controlled empirical experiments to evaluate programmer productivity, in Section 5.

2. Background

2.1 Alternative paradigms

Language designers have proposed many alternatives to the dominant paradigm of lock-based concurrency [Skillicorn and Talia 1996]. I set out to investigate two paradigms that have gained some popularity, and claim superior usability to the current *de facto* standard.

2.1.1 Message Passing

Message-passing systems provide a private memory for each thread of control. All communication is by discrete messages passed to other threads, which process them one at a time. Examples include Hoare's Communicating Sequential Processes [Hoare 1978], or the more dynamic Actor model [Hewitt et al. 1973] most popularly associated with the Erlang programming language [Erl].

2.1.2 Transactional Memory

Transactional memory systems have threads of control sharing a single memory space. Their memory accesses are grouped into transactions, and a run-time system detects and rolls back conflicts to ensure that each transaction occurs atomically or not at all. [Peyton-Jones 2007]

Transactional memory is quite similar to classic threading and locking. The difference between manual locking and transactional memory can be compared to the difference between manual and garbage-collected memory management [Grossman 2007].

2.2 Evaluation

Like most programming language features, concurrency paradigms have historically been built on a hunch, and evaluated by anecdote and holy war. Even when parallel programming systems *are* evaluated for usability, researchers compare whole languages and runtime systems rather than the principles they embody (see Related Work in Section 6). This makes their results less informative for the design of new programming systems.

I chose to evaluate these paradigms empirically with a controlled experiment. In this experiment, subjects solved the same problem, in the same language, varying only the concurrency paradigm.

3. Materials and Methods

3.1 Experiment structure

The goal of this experiment was to compare programmer performance using different parallel paradigms, keeping the programming language and environment constant.

I tested three parallel paradigms: the Actor model, transactional memory, and standard shared-memory threading with locks (henceforth "SMTL"). I also tested subjects on a sequential control condition, as a positive control: it is generally agreed that sequential programming is substantially easier than SMTL, and any acceptably powerful study should show this effect clearly.

I provided all four programming models for the Java programming language. Java is a widely adopted language, taught in the Cambridge undergraduate curriculum, and provides an uncontroversial baseline for this experiment.

Each subject solved the same problem twice: once with the standard (SMTL) Java threading model, and once with Actors, transactional memory, or no parallelism at all (the sequential condition). The whole session took approximately 4 hours per subject. Scheduling was balanced, with half of the subjects solving the SMTL condition first, and the other half solving it second.

Each subject filled out a questionnaire before the experiment, to assess their level of experience and self-perception of skill. After each task, the subject filled out a questionnaire indicating the level of subjective difficulty and opinions of the concurrency model used.

During the experiment, subjects' screens were recorded, and a webcam recorded the subject to monitor off-computer events. A snapshot of the project directory was taken at 1-minute intervals, and instrumented tools logged each invocation of the compiler or run of the resulting program.

3.2 Subjects

Seventeen subjects were recruited from the undergraduate (10) and graduate (7) student population of the Cambridge Computer Laboratory. Of these, eleven successfully completed both tasks.

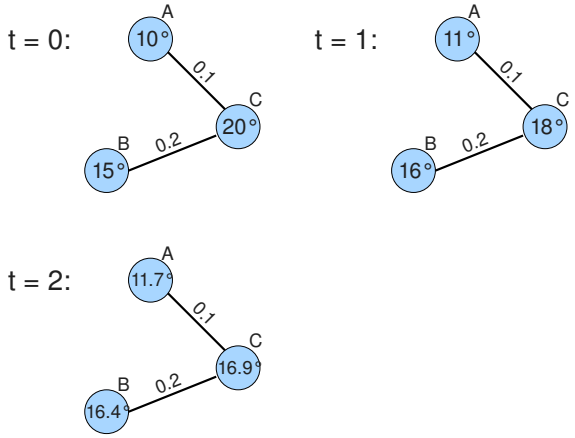
3.3 Task

I chose to minimize variability, at the expense of significant learning effects, by using a single problem. This was an "unstructured grid" problem, in the classification of the View from Berkeley [Asanovic et al. 2006].

I presented a toy physics problem, modelling heat flow between identical blobs, connected by rods of varying conductivity. If the temperature of blob i at time t is represented as T_t^i, the temperature change due to a rod of conductivity k connecting blobs a and b is:

$$T_{n+1}^a = T_n^a + k(T_n^b - T_n^a)$$

A graphical example is shown below:



To reduce the time subjects spent writing irrelevant I/O code, I provided them with code to load a sample data set into memory and pass it as arguments to a function call. I provided the inputs in a deliberately un-useful data structure, and the written instructions instructed each subject to translate the data into their own choice of structure. The subject was instructed to translate the solution back to the original un-useful format and call a function to check its correctness.

