Generating Testable Hypotheses from Tacit Knowledge for High Productivity Computing

Sima Asgari[1], Lorin Hochstein[1], Victor Basili[1,2], Jeff Carver[3], Jeff Hollingsworth[1], Forrest Shull[2], Marvin Zelkowitz[1,2]
[1]Computer Science Department - University of Maryland, College Park, 20742 MD, USA
[2]Fraunhofer Center for Experimental Software Engineering, College Park, 20742 MD, USA
[3]Mississippi State University, Mississippi State, MS 39762, USA
{sima,lorin,basili,hollings,mvz}@cs.umd.edu, fshull@fc-md.umd.edu, carver@cse.msstate.edu

Abstract
In this research, we are developing our understanding of how the high performance computing community develops effective parallel implementations of programs by collecting the folklore within the community. We use this folklore as the basis for a series of experiments, which we expect, will validate or negate these assumptions.

Keywords
High Productivity Development Time Experimental Studies, Tribal Lore, Folklore, Tacit Knowledge Solicitation, Testable Hypotheses, Focus Groups

1. Introduction
The DARPA High Productivity Computing Systems (HPCS) project has goals of "providing a new generation of economically viable high productivity computing systems for national security and for the industrial user community," and initiating "a fundamental reassessment of how we define and measure performance, programmability, portability, robustness and ultimately, productivity in the HPC domain"[1].

In order to reassess the definitions and measures in a scientific domain it is necessary to study the basis and source of those definitions and measures. These sources are usually found in the related literature and various documentations existent in the community. However the large amount of tacit information that is merely in people's minds often remains neglected.

Historically, there has been little interaction between the HPC and the software engineering communities. The Development Time Working Group of the HPCS project is focused on development time issues. The group has both software engineering researchers as well as HPC researchers. The strategy of the working group is to apply empirical methods to study parallel programming issues. We have applied similar methods in the past to researching development time issues in other software domains [7].

Because of little interaction between the HPC and SE communities in the past, those of us on the SE side have very little knowledge about the nature of software development in the HPC domain. While the HPC community has not focused on development time issues in the sense of generating publications on these subjects, it has assuredly accumulated a wealth of experience about such matters, leading some HPC practitioners to refer to the field as a "black art". Indeed, those in the community tend to harbor strong (and sometimes contradictory) beliefs about development time issues. It would be inappropriate to disregard this body of knowledge simply because it has not been packaged in a suitable format. Unfortunately, since it currently exists only as tacit knowledge, it is not obvious how to best leverage this expertise. While there has been previous research in trying to capture the needs of HPC programmers as they relate to software development issues [2, 3], there has been little research in trying to capture the knowledge of HPC programmers on software development issues, with a notable exception [5]. In this paper, we describe the initial stages of our work to collect this knowledge, which we refer to as "tribal lore" or "folklore".

By tribal lore or folklore we mean the common beliefs about the interaction between variables such as code development effort, development activities such as debugging, programming models, languages, execution time, etc.

We conducted two separate studies to solicit HPCS folklore and the types of defects common to high-end programming. This paper discusses the process of knowledge solicitation, some initial analysis of the collected information and hypotheses created.

An initial conclusion from the folklore is that debugging parallel code is a particularly difficult task. In order to quantify the debugging difficulty we need to analyze the defects (bugs) in the code to find out the types of defects that programmers encounter when writing parallel code, to understand how common these defects are and to specify how difficult they are to fix.

2. Knowledge Solicitation Process
The development time working group of HPCS is responsible for investigating issues concerning development time within the HPCS framework. We conduct experimental studies by collecting various data during the code development phase of high productivity computing by novices (university students working on class assignments) and professionals working on real projects (case studies) or small sample problems (observational studies).

As the initial set of hypotheses that should be investigated using the collected data, we generate hypotheses from the tacit knowledge collected from the HPC community members. After capturing this knowledge, several testable hypotheses are generated around each issue and we investigate them using the development data that we've collected.

Figure 1 shows the process of knowledge solicitation and analysis. The area inside the dotted rectangle in figure 1 is the current part of the study that we discuss in this paper.

