My main research interest is to improve the reliability of parallel software systems. Specifically, my research focuses on tools and techniques that facilitate the detection and debugging of parallel software errors. There are two main reasons why software reliability has been and continue to be a very important issue in recent years, especially so with high performance parallel software systems. First, systems are becoming more and more software intensive rather than hardware intensive. Many safety critical systems such as air traffic control, weapon control, energy systems, medical systems, etc. are all software intensive. Second, unreliable software is costly, in terms of money (lost customers, lawsuits, etc) and human time (developers and users trying to determine the cause of the errors and fix the bugs that cause the errors) The situation is even more serious in high performance parallel systems, where code sometimes must run for days and resources are scarce.

Even though there are many techniques and tools that help developers debug parallel software errors, they basically fall into one of these three categories: static methods, dynamic methods, and model checking. Static methods have the advantages of being input-independent since they verify the program at the source code level. However, they tend to provide too many false alarms, especially for large code base, due to the lack of runtime knowledge. Model checking methods are very powerful for small programs in terms of verification coverage but they quickly become impractical for large software due to the infeasibility of building models for such software. Dynamic (runtime) methods (such as testing, dynamic verification) are the most applicable methods for large scale software since they produce no false alarms and also require little work from the tool users. Hence, my primary research focus is efficient and scalable dynamic tools which help increase parallel software reliability by facilitating error detection and debugging.

**Research Background**

Towards the aforementioned goal of empowering and enabling developers to create more reliable software, I have designed and implemented DAMPI [1,2], a scalable dynamic verification framework for message passing programs. DAMPI offers verification coverage for large scale MPI programs over the space of MPI non-determinism. It is currently the only tool that can offer such coverage at large scale. DAMPI was highly regarded by the community, winning the 2nd prize in the ACM Student Research Competition held at PLDI 2010 during its early stage. The main DAMPI work was also accepted for publication in SuperComputing 2010. Prior to my work on DAMPI, I also co-designed and implemented ISP [3,4,5,6], another dynamic verifier for MPI program, aiming at smaller scale parallel systems. ISP guarantees full exploration of all possible outcomes of non-deterministic MPI programs. Within such space, ISP tests the programs for common MPI errors such as deadlocks, resource leaks, type errors, etc.

I also contributed to Umpire, a dynamic testing tool for MPI programs. Specifically, I was responsible for porting Umpire to Linux (Umpire originally worked only with AIX). The task involving making necessary changes to Umpire’s data structures to make it compatible with newer MPI implementations. A version of Linux Umpire is now installed and use within the Lawrence Livermore National Laboratory (LLNL).
All of the projects that I’ve actively been working on aim to increase the reliability of high performance parallel software and increase the programmers’ productivity by facilitating the debugging and testing process.

**Current Research**

The MPI standard allows many non-deterministic constructs such as MPI_Recv (MPI_ANY_SOURCE) or MPI_Waitany. These constructs provide great flexibility for programmers and in some instances, make the coding process simpler. However, if not being used carefully, these constructs can (and often) create unexpected outcomes in the programs. Bugs of this type are among the hardest to debug because they appear intermittent and do not show up in all traces. Unfortunately, existing tools do not offer coverage with respect to MPI non-determinism, simply because most dynamic tools are restricted to the information observed during the current program execution and lack the ability to evaluate any alternative outcomes of non-deterministic constructs.

In my past and current research, I have worked on DAMPI and ISP, two runtime MPI tools that can detect common MPI errors such as deadlocks, resource leaks, type mismatch with coverage over the space of MPI non-determinism. One of the key problems addressed in these works is: given a program with non-deterministic MPI constructs and a test input, is there an efficient way to test the program for other non-deterministic outcomes?

