Can Software Engineering Solve the HPCS Problem?

Eugene Loh
Sun Microsystems Inc.
16 Network Circle, UMPK16-303
Menlo Park, CA 94025 USA
+1 831 655-2883
eugene.loh@sun.com

Michael L. Van De Vanter
Sun Microsystems Inc.
16 Network Circle, UMPK16-304
Menlo Park, CA 94025 USA
+1 650 786-8864
michael.vandevanter@sun.com

Lawrence G. Votta
Sun Microsystems Inc.
18 Network Circle, UMPK18-216
Menlo Park, CA 94025 USA
+1 650 786-7514
lawrence.votta@sun.com

ABSTRACT

The High Productivity Computing Systems (HPCS) program seeks a tenfold productivity improvement. Software Engineering has addressed this goal in other domains and identified many important principles that, when aligned with hardware and computer science technologies, do make dramatic improvements in productivity. Do these principles work for the HPC domain?

This case study collects data on the potential benefits of perfective maintenance in which human productivity (programmability, readability, verifiability, maintainability) is paramount. An HPC professional rewrote four FORTRAN77/MPI benchmarks in Fortran 90, removing optimizations (many improving distributed memory performance) and emphasizing clarity.

The code shrank by 5-10x and is significantly easier to read and relate to specifications. Run time performance slowed by about 2x. More studies are needed to confirm that the resulting code is easy to maintain and that the lost performance can be recovered with compiler optimization technologies, run time management techniques and scalable shared memory hardware.

Categories and Subject Descriptors

D.2.0 [**Software Engineering**]. D.1.3 [**Programming Techniques**]: Concurrent Programming –*parallel programming*

Keywords

High Performance Computing, Software Productivity, Software Maintenance, HPCS

1. INTRODUCTION

The High Performance Computing (HPC) community faces a crisis on two fronts: concerns about correctness demand increased verification [16], and programming itself is becoming more complex due to growing problem sizes and the need for massive parallelism. The HPC productivity bottleneck is moving from machines to people, much as it has in other domains, but Software Engineering has had little impact, in no small part by failing to address traditional HPC priorities. Successful application of Software Engineering principles and practices must be grounded in the particular practices and priorities of HPC, and must be substantiated with cost-benefit data.

The case study presented here starts collecting that data. Based on well-known HPC benchmark codes, the study establishes a baseline evaluation of benefits from a perfective maintenance exercise in which manual optimizations are discarded and modern programming language abstractions are exploited for readability.

The outcome is dramatic. Code shrank, in most cases by a factor of 10, and the relationship between code and specification, previously inaccessible, became evident. The former is known to reduce software cost, and the latter is an essential step toward verification. Run time performance typically suffered by a factor of 2, a penalty that may be neutralized by automatic optimization and parallelization at both compile and run time.

These results emphasize the need for empirical studies and for a data-driven evaluation of solutions *in the context of HPC*. Related benefits such as requirements tracing and portability must also be quantified and brought into the cost-benefit equation. The study also casts light on the HPC community's pursuit of performance at the expense of human effort. A credible cost-benefit analysis starts with "modern" implementations, and explicitly includes human productivity [6].

This research is part of the High Productivity Computing Systems (HPCS) program, funded by the Defense Advanced Research Projects Agency (DARPA) to "create new generations of high end programming environments, software tools, architectures, and hardware components" from the perspective of overall productivity rather than unrealistic benchmarks [5].

Section 2 describes the study design. Section 3 presents the results, followed by further analysis in Section 4. Section 5 discusses the findings in the broader context of the program, and mention of related work appears in section 6.

2. THE EXPERIMENT

This is an exercise in perfective maintenance: changing programs in ways that do not affect essential functionality but which improve maintainability [12][17].

SE-HPCS *'05*, May 15, 2005, St. Louis, Missouri, USA.
 1-59593-117-1/05/0005...$5.00.

2.1 Hypothesis

This empirical study has elements of both a quasi-experiment [8] and a case study [19] – the quasi-experiment due to the repeated test (treatment of reordering maintenance and performance goals) of different benchmarks and a case study due to the single professional subject. The use of hypotheses allow us to test and explore the amount of code size reduction and the amount of performance expense. For completeness there are two hypotheses.

H1: Perfective maintenance will not reduce the code size.

H2: Perfective maintenance will not change execution performance.

2.2 Experimental Setting

The subject is a computational physicist with fifteen years of experience in the computing industry, conventionally trained to prioritize performance over maintainability. The experiment was performed using a modern software engineering development environment and workstation for development and a target supercomputer for benchmark execution.