2.1 HPCS Folklore
One of the main goals of the development time working group of HPCS project is to leverage HPC community's knowledge of development time issues. In order to do so, we are soliciting expert opinion on issues related to HPC programming by collecting elements of folklore through surveys, generating discussion among experts on these elements of the lore to increase

[1] http://www.highproductivity.org

precision of statements and to measure degree of consensus and finally generate testable hypotheses based on the lore that can be evaluated in empirical studies.

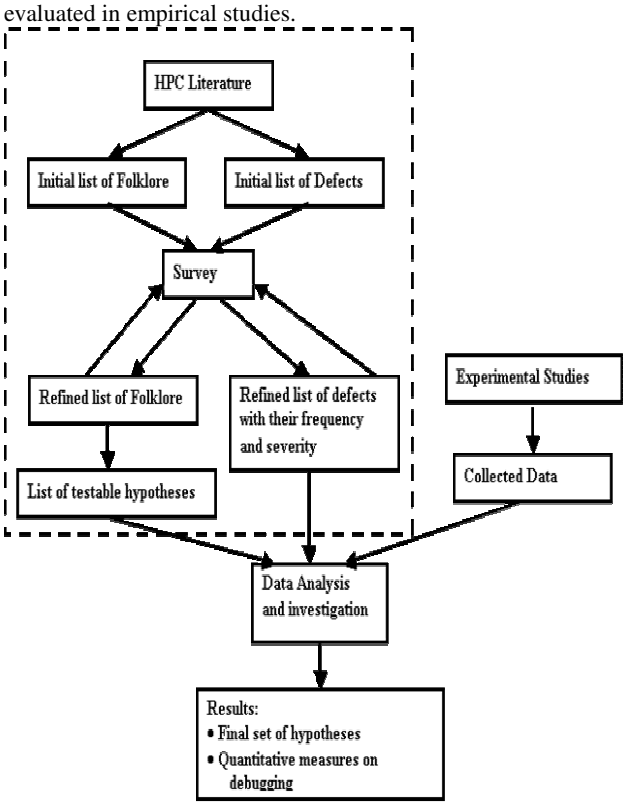

Figure 1: Folklore and defect solicitation process

Before starting the exploratory experiment of collecting peoples' anecdotal beliefs through surveys, we needed an initial set of such anecdotes to both encourage thinking and also use as examples of what we are interested in.

To gather the folklore in HPC, a member of the study group, who is an HPC professor, conducted an informal scan of several sources including lecture notes used in introductory HPC classes at the University of Maryland as well as scanning the Internet for related keywords (including "HPC folklore" and "HPC folklore"). The goal of this process was not to be exhaustive, but instead to gather a sense of the type of information that a beginning HPC programmer might find. This initial list of 10 ideas (the left column of the table in Appendix 1) was recorded and used as the basis for our first survey.

We then asked 7 HPC specialists and professors who regularly teach HPC classes to comment on the initial list. They were asked to give an "agree", "disagree" or "don't know" answer to each lore, give their comments or change suggestions and add any folk lore that they are aware of but is not on the list.

Figure 2 shows the answers. The folklore number 11 in Appendix 1was added by one of the participants at this stage. Generally the comments revolved around clarifying the domain to which the bit of lore applied. For example was the bit of lore talking about a user programming model such as OpenMP or hardware architecture such as a multi-threaded machine.

In order to clarify the questionable points we scheduled a discussion session among the participants. This discussion resulted in some modifications in the way folklore sentences were phrased. The right column of the table in Appendix 1 is the result of this modification.

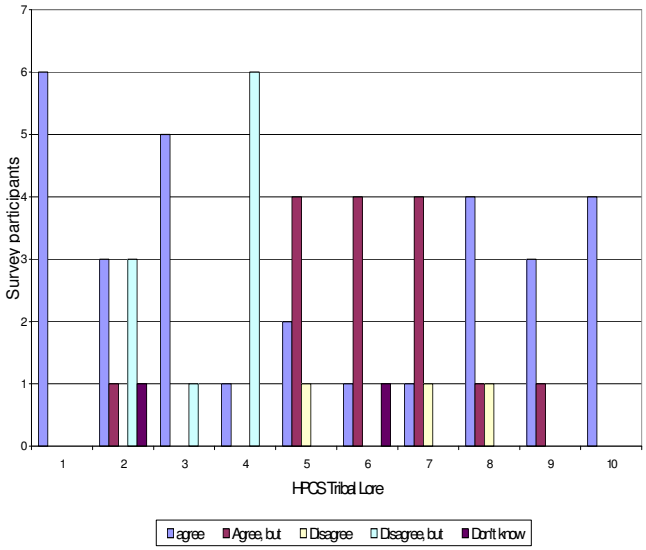

Figure 2: Responses to the initial list of HPC folklore

At some point during the discussion, the participants agreed that "MPI programs don't run well when you use lots of small messages because you get latency-limited". In order to include this in the folklore list, the lore number 12 was added to the list.