1) **DAMIPI**: In the past year I’ve been working on designing and implementing DAMPI, a scalable dynamic verification framework for MPI programs. DAMPI is designed as multiple modules within the PnMPI framework, which makes it easy to inter-operate with other MPI tools in the future. This is due to the PnMPI’s ability to stack multiple MPI tools together. In a nutshell, DAMPI takes an MPI program and its input, runs it on its intended platform, tests it for common MPI errors, and detects whether alternative behaviors (due to the usage of non-deterministic constructs) are possible. If alternative outcomes are deemed possible, the program is replayed and DAMPI would force those outcomes to search for errors. The verification is fully customizable to allow users to focus coverage to region of interests. DAMPI has been tested with a wide range of large scale applications with over a thousand processes with very impressive results in terms of overhead[1]. DAMPI is currently the only tool that offers coverage for MPI non-determinism at large scale. This is largely due to a novel distributed algorithm that uses Lamport clock to establish happens-before relationship between MPI calls originating from different processes. While the choice of Lamport clock might omit some possible schedules in the search space, our experience with MPI benchmarks has shown that the search space being omitted is usually tiny (if at all). Clearly, the benefits of scalability far outweighs the cost of using vector clocks (which offer higher precision at the cost of scalability), which has been the central design philosophy of DAMPI since the beginning.

2) **ISP**: Prior to my work on DAMPI, I co-designed and implemented ISP, another dynamic verifier for MPI program. Although the goals of ISP and DAMPI are the same, the algorithms
behind each of them are totally different. ISP achieves coverage through the use of a centralized scheduler, which controls the execution of MPI programs. Essentially, ISP maintains the happens-before relationship using a centralized process, while DAMPI does so in a distributed manner (and hence the scalability of DAMPI). Due to its centralized scheduler and other limitations, ISP does not work well beyond a few dozens of processes. However, it remains a great tool for developers to use during their development cycle, where testing is usually restricted to a small number of processes. My work on ISP turned it from a research project into a solid tool that has been released to the community. GEM, an ISP plugin that allows ISP to be invoked under Eclipse, is now a part of Eclipse Parallel Tools Platform (PTP).

3) **Other current research:** My work on DAMPI also motivated my participation in the piggyback proposal currently being presented to the MPI forums to be included in the MPI 3.0 standard. Piggybacking, the act of sending extra information along with outgoing messages, is a key module of the DAMPI framework. The proposal effort is led by Dr. Martin Schulz of LLNL and joined by several other people. We proposed several different methods to efficiently transferring extra data with every MPI message. The proposal is being favorably considered by the MPI forums.

**Future Research:**

1) **Improving reliability of heterogeneous parallel systems:** At the time of this writing, two systems in the top ten of the top500 list are accelerator based. Many more scientific computing code will turn to accelerator based systems to take advantage of the larger degree of parallelism. However, research in correctness is still in its early stage and the needs for tools as well as techniques are greater than ever. I would like to apply my existing knowledge and expertise in working with message passing tools to these new heterogeneous systems to create tools that help developers make software more reliable.

2) **Extend correctness checking into hybrid code base:** In order to fully take advantage of the multi-core processors available in today’s high performance systems, more and more programmers are switching to hybrid programming: MPI + OpenMP, MPI + pthreads, etc rather than pure message passing. However, the extra performance does not come without its own pitfalls. The increased complexity and parallelism makes the program more error prone and harder to debug. Now correctness checking tools have to deal with an extra source of non-determinism coming from the interaction of multiple threads running at the same time. I would like to extend DAMPI to be able to handle hybrid code base as well.

3) **Heuristic based search:** Full coverage of MPI programs is often infeasible, even if desirable. The issue will become even more prevalent with hybrid code due to the (much) higher degree of non-determinism. Given a limited amount of time, full coverage tend to bias the search towards the beginning of the search space. Furthermore, it’s our experience that bugs are often caught with limited exploration depth. This means an efficient dynamic verifier needs to have the ability to restrict its search space intelligently. Towards this goal, we have experimented with several heuristic based searching algorithm in DAMP, one of which is bounded mixing [1], which is based on our observation that the impact of a non-deterministic outcome has limited scope.
In other words, by the time the execution reaches the middle of a long running program, the choice made in the very beginning of the program (upon a non-deterministic construt) no longer matters. By controlling the range of the impact, users can greatly limit the searching space without losing much code coverage. I would like to actively explore other heuristics to offer the users even more option to fine tune their search space.

References