3.4 Implementing Different Paradigms

I aimed to replicate the user experience of programming in each paradigm, while keeping the underlying language as close as possible to idiomatic Java.

This involved trade-offs which made performance or scalability comparisons between different conditions impractical. This is a disadvantage, as scalability is the ultimate goal of all such parallel programming, but other studies have examined the scalability characteristics of different concurrency mechanisms, and I considered usability the more important target for the present study.

3.4.1 SMTL and sequential

Java's native concurrency model is based upon threads and mutual-exclusion locks, so standard Java was used for the SMTL and sequential conditions.

3.4.2 Transactional Memory

For the transactional memory condition, I used an existing transactional memory system for Java, Deuce [Korland et al. 2009]. Deuce modifies Java bytecode during loading, transforming methods annotated with `@Atomic` into transactions. For example:

```
class TransactionalCounter {
  private int x;

  public @Atomic increment() {
    x = x + 2;
  }
}
```

However, Deuce does not instrument array accesses, or classes in the Java standard library, so it cannot be used with idiomatic Java. Instead, I enabled Deuce's "single global lock" mode, which makes all atomic methods mutually exclusive. This preserves the semantics of transactional memory, but prevents us from evaluating performance.

3.4.3 Actor Model

Implementing the actor model for Java presented a challenge. Java assumes mutable shared memory throughout its design, whereas actors require disjoint memories and messages which the sender cannot change after they are sent.

One of the touted advantages of the actor model is that the system enforces actor isolation, preventing the user from making a class of mistakes. Enforced isolation is therefore necessary to realistically model the desired user experience.

I considered Kilim [Srinivasan and Mycroft 2008], an implementation of the Actor model in Java with an annotation-based type system to enforce actor isolation. Kilim enforces a complete transfer of ownership of mutable objects sent in messages, so that only one actor can refer to a mutable object at any one time. However, this is a substantial departure from idiomatic Java. In addition, no version of Kilim including this type system is publicly available, and the author warned that his pre-release version was unreliable. I therefore concluded that Kilim was not suitable for this experiment.

I also considered using a run- or compile-time system to ensure that only immutable objects – objects whose fields are all `final`, and members are similarly immutable – could be passed in messages. However, Java's standard library is built with mutable objects. I wished to evaluate "Java with Actors", not "Java with Actors and all standard data structures removed", and so this option was also rejected.

Instead, I implemented a reflection-based runtime system, using deep copies for isolation. In my implementation, each actor is represented by an object, to which no reference is held by any other actor. Other actors interact only with a wrapper class, `Actor`, through which they can send messages to this hidden object. Messages are named with strings, and handled by methods of the same name, so an `"add"` message is handled by the `add()` method.

All arguments to messages and constructors are deep-copied, using the Java serialization mechanism. This enforces isolation, by preventing multiple actors from obtaining pointers to the same mutable object. (Isolation can still be violated, by use of `static` fields, but I decided that this could be verbally forbidden without seriously affecting coding style.)

An example use of this framework follows:

```
// In AddingActor.java:
public class AddingActor {
  private int x = 0;

  public void incrementBy(int y) {
    x += y;

    System.out.println("x␣=␣" + x);
  }
}

// In Main.java:

public class Main {
  public static void main(String[] args) {
    Actor a = new Actor("AddingActor");
    a.send("incrementBy", 2);
    a.send("incrementBy", 5);
  }
}
```

Of course, this isolation comes at a significant performance cost, making scalability analysis impractical. I do not pretend that this grafting of isolation onto a shared-memory language is an elegant one – merely that it models the user experience I wished to emulate.

4. Results

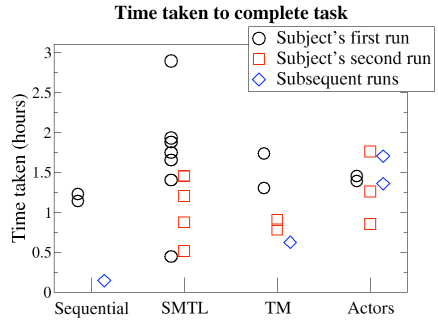
The results of this experiment were inconclusive, showing no significant difference in any objective measurement between the four test conditions. The subjective measurements, however, did show a statistically significant preference for the transactional memory (TM) model over shared-memory threading with locks (SMTL), and suggest a (not statistically significant) preference for the Actor model over SMTL.