At the next step of the study, a survey form was compiled from the current list of 12 folklore and distributed to the participants at the "High Productivity Computing Systems, Productivity Team Meeting" held in January 2005. In order to avoid any bias, some of the randomly selected lore were rephrased to imply the logically inverse sentence. Two sets of survey forms were compiled and distributed randomly.

In Figure 3 (the survey results), the numbers on the x-axis represent the folklore numbers, where the numbers marked with * show that the altered version of the lore was used. In version 1 of the survey the altered phrases of the folklore 1,3,4,6,7,8 and 11 and the original version of the folklore 2,5,9,10 and 12 were used, and in the second version of the survey they were switched. In Figure 3, the third column for each folklore, marked as 'mean", represents the mean value from the two surveys.

The total number of respondents was 10 for the first version and 18 for the second version of the survey. In most cases more than 50% of the participants agreed with the positive lore and disagreed with the altered ones. It seems that folklore numbers 5 and 11 need further investigation since there is less than 30% agreement on them. This emphasizes the fact that there could be large inconsistency between experts' viewpoints and also that the phrasing of the folklore is a very important factor. Before trying to create testable hypotheses based on the folklore, we are updating the folklore phrasing to make sure to collect proper data for testing those hypotheses. Also in order to avoid misinterpretation, the folklore should be phrased as clear and unambiguous as possible, starting from the most controversial ones.

The list of HPC folklore is still in primary stage and needs further refinement. We are classifying and analyzing the comments given to the survey by participants. We are also conducting the survey in our upcoming classroom studies to see how comparable students' and professionals' knowledge is.

