Panini: a Capsule-oriented Programming Language for Implicitly Concurrent Program Design

Eric Lin Hridesh Rajan

Iowa State University
Ames IA, USA
{eylin, hridesh}@iastate.edu

Abstract

This demonstration will present Panini, a new programming language designed with an objective to help programmers with concurrent programming. Current abstractions for concurrency fall into two categories: explicit abstractions for concurrency such as threads, and implicit abstractions for concurrency such as actors. Explicit concurrency abstractions are hard to use, reason about, and error prone. Implicit abstractions like actors help solve these problems, but they require programmers to adapt to the asynchronous style of programming. Many programmers find this adaptation hard. We will demonstrate the notion of *capsules* in Panini. A capsule is an implicit abstraction for concurrency that has many properties of actors, but provides a logically synchronous programming model to programmers. Main technical challenge in realizing capsules was to maximize concurrency while minimizing overhead and abstracting away all the details of concurrency from the Panini programmer. We will also demonstrate scalability benefits of Panini programs without writing a single line of explicitly parallel code, and compile-time checking of concurrency-related properties such as confinement violation, and sequential consistency.

Categories and Subject Descriptors D.3.3 [*Language Constructs and Features*]: Concurrent programming structures

Keywords capsules, implicit concurrency, modularity, synergy

1. Problem: Concurrent Programming is Hard

Concurrent programming is hard. Most programmers find it difficult to write concurrent programs. Moreover, a large number of programmers are used to thinking sequentially about their programs. This makes the task of reasoning about a concurrent program even more difficult because it requires reasoning about interleaving of concurrent tasks. On top of that, to ensure concurrency safety, the concurrency construct applied in the program often obscures the logic flow of the program, resulting in difficulty understanding both the logic of the program and the concurrency constructs.

As an example, Figure 1 is a simple concurrent program in which a number of “worker” tasks execute a Monte Carlo approximation of π concurrently; a “master” task combines the results as the workers finish. Figure 1, left shows an explicitly concurrent implementation of this program in Java. The explicitly concurrent Java program has the application’s concerns tangled with two concurrency concerns: creation and starting of new threads, and synchronization between these threads. This makes it harder to write, test, reason about, and evolve this concurrent program.

2. Solution: Eliminate Concurrent Programming

Capsule-oriented programming is a new style designed to address these challenges of concurrent programming [1]. The first objective of capsule-oriented programming is to allow programmers to design and implement more modular programs, and in doing so expose implicit concurrency in program design. To achieve this goal, we have developed a new abstraction called *capsule*. A capsule is like a process, it encapsulates objects stored in its local region of memory and provides public operations that can be called by other capsules. A significant difference between capsules and similar abstraction *actor* is that inter-capsule calls are logically synchronous, and the topology of capsule-oriented can be statically determined. These differences decrease the impedance mismatch between capsule-oriented programming model and the mainstream imperative programming model, and makes it easier to design polynomial-time algorithms for analyzing capsule-oriented programs that can be integrated into industry-strength compilers.

To illustrate, consider a capsule-oriented implementation of our motivating example in Figure 1, right. This program has two capsules that are combined to form a static topology in the design block on lines 13-16. As in the Java program, a “Master” task combines the results as the workers finish, but all of the concurrency details are hidden from the programmer. Each call to `compute` on line 21 executes asynchronously in an instance of a `Worker` capsule; the returned `Number` object is transparently replaced by a future for the eventual result. The futures provide an implicit barrier; that is, in the call to `value` on line 22, the execution of the `run` procedure in master capsule blocks until the corresponding *Worker* has finished computing its result.

The explicitly concurrent Java program has the applications’s concerns tangled with the concurrency concerns, whereas the Panini program abstracts away the details of concurrency. As Figure 1 shows, the performance of the Panini program is comparable to that of the thread-based program. A more significant potential benefit is that the Panini compiler can be employed to guard against race conditions when parallelism is introduced into an application.

The Panini language and associated tools are made available as an open source project under Mozilla Public License 1.1. The source code and examples, as well as pre-built binaries, are available for download from `http://www.paninij.org/`.