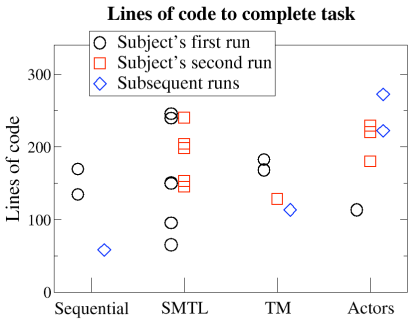
I also document an unexpected phenomenon in the completion data, suggesting a bimodal distribution: Subjects either completed the first task within two hours, or could not within the whole four-hour session.

4.1 Objective Measures of Programmer Effort

Programmer effort is difficult to measure objectively, and sophisticated proxy measurements are contentious. I therefore chose two simple metrics: the time taken for subjects to complete the task, and the number of (non-comment) lines of code in the final program.

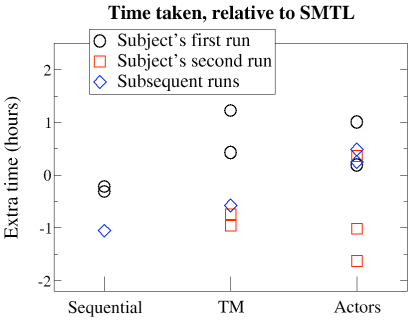
We begin with the aggregate data, showing completion times for every trial on the same graph. In all the graphs in this section, lower numbers indicate better performance.

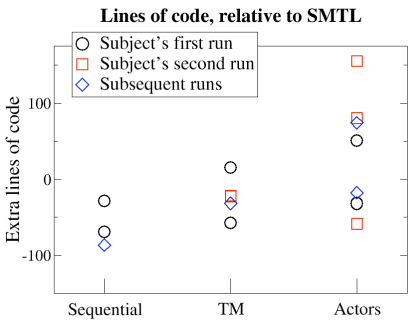




The aggregate data shows no significant difference between the four conditions. This is unsurprising, due to high inter-subject variability.

We now consider within-subject differences in performance between conditions. Each point on the following graphs represents the difference between a subject's performance in the test condition and the SMTL control. Lower numbers indicate better performance than on the SMTL trial; higher numbers indicate worse performance.





Overall, these measurements showed no significant difference between the conditions under test.

In the "time taken" metric, the difference between first and second trials dominates all other variation. The sequential condition (our positive control) shows a suggestive decrease in time taken, but the other two conditions appear dominated by this learning effect.

The learning effect is significant: a paired t test finds a significant difference between subjects' first and second trials ($p = 0.04$).

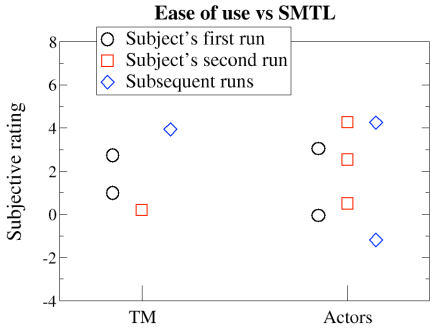
In the "lines of code" metric, we do not see such a difference between first and second trials. The sequential condition again suggests a decrease, but there is nothing particularly convincing here.

In both metrics, the Actor model shows remarkably high variation in performance.

4.2 Subjective impressions

After each task, subjects were given a questionnaire requesting their subjective impression of the task. As well as overall task difficulty, they were asked how the framework compared to writing sequential code and SMTL, and asked directly how easy they found using the framework.

This graph presents the average of all these subjective ratings (sign-normalized such that higher ratings are good, and presented as a difference from ratings in the SMTL condition). The sequential condition is omitted, as most of the survey questions were inapplicable to it.



A within-subjects ANOVA finds that subjects tested on transactional memory significantly preferred it to SMTL ($p = 0.04$).

The preference of subjects tested on the Actor model did not reach the 5% significance level ($p = 0.07$).