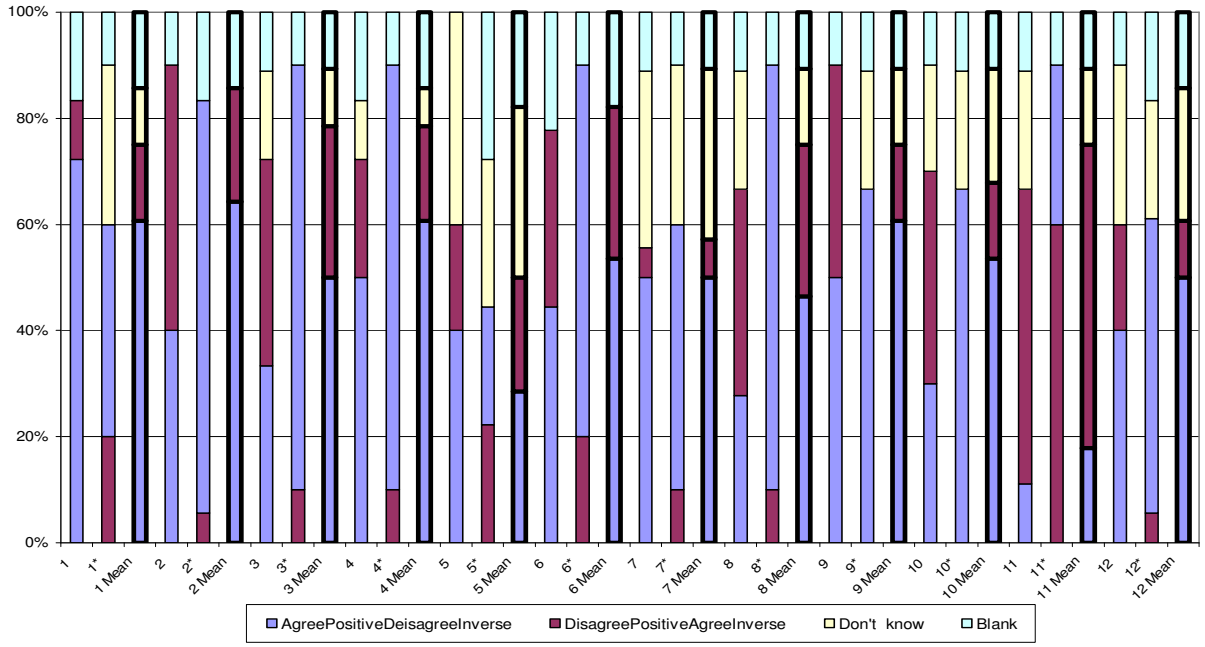

Figure 3: Folklore Survey Results

2.2. Testable Hypotheses

We use the revised folklore to produce testable hypotheses and investigate the hypotheses using the collected data. An example for testable hypotheses is lore number 4 in the updated list of Appendix 1:

Folklore 4: *Debugging race conditions in shared memory programs is harder than debugging race conditions in message passing programs.*

At the first discussion session, the following points were brought up:

- When working in the shared memory model, either it works right away or you will never figure out why.
- Bugs in shared memory are hard to deal with because they can be non deterministic, more subtle and harder to track down.
- Shared memory programs are far easier to develop because:
  - They provide a global address space
  - You do not have to think about the details that you do in message passing
  - You can incrementally develop shared memory programs
- In some cases, it may be harder to debug shared memory programs.

The following hypotheses were created from the above:

Hypothesis 1: *The average time to fix a defect due to race conditions will be longer in a shared memory program compared to a message-passing program.*

To test this hypothesis we measure the time to fix defects due to race conditions.

Hypothesis 2: *On average, shared memory programs will require less effort than message passing, but the shared memory outliers will be greater than the message passing outliers.*

To test this hypothesis we measure the total development time.

Hypothesis 3: *There will be more students who submit incorrect shared memory programs compared to the message-passing programs.*

To test this hypothesis we measure the number of students who submit incorrect solutions.

Table 1: Initial Defects List

Message Passing
M1 ·Deadlock sender and receiver waiting for each other
M2 ·Async Send/Recv and updating variables before send completes
M3 ·Async Send/Recv and reading variables before they arrive
M4 ·Not all processes call a collective communication operation
M5 ·Process tries to send a message to itself
M6 ·Type inconsistencies in Send/Recv
Shared Memory
S1 ·Synchronization bugs
S2 ·Variables that should be thread private are shared
S3 ·Variables that should be shared are private
S4 ·Different locks used for the same variable (i.e. one shared object and a reader lock and a writer lock)
S5 ·Program tires to acquire a lock it already holds
Decomposition
D1 ·Same work done on more than one node (when not intended)
D2 ·Some work not done

3. Defects (bugs) in HPC code

The types of defects that occur in code, their frequency of occurrence, and the effort required to fix them have an impact on productivity. In order to be able to test the defect related folklore, such as folklore 4, which discussed above, we need to analyze defects. We have started a study to analyze the defects by classifying defect types, how common they are and how difficult they are to fix. The process is similar to the one used for the folklore.

We asked a HPC specialist to compile an initial list of defects from the literature and his own experience. Table 1 shows this initial list. At the next step of the study, a survey form from the initial list of defects was compiled and distributed to the participants at the "High Productivity Computing Systems, Productivity Team Meeting" held in January 2005.

In this defect survey we asked the participants to identify the frequency of each defect on a 1 to 5 scale where 1 is the lowest and 5 is the highest frequency. They were also asked to identify the severity of the defect as low, medium or high and at the end they were asked to add any defects that they have experienced but is not on the list.

The initial list of table 1 was used for the survey and table 2 is the list of defects added by the respondents. These new defects will be added to the list for the next round of surveys.

The initial analysis of the survey results shows that for 11 out of 13 defects, more than 60% of respondents believe that the frequency is low or medium, except for 2 defects S1 and S2. The initial conclusion for this observation could be: **"Shared memory defects are more frequent than other types of defect"**, which is a hypothesis generated from the folklore analysis.

We were also able to sort the defects based on their severity. The ascending order of severity based on survey results would be: M5, D1, S5, M4, M6, S3, D2, M1, S4, S2, M3, S1, M2, where M5 is the least and M2 the most severe defect. Investigating the validity of above conclusions as well as drawing further conclusions relies on the results from our ongoing and upcoming survey studies.