SPLASH ’13, October 26–31, 2013, Indianapolis, Indiana, USA.
Copyright is held by the owner/author(s).
ACM 978-1-4503-1995-9/13/10.
http://dx.doi.org/10.1145/2508075.2514573

Java program with threads and synchronization to compute π

```
class Worker implements Runnable {
  long num;
  private final CountDownLatch doneSignal;
  Worker(long num, CountDownLatch doneSignal) {
    this.num = num;
    this.doneSignal = doneSignal;
  }
  Number _circleCount = null; //Emulates return value of worker
  Number getCircleCount() { return _circleCount; }
  public void run() {
    _circleCount = new Number(0);
    for (long j = 0; j < num; j++) {
      double x = Math.random();
      double y = Math.random();
      if ((x * x + y * y) < 1) _circleCount.incr();
    }
    doneSignal.countDown();
  }
}
class Master {
  void assign(long totalCount, int numWorkers) {
    CountDownLatch l = new CountDownLatch(numWorkers);
    Worker[] workers = new Worker[numWorkers];
    for (int i = 0; i < numWorkers; ++i) {
      workers[i] = new Worker(totalCount/numWorkers, l);
      new Thread(workers[i]).start();
    }
    try {
      l.await();
    } catch (InterruptedException e) { /* Error recovery */ }
    Number[] results = new Number[numWorkers];
    for (int i=0; i< numWorkers; i++)
      results[i] = workers[i].getCircleCount();
    long total = 0;
    for (Number result: results) total += result.value();
    double pi = 4.0 * total / totalCount;
  }
}
public class Pi {
  public static void main(String[] args) {
    Master master = new Master();
    master.assign(50000000,10);
  }
}
```

Panini program to compute π

```
capsule Worker {                          // Capsule declaration
        Number compute(int num) {          // Capsule procedure
                Number _circleCount = new Number(0);
                for (int j = 0; j < num; j++) {
                        double x = Math.random();
                        double y = Math.random();
                        if ((x * x + y * y) < 1) _circleCount.incr();
                }
                return _circleCount;
        }
}
capsule Master {
        design {
                Worker workers[5000000]; //Capsule arraydeclaration

        }
        void run(){
                Number[] results = new Number[workers.length];
                for (int i = 0; i < workers.length; i++)
                        results[i] = workers[i].compute(50000000);
                        // Inter-capsule procedure
                              calls long total = 0;
                for (Number result: results) total += result.value();

                double pi = 4.0 * total / 50000000;
        }
}
```

Performance results

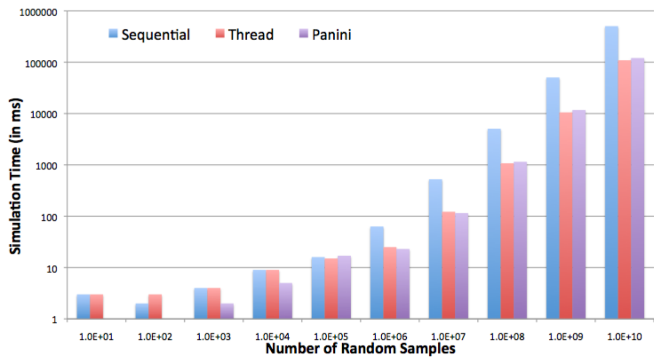

Figure 1. Panini programs are shorter, get speedup, and don't have 2 type of bugs: sequential inconsistencies and data races due to sharing.

3. Demonstration Description

We will present the key features of the Panini Language through several examples. In particular, we will show a comparison of an example with a traditional approach of using explicitly concurrent language features, and the same example in Panini for contrast. We will demonstrate installation of the Panini compiler, compilation, and profiling process for Panini programs. The Panini compiler is built on top of the standard OpenJDK Java compiler (javac) and is fully backwards compatible with pure Java programs.

Acknowledgements This work has been supported in part by the US National Science Foundation (NSF) under grants CCF-11-17937, CCF-10-17334, and CCF-08-46059.

References

[1] H. Rajan, S. M. Kautz, E. Lin, S. Kabala, G. Upadhyaya, Y. Long, R. Fernando, and L. Szakács. Capsule-oriented programming. Technical Report 13-01, Iowa State U., Computer Sc., 2013.

Biographies This demonstration will be carried out by Eric Lin and Hridesh Rajan. *Eric Lin* is a graduate student in the department of computer science at Iowa State University. He earned his undergraduate degree in Computer Science from Iowa State University in 2012. Eric has worked on the frontend and the backend of the OpenJDK-based Panini compiler, focusing on code transformation strategies and type-checking. *Hridesh Rajan* has extensive experience in developing new languages. He developed the Panini language. Prior to that, he developed the Boa language, a language for data-mining large software repositories. He has also developed the Ptolemy language, an event-based language for advanced separation of concerns and the aspect-oriented language Eos. Rajan has successfully given multiple tutorials and demonstrations on other topics at AOSD'10, FSE'10, ECOOP'11, AOSD'11, and ASE'11.

Target Audience The target audience for this demonstration is intermediate level developers and researchers. Participants will need a working knowledge of Java and object-oriented development. The demonstration will provide any additional background material.