4.3 Metric correlation

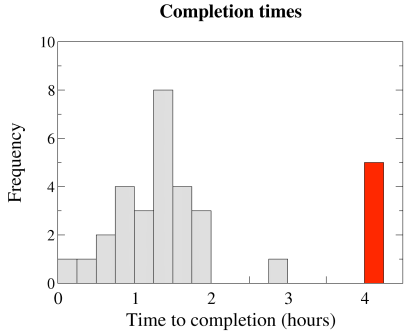
These three metrics (time taken, lines of code, and subjective rating) appear to differ widely. I observed a weak correlation between time taken and lines of code ($r = 0.25$), and between time taken and subjective rating ($r = -0.18$), but none at all between lines of code and subjective rating ($r = 0.05$).

This small correlation suggests that they measure *something*, but their disagreement implies that these metrics should not be wholly trusted.

4.4 Completion times: Does the camel have two humps?

An unexpected phenomenon is visible in the subjects' completion times: Subjects either finished the first run within two hours, or could not finish it at all within the four-hour session. (There was only one exception, who completed the first task in three hours. He did not attempt the second.)

The distribution of completion times for all tasks is displayed below. The solid bar represents subjects who did not complete a task at all. The yawning gap between these five subjects and the rest of their cohort suggests that the distribution is not continuous:



A bimodal distribution of programmer ability has been posited before [Dehnadi and Bornat 2006], but this controversial suggestion distinguished between those who entirely "cannot learn" to program and those who can.

By contrast, these unsuccessful subjects wrote valid code, with control and data structures no less complicated than their successful peers'. They did not give up, or stop debugging their programs, until stopped at the four-hour mark. Post-session interviews indicate that they correctly understood the problem. In short, the circumstantial evidence does not support the idea that these subjects were simply incompetent, gave up, or failed to understand the problem.

This result, then, remains an intriguing mystery.

5. Discussion

This inconclusive result is disappointing, but also instructive.

Broadly, it illustrates the difficulties of empirical research into such a complicated phenomenon as programmer productivity. It also illustrates the power of many hard-to-avoid confounding factors, which I will discuss in a moment.

This study also has something to say about the relative merits of subjective and objective research. It is telling that the subjective survey results actually succeeded in reaching statistical significance, whereas the empirically-observed proxies for effort barely suggested anything.

This is not to say that subjective studies are inherently superior – their substantial biases are the reason we have empirical studies in the first place. However, human introspection can sometimes tell us more readily about difficult-to-measure phenomena such as "effort" than objective (but weak) proxies.

5.1 Possible explanations

I will now consider some confounding factors which might have caused this inconclusive result, even in the presence of large usability differences in the frameworks tested.

5.1.1 Weak metrics

Their dismal internal consistency suggests that the available metrics for programmer effort are not powerful or reliable tools. More accurate instruments would be required to measure any but the largest effects.

5.1.2 Learning effects and subject variability

The learning effect between the two trials for each user greatly interfered with the results. However, designing a study such as this one inevitably puts the experimenter between a rock and a hard place.

If I had designed an experiment where each subject attempts only one task, under one condition, the results would have been swamped by inter-subject variation. It would take an impractical number of subjects to see any effect at all.

The design I actually chose controls for some subject variability – but far from all – at the expense of a short-term learning effect that ends up swamping the results.

5.1.3 Familiarity

Most programmers have experience with the standard threading model, and so come to this study with a substantial familiarity bias in favor of the SMTL condition. By contrast, few subjects had previous experience with transactional memory, and none with the Actor model. This study therefore captured the effort required to learn these models for the first time, as well as the effort of solving the problem.

This might be mitigated with a practice session before the study, in which users gained some experience with the unfamiliar model before attempting the task. However, no length of practice session can eliminate the learning curve entirely, and familiarity is an ever-present confounding factor.

5.1.4 Toy Problem Syndrome

The task in this experiment was, necessarily, a small problem which could be solved in two hours and 200 lines of Java. Most solutions included a single, symmetrical concurrent kernel. By contrast, the usability problems of concurrency are often related to complexity, and the inability to hold the behavior of all threads in the programmer's head at once.

The proverbial "concurrency problem from hell" is an intermittent deadlock whose participants are distributed across

80,000 lines of code written by ten different people. Such situations are difficult to model experimentally.

It is a perennial problem that empirical studies of programming can only ever test toy problems, and it is difficult to imagine any experimental design which could sidestep it. Lengthening the task might have helped slightly, but recruiting volunteers for a four-hour study was difficult enough as it was. (There is a reason that much of the work in this field is done by educators, at institutions where course credit may be awarded for participating in such experiments.)