3.1 Empirical Defect Study

We are gathering empirical defect data from our HPC development time studies. In a pilot study students that were developing a program for 1D quantum dynamics simulation in C (approximately 150 SLOC) were asked to track time to fix defects while parallelizing code in MPI. As seen in figure 4, in this study **"the defects related to I/O activities are the most time consuming to fix"**. This is another generated hypothesis that is being investigated in our current studies.

4. Conclusion and Future Work

In this paper, we have described our efforts in collecting elements of the collective knowledge of the HPC community, or "folklore", that relate to issues of development time. We have

Table 2: Added defects

MPI sends never received, code runs, but resources never reclaimed
Message failure
Message reordering[2]
Bookkeeping errors in domain decomposition (indexing errors)
Loop with data dependencies get parallelized
Loop without data dependencies does not get parallelized
Pointer problems
Thread stack overflow
Using any distributed memory machine

[2] "Forgetting messages could be reordered"

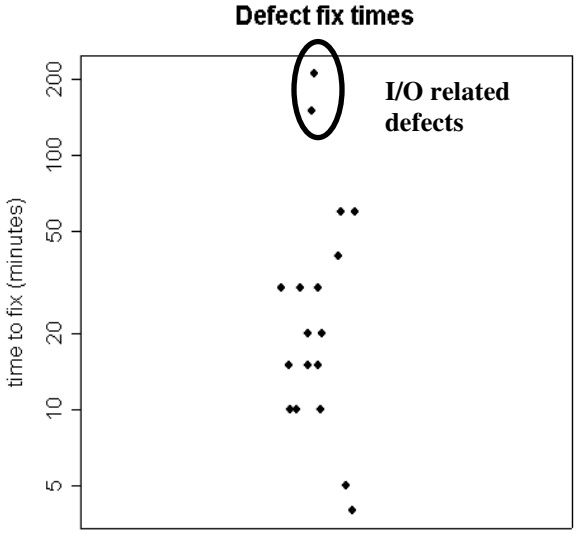

Figure 4: Time to fix defects

employed methods traditionally used in the social sciences such as focus groups and surveys [6]. This work is complementary to our other research in the area, where we are conducting experimental studies to collect development time data and analyze this data by searching for empirical relations between variables such as activity, effort, workflow, performance, and code size.

To run good experiments, we need to develop relevant testable hypotheses. To this end we have tried to understand what the community believes to be true about high end computing and to make explicit the tacit assumptions about a number of issues.

We have been soliciting expert opinion on the issues related to HPC programming by collecting elements of folklore through surveys, generating discussion among experts on these elements of the lore to increase precision of statements and to measure degree of consensus and finally generate testable hypotheses based on the lore that can be evaluated in empirical studies. In some cases we were also able to generate new hypotheses based on the logical relationship between the collected lore.

It is important to note that in order to keep the survey questions simple and not confusing; we had to use the short-and-pithy statement of the lore, although they usually do not reflect people's full understanding of the lore. Therefore the survey respondents may think we are oversimplifying the statements. This is an issue that needs further consideration.

The results so far indicate that there is a large variation in beliefs among experts. For 10 items out of a total of 12 folklore items, the results show agreement among 46% to 65% of the respondents, the maximum agreement being 64%. Two items of lore were less than 30% agreed upon. These items clearly need more clarification.

There are several explanations for this variation. First, it is possible that there is not a wealth of common beliefs in the community about high end computing. Second, it is possible that most beliefs are bound by a context. Thus each individual brings to the table a variation of that common belief based upon their own specialized experiences. This could either mean that if we could define the context variables surrounding each lore, we might find small common sets of lore, or it could mean that the contexts are so diverse that each individual represents his or her own lore. It is also possible that we have not sufficiently characterized folklore in our statements, causing confusion in the answers. This could be in the original statements themselves (e.g., not providing sufficient context) or in the negation of the

statements (not truly capturing the inverse of the original statement). In any case, it is clear that in some cases, we have not captured a verifiable folklore and thus need to work on better formulating our hypotheses.

What would be a reasonable percentage of agreement? Can the hypotheses be clearly stated to minimize the variation and offer empirical support for the folklore? These are the issues we are currently working on.

We believe continuing to develop the folklore is of value. Evaluation of the testable hypotheses generated based upon the folklore could lead to a higher degree of consensus and to the creation of a set of empirically supported measures of productivity in HPC domain.