2.3 Experimental Design

The subject revised four parallel benchmarks relevant to the HPC community (external validity): three of the NAS Parallel Benchmarks (NPB) [13][15] and the ASCI sPPM benchmark [10], all written in FORTRAN 77 [2]. This report refers to both the NPB benchmark specifications, which date back to the NPB1 release, and the NPB serial and parallel (MPI) implementations, which are part of the NPB2.3 release [13].

The goal of the perfective maintenance task was to prioritize maintainability over performance:

- remove specialized code for distributed memory;
- remove source level optimizations;
- use abstractions provided by Fortran 90 [7], a modern superset of FORTRAN 77; and
- remove code known to be not portable.

The dependent variables are code size and execution performance on the target system.

The construct validity threats are summarized as: do we think what and how we are changing and removing from the code accomplishes the priority shift? Clearly, the treatment accomplishes much of the priority shift, and the issue is one of how much more could be accomplished. The results in the next section allay much of this concern.

The internal validity addresses the concern of undetected influences on the dependent variables code size and execution performance. Inspection of the code and executing benchmark test limit the possibility and magnitude of any influences.

Finally, the external validity threats are effects that limit our ability to generalize the experiment. Specifically, we are studying benchmarks here – kernels of computation – to expect the ratio of size reduction to execution performance is not relevant and thus not a threat. We would like to generalize to the kernels of major computational codes. How the benchmarks are created, calibrated and maintain address this validity threat.

3. DATA

Table 1 summarizes quantitative results of the experiment, expressed both as reduction in lines of code (LOC) and run time performance penalty; measurement details appear in Appendix A. Each benchmark is discussed separately below, followed by comments on qualitative results. The experiments appear in the order they were conducted. Some learning about Fortran 90 and computational fluid dynamics occurred during the experiments.

Table 1: Summary of Results

Benchmark	LOC before	LOC after	Size reduction	Slowdown
NAS BT	3687	484	8x	3x
NAS MG	1701	150	11x	2x?/6x?
NAS CG	839	81	4x/10x	≥2x
ASCI sPPM	13606	1358	10x	2x

Approximately 2x of the size reduction was due to removal of explicit parallelism, a result consistent with other studies of the size contribution of MPI [4][9].

3.1 NAS BT Benchmark

BT is a "synthetic application," representative of applications involving computational fluid dynamics. The subject had general familiarity with NAS Parallel Benchmarks, but none with BT. The experiment took approximately two weeks of full time effort.

Table 2 summarizes code size changes between the reference implementation and revised code. These summaries are approximate, given the difficulty of precise accounting.

Table 2: NAS BT Lines of Code

Description	LOC		
	NPB (MPI)	NPB (serial)	Revised
Global declarations	269	246	19
Main program	105	74	54
Initialize	201	148	19
Time step & solve	1165	596	62
Exact solution	13	13	35
Compute dU/dt	625	596	82
Compute left-hand sides	717	716	102
Self test	247	228	111
Inter-process comm..	345	0	0
Total	**3687**	**2617**	**484**

Reductions varied considerably, for example:

- Global declarations shrank for three reasons: removal of confusing intermediate constants (manual optimizations); adoption of Fortran 90 array syntax and module support; and removal of variables that didn't need to be global.
- The main program and self test contain code for which little improvement was possible.
- Initialize was reduced by relocating some code, removing unnecessary code (previously hard to detect because of code complexity), and adopting Fortran 90 array syntax.
- Time step & solve showed the greatest reduction: removing hand optimizations and exploiting Fortran 90 array syntax and MATMUL. As the code clarified, comments (not counted in the LOC results) also shrank.
- Exact solution acquired related bits of code from elsewhere.

- Time differential dU/dt shrank, not only by exploiting Fortran 90 features (bringing the code in much closer alignment with the specification), but also by detection and removal of approximately 300 lines of redundant code.

One of the most striking results was a dramatic increase in correspondence between specification and code (section 3.6).

The revised code exhibited a 2.7x slowdown on the class W data set (one of several specified as part of the NAS Parallel Benchmarks) when compared to the original implementation.

3.2 NAS MG Benchmark

MG is a "kernel" benchmark, intended to characterize multigrid computations. It is much shorter than BT: the specification is 2.5 pages of PDF, in contrast to 30 pages for BT. The subject, who had prior implementation experience with the benchmark, set out to simplify the NPB reference code, but eventually rewrote it from scratch in a few hours. The experiment took about one working day over a period of a week. Table 3 summarizes changes in code size between the reference implementation and the revised code.

Table 3: NAS MG Lines of Code

Description	LOC		
	NPB (MPI)	NPB (serial)	Revised
Global declarations	80	89	0
Main program	201	160	35
Initialize v	202	213	35
Operators	281	277	80
Communications	665	144	0
Other	272	124	0
Total	**1701**	**1007**	**150**

Code reductions were similar to those for the BT benchmark, and the result was similarly more concise and readable.