5.2 Additional threats to validity

Lest we think that these big problems are the only ones, a number of issues would have qualified even a significant positive result from this experiment:

- The choice of problem greatly affects the suitability of different paradigms. There is unlikely to be One True Model which outperforms all others, all the time. Choice of a problem which can be more easily modelled with shared state, or message passing, could therefore have a significant effect on the results.

  Controlling for this issue would require a large-scale study including representatives of, say, all the Berkeley Dwarfs [Asanovic et al. 2006], in an attempt to cover all common parallel computations.

- Software spends almost all of its life being maintained, but this experiment only observed the initial creation of a program. Safe and easy modification – low *viscosity*, in the terms of the Cognitive Dimensions framework [Green and Petre 1996] – might even be more important than the ease with which a program is first written.

- The frameworks used in this experiment lack features available in industrial implementations. For example, the Actor model implementation lacked multicast or scatter-gather support (available in the MPI framework [Snir and Otto 1998]). The SMTL condition explicitly prohibited the use of utilities such as synchronization barriers from the `java.util.concurrent.*` package. Subjects repeatedly re-implemented both barriers and scatter-gather patterns during the experiment.

- Actor messages, unlike normal method calls, are only checked at run time, not at compile time. This may have spuriously reduced performance in that condition.

- The unrealistic performance of the experimental frameworks might have cause subjects to mis-optimize their programs. (This was not borne out by observations or discussions after experimental sessions; no subject so much as profiled their code for bottlenecks.)

- The students participating in this study may not be representative of professional programmers. This could go either way: they might be more flexible and open to new techniques, or less practiced at quickly getting the hang of unfamiliar tools.

- The awkward data structures used to provide inputs appear to have had an anchoring effect on the subjects. Several subjects largely copied the provided data structures instead of devising their own, despite emphatic instructions to the contrary.

6. Related Work

The study closest to the present one was conducted at IBM [Ebcioglu et al. 2006]. They compared performance using three languages for supercomputing clusters: MPI (message passing in C) [Snir and Otto 1998], Unified Parallel C [El-Ghazawi et al. 2005], and IBM's x10 language [Charles et al. 2005]. x10 was convincingly superior, but I expect that this is due more to its garbage collection, memory safety and similarity to Java than to its approach to concurrency. No significant difference was found between MPI and UPC.

This group also described the methods used to observe subjects during that experiment [Danis and Halverson 2006]. Although the software they used is not publicly available, this significantly inspired my methods.

Hochstein et al. [2008] taught a class of students either MPI or a shared-memory C variant for a research computer architecture. They found MPI to involve significantly more effort, but as VanderWiel et al. [1997] note, most of the extra effort is likely to do with manually packing and unpacking message buffers rather than message-passing *per se*.

That study is part of a larger collaboration between several universities to investigate HPC usability, using students from scientific computing classes [Hochstein et al. 2005].

A number of other studies have evaluated the usability of parallel systems by implementing larger projects (often benchmarks), and discussing the experience. These are not controlled experiments, but they avoid the Toy Problem Syndrome, and can control for problem choice. Cantonnet et al. [2004] evaluated UPC on the NAS benchmark suite, Chamberlain et al. [2000] compared Fortran variants, Single-Assignment C and ZPL on a single NAS benchmark, and VanderWiel et al. [1997] compared several C-based languages and HPF over a variety of benchmarks.

Most of this work either predates the multi-core era, or concentrates on distributed-memory supercomputing systems. It is therefore not enormously useful for evaluating the pressing problems facing general-purpose computing today, or the proposed solutions, which make heavy use of language features not available in Fortran or C. However, their methods, and quality of results, are still instructive.

Acknowledgments

I am extremely grateful to Luke Church and Alan Blackwell for their assistance, suggestions and feedback while designing and performing this experiment, as well as to my supervisor, Simon Moore.

References

Erlang programming language. http://www.erlang.org/.

K. Asanovic, R. Bodik, B. Catanzaro, J. Gebis, P. Husbands, K. Keutzer, D. Patterson, W. Plishker, J. Shalf, S. Williams, and K. Yelik. The landscape of parallel computing research: A view from berkeley. Technical Report UCB/EECS-2006-183, EECS Department, University of California at Berkeley, December 2006. URL http://www.gigascale.org/pubs/1008.html.

Franois Cantonnet, Yiyi Yao, Mohamed Zahran, and Tarek El-Ghazawi. Productivity analysis of the upc language. *Parallel and Distributed Processing Symposium, International*, 15:254a, 2004. doi: http://doi.ieeecomputersociety.org/10.1109/IPDPS.2004.1303318.

