Self-Adaptive Parallel Programming
Through Tunable Concurrency
Tai Nguyen  Xinghui Zhao
School of Engineering and Computer Science, Washington State University
{tai.t.nguyen, x.zhao}@wsu.edu

Abstract
Recent advances in hardware architectures, particularly multicore and manycore systems, implicitly require programmers to write concurrent programs. However, writing correct and efficient concurrent programs is challenging. We envision a system where the concurrent programs can be self-adaptive when executing on different hardware. We have developed two different tuning policies, which enable users’ programs to adjust their level of concurrency at compile-time and run-time respectively.

Categories and Subject Descriptors D.1.3 [software]: Programming Techniques—Concurrent Programming

General Terms Performance, Experimentation

Keywords Actors; Parallel Programming; Concurrency

1. Introduction
Recent studies show that no matter how many cores there are in the processor, for most of the desktop applications, only 2 or 3 cores are more than adequate, and others are underutilized [2]. The mismatch between users’ programs and the underlying hardware presents challenges in leveraging the full power of the multicore technology. In this paper, we address these challenges by proposing a new parallel system that separates concurrency from the functionality code of a program. Specifically, we develop a tool which serves as a tune knob in between users’ programs and the underlying hardware, by dynamically adjusting the thread-level concurrency based on different tuning policies. These tuning policies can be implemented separately from users’ programs as plug-in modules at run-time.

2. Tunable Concurrency
Our system is built on top of the Actor model of concurrency [1], because it provides a convenient and less error-prone way to write concurrent programs. The Actor model encapsulates objects along with threads of execution. Therefore, earlier actor frameworks usually use one-thread-per-actor implementation of actors. However, it turns out that in practice, one-thread-per-actor implementation of actors is not particularly efficient, because of the overhead caused by context-switching among actor threads. It is more efficient to have a pool of threads, where each thread processes messages for multiple actors in some order, as shown in the latest version of ActorFoundry [3]. This observation has led us to explore the potentials of programming with decoupled concurrency. In other words, the code of functionality can be separated from the code of concurrency. In ActorFoundry, a fixed number of worker threads are employed for processing messages for all actors, and this number does not change throughout the course of execution.1 We investigate the opportunities to dynamically tune this number to change level of concurrency at run time based on different policies.

Static Tuning: Separating level of concurrency from users’ programs enable us to tune the level of concurrency based on the hardware configuration. As shown in Figure 1(a), static tuning is a concurrency tuning policy which takes into consideration the underlying hardware configuration at the compile-time, and then makes decisions on the suitable level of concurrency for users’ programs. Static tuning is performed at the compile-time, therefore it does not introduce run-time overhead which affects the applications’ performance. However, static tuning requires that the information about the available hardware resources is known in advance, and the application can fully utilize the hardware resources, i.e., no other computations are competing for resources.

Dynamic Tuning: Figure 1(b) shows the dynamic tuning policy which should be enforced at the run-time. Specifically, the dynamic tuning policy initializes the thread pool by setting the number of worker threads to be a small number,

---

1 Unless starvation happens, which violates the Actor model’s fairness property, in that case more workers will be created in ActorFoundry.
i.e., 1. During run-time, it increases the number of worker threads, and observes the progress of the user’s application. If the feedback is positive, which means increasing the number of worker threads results in better performance than before, the dynamic tuning increases the number of worker threads again, until the feedback from user’s application becomes negative, which indicates that a balance has been reached. At this point, increasing the number of worker threads again usually results in negative impact on the overall performance, because of unnecessary context switching. Dynamic tuning can be invoked periodically with different tuning directions (increasing or decreasing), aiming for achieving better performance.

Dynamic tuning is different from static tuning in that it does not require prior knowledge about the available hardware resources. It is performed at run-time and adjusts the level of concurrency dynamically based on the progress of the user’s application. Therefore, it is a feedback based approach, and it is more flexible in adapting to changes of resource availability in the system. However, it does require that the progress of application can be evaluated at run-time. For a large class of iteration based problems, such as scientific computing problems, the progress of application can be simply represented by the number of iterations that have been completed.

3. Experimental Results
We illustrate the effectiveness of our approach using a case study, Gravitational N-Body Problem (GNBP). We choose GNBP as an example, because it represents a large class of computations which consist of a series of parallel computations connected by barriers. We have implemented GNBP using manager-worker style. Specifically, a manager actor broadcasts a signal to the actors which represent the bodies. The body actors then read the information about all bodies, including their coordinates and masses, use the information to calculate the forces, velocities, and new coordinates for their corresponding bodies, and then send their updated information to the manager.

We run a 20000-body computation on both hardware for 200 iterations. The results are shown in Figure 2. In both cases, our approach outperforms the original ActorFoundry, and dynamic tuning achieves the best performance among the three.

![Figure 1. Concurrency Tuning](image)

![Figure 2. Performance Comparison of Tuning Policies](image)

4. Conclusion
In this paper, we present a parallel programming paradigm, in which concurrency is separated from the functionality code. Experimental results show that our approach enables users’ programs to achieve optimal performance on different hardware without changing the code. Work is ongoing to apply the idea of concurrency tuning to non-actor based applications and data-intensive applications.

Acknowledgements
Support from the WSU New Faculty Seed Grant and the WSUV Research Mini Grant is gratefully acknowledged.

References