We have also begun to try to understand the nature of defects in high end computing and use some of our methods in generating folklore about development in general to defects in particular. Results at this writing are preliminary but we do have some agreement that shared memory defects are more frequent than any other type of defect. This is the kind of hypothesis we can test in case studies.

Acknowledgements

This research was supported in part by Department of Energy contract DEFG0204ER25633, to the University of Maryland.

5. References

[1] J. Kontio, J., Lehtola, L., and Bragge, J. "Using the Focus Group Method in Software Engineering: Obtaining Practitioner and User Experiences." Proceedings of 2004 International Symposium on Empirical Software Engineering (ISESE'04), (Redondo Beach, CA, 19-20 Aug. 2004), 271-280.

[2] Shull F., Basili V. R., Boehm B., Brown A. W., Costa P., Lindvall M., Port D., Rus I., Tesoriero R., and Zelkowitz M. V., "What We Have Learned About Fighting Defects", Proceedings of 8th International Software Metrics Symposium, Ottawa, Canada, IEEE, June 2002, pp. 249-258.

[3] C.M. Pancake, "Establishing standards for HPC system software and tools", NHSE Review, Nov. 1997.

[4] S. Squires, W. Tichy, L. Votta, "What Do Programmers of Parallel Machines Need? A Survey", Second Workshop on Productivity and Performance in High-End Computing (P-PHEC) , 2005.

[5] J. Dongarra et al., eds. "Sourcebook of Parallel Computing", Morgan Kaufmann, 2003

[6] C. Robson, "Real World Research: A Resource for Social Scientists and Practitioner-Researchers", 2nd Ed. Blackwell Publishers, 2002.

[7] V. Basili, F. McGarry, R. Pajerski, M. Zelkowitz, "Lessons learned from 25 years of process improvement: The rise and fall of the NASA Software Engineering Laboratory", IEEE Computer Society and ACM International Conf. on Soft. Eng., Orlando FL, May 2002, 69-79.

Appendix 1: List of HPC folklore

Initial List	Updated List
[1] Use of Parallel machines is not just for more CPU power, but also for more total memory or total cache (at a given level).	[1] Many people use parallel machines primarily for the large amount of memory available (cache or main).
[2] It's hard to create a parallel language that provides good performance across multiple platforms.	[2] It's hard to create a parallel language that provides good performance across multiple platforms
[3] It's easier to get something working in using a shared memory model than message passing.	[3] It's easier to get something working using a shared memory model than message passing.
[4] It's harder to debug shared memory programs due to race conditions involving shared regions.	[4] Debugging race conditions in shared memory programs is harder than debugging race conditions in message passing programs
[5] Explicit distributed memory programming results in programs that run faster since programmers are forced to think about data distribution (and thus locality) issues.	[5] Explicit distributed memory programming results in programs that run faster than shared memory programs since programmers are forced to think about data distribution (and thus locality) issues
[6] In master/worker parallelism, the master soon becomes the bottleneck and thus systems with a single master will not scale.	[6] In master/worker parallelism, a system with a single master has limited scalability because the master becomes a bottleneck.
[7] Overlapping computation and communication can result in at most a 2x speedup in a program.	[7] In MPI programs, overlapping computation and communication (non-blocking) can result in at most a 2x speedup in a program.
[8] HPF's data distribution process is also useful for SMP systems since it makes programmers think about locality issues.	[8] For large-scale shared memory systems, you can achieve better performance using global arrays with explicit distribution operations than using Open MP.
[9] Parallelization is easy, Performance is hard. For example, identifying parallel tasks in a computation tends to be a lot easier than getting the data decomposition and load balancing right for efficiency and scalability.	[9] Identifying parallelism is hard, but achieving performance is easy.
[10] It's easy to write slow code on fast machines.	[10] It's easy to write slow code on fast machines. Generally, the first parallel implementation of a code is slower than its serial counterpart.
[11] Experts often start with incorrect programs that capture the core computations and data movements. They get these working at high performance first, and then they make the code functionally correct later.	[11] Sometimes, a good approach for developing parallel programs is to program for performance before programming for correctness.
[12] N/A	[12] Given a choice, it's better to write a program with fewer large messages than many small messages