Bradford L. Chamberlain, Steven J. Deitz, and Lawrence Snyder. A comparative study of the nas mg benchmark across parallel languages and architectures. In *Supercomputing '00: Proceedings of the 2000 ACM/IEEE conference on Supercomputing (CDROM)*, page 46, Washington, DC, USA, 2000. IEEE Computer Society. ISBN 0-7803-9802-5.

Philippe Charles, Christian Grothoff, Vijay Saraswat, Christopher Donawa, Allan Kielstra, Kemal Ebcioglu, Christoph von Praun, and Vivek Sarkar. X10: an object-oriented approach to non-uniform cluster computing. *SIGPLAN Not.*, 40(10):519–538, 2005. ISSN 0362-1340. doi: http://doi.acm.org/10.1145/1103845.1094852.

C. Danis and C. Halverson. The value derived from the observational component in integrated methodology for the study of hpc programmer productivity. In *P-PHEC workshop, held in conjunction with HPCA*, February 2006.

Saeed Dehnadi and Richard Bornat. The camel has two humps (working title). 2006. URL http://www.cs.mdx.ac.uk/research/PhDArea/saeed/paper1.pdf.

Kemal Ebcioglu, Vivek Sarkar, Tarek El-Ghazawi, and John Urbanic. An experiment in measuring the productivity of three parallel programming languages. In *P-PHEC workshop, held in conjunction with HPCA*, February 2006.

Tarek El-Ghazawi, William Carlson, Thomas Sterling, and Katherine Yelick. *UPC*. Wiley, 2005. ISBN 9780471220480.

T. R. G. Green and M. Petre. Usability analysis of visual programming environments: a 'cognitive dimensions' framework. *Journal of Visual Languages and Computing*, 7:131–174, 1996.

Dan Grossman. The transactional memory / garbage collection analogy. In *OOPSLA '07: Proceedings of the 22nd annual ACM SIGPLAN conference on Object-oriented programming systems and applications*, pages 695–706, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-786-5. doi: http://doi.acm.org/10.1145/1297027.1297080.

Carl Hewitt, Peter Bishop, and Richard Steiger. A universal modular actor formalism for artificial intelligence. In *IJCAI*, pages 235–245, 1973.

C. A. R. Hoare. Communicating sequential processes. *Commun. ACM*, 21(8):666–677, 1978. ISSN 0001-0782. doi: http://doi.acm.org/10.1145/359576.359585.

L. Hochstein, V.R. Basili, U. Vishkin, and J. Gilbert. A pilot study to compare programming effort for two parallel programming models. *Journal of Systems and Software*, 81(11):1920–1930, 2008. cited By (since 1996) 2.

Lorin Hochstein, Jeff Carver, Forrest Shull, Sima Asgari, and Victor Basili. Parallel programmer productivity: A case study of novice parallel programmers. In *SC '05: Proceedings of the 2005 ACM/IEEE conference on Supercomputing*, pages 35+, Washington, DC, USA, 2005. IEEE Computer Society. ISBN 1-59593-061-2. doi: 10.1109/SC.2005.53. URL http://dx.doi.org/10.1109/SC.2005.53.

Guy Korland, Nir Shavit, and Pascal Felber. Poster: Noninvasive java concurrency with deuce stm. In *Systor '09, Haifa, Israel*, 2009.

Edward A. Lee. The problem with threads. *Computer*, 39(5):33–42, 2006. ISSN 0018-9162. doi: http://doi.ieeecomputersociety.org/10.1109/MC.2006.180.

S. Peyton-Jones. Beautiful concurrency. In G. Wilson, editor, *Beautiful Code*. O'Reilly, 2007.

David B. Skillicorn and Domenico Talia. Models and languages for parallel computation. *ACM Computing Surveys*, 30:123–169, 1996.

Marc Snir and Steve Otto. *MPI - The Complete Reference*. MIT Press, Cambridge, MA, USA, 1998. ISBN 0262692155.

Sriram Srinivasan and Alan Mycroft. Kilim: Isolation-typed actors for java. In *European Conference on Object Oriented Programming ECOOP 2008*, 2008.

Steven P. VanderWiel, Daphna Nathanson, and David J. Lilja. Complexity and performance in parallel programming languages. *High-Level Programming Models and Supportive Environments, International Workshop on*, 0:3, 1997. doi: http://doi.ieeecomputersociety.org/10.1109/HIPS.1997.582951.