The revised code performs poorly, largely due to a compiler problem (believed fixable) with stencil performance for array syntax. It appears that the code autoparallelizes reasonably well, at least to conventional scales. Details appear the full report [11].

3.3 NAS CG Benchmark

CG is another "kernel" benchmark, approximately as complex as MG and much simpler than BT. It is intended to characterize conjugate gradient computations. The subject had prior implementation experience with the benchmark. The experiment took place over two days with most of the effort going to simplifying the NPB code and working with sparse matrices. Table 4 summarizes changes in code size between two of the reference implementations and the revised code.

Table 4: NAS CG Lines of Code

Description	LOC		
	NPB (MPI)	NPB (serial)	Revised
Core functionality	839	309	81
Data fabrication	197	197	158
Total	**1036**	**506**	**239**

The relative compactness of the derived code derives from much the same phenomena reported for the BT and MG benchmarks.

A significant portion of the perfective maintenance effort (and the resulting code) for CG involved reverse engineering an arcane data fabrication algorithm used in the original code. The algorithm, which is not fully determined in the specification, must be reproduced exactly in order to satisfy correctness checks. Using other reasonable algorithms for fabricating matrix data would permit overall code reduction closer to 10x.

The revised code autoparallelizes well. More details on scalability appear in the full report [11].

3.4 ASCI sPPM Benchmark

sPPM is a computational fluid dynamics (CFD) benchmark that uses the "simplified" Piecewise Parabolic Method (sPPM). Its performance targets were 1 teraflops and beyond for procurements within the Accelerated Strategic Computing Initiative (ASCI) [1].

Although not a "real application," sPPM is the most complex of the four codes studied. The shock-wave physics is handled with fairly involved numerical algorithms. Performance optimizations include cache blocking, vectorization, multi-thread and multi-process parallelism, overlapping communication and computation, and dynamically scheduling work.

The subject was initially unfamiliar with the sPPM code and with CFD. The experiment was part-time for less than one month. Table 5 summarizes changes in code size between reference implementation and revised code. These line counts include makefiles, input decks, and so on, but these are relatively small.

Table 5: sPPM Lines of Code

Description	LOC	
	ASCI	Revised
Main program	2486	484
Shock dynamics	2742	616
Boundaries	3431	22
Time stepping	2312	75
Global declarations	238	25
Some C I/O functions	366	54
Timers	49	17
Multi-platform threads support	1500	0
Makefile	407	21
Run script	57	23
Input deck	7	10
Reference output	11	11
TOTAL	**13606**	**1358**

Some reasons for code reduction have already been observed for the other benchmarks. Additionally:

- sPPM strives for functional and performance portability -- for example, targeting superscalars, vectors, etc.
- sPPM suffers complexity from trying to maintain small access strides in memory for best performance.
- A great deal of the I/O is unnecessarily complicated.

- There was a great deal of support for legacy threads, which was made obsolete by adoption of OpenMP [14].
- Sophisticated schemes overlapped communication and computation, something that would be best left to platform infrastructure in an idealized world.
- Routines were replicated with special case optimizations.

Remarkably, given the extensive simplification of the code, the revised version on a single CPU ran only 2.1x slower on a test problem than the reference implementation. The cause of the slowdown requires investigation, but is likely due, at least in part, to the large memory strides in the revised version.

The challenge for computing systems is to achieve automatically the impressive scalability achieved by extensive manual techniques used in this reference implementation. There are reasons to be hopeful, including the large problem sizes that are needed to study multi-scale physics in 3d CFD and the higher-level expression that results when the source code is improved, but demonstration of such automatic scalability remains the subject of future work.

3.5 Manual Optimizations

The subject reported severe difficulty understanding some of the manually optimized code:

- global definition of many intermediate constants in an attempt to identify common subexpressions;
- manually unrolled loops; and
- functions expanded in-line, for example for derivatives.

Some of those could actually be counterproductive in today's computational environments. All optimizations confounded the relationship between specification and code.

3.6 Expressiveness and Fortran 90

Language features new to Fortran 90 enable code that is both shorter and expressed more directly in terms of the problem. For example, the timed portion of the NAS MG reference implementation appears in Figure 1:

```
call resid(u,v,r,n1,n2,n3,a,k)
callnorm2u3(r,n1,n2,n3,rnm2,rnmu,nx(lt),ny(lt),nz(lt))
old2 = rnm2
oldu = rnmu
do  it=1,nit
    call mg3P(u,v,r,a,c,n1,n2,n3,k)
    call resid(u,v,r,n1,n2,n3,a,k)
enddo
call norm2u3(r,n1,n2,n3,rnm2,rnmu,nx(lt),ny(lt),nz(lt))
```

Figure 1: NAS MG timed portion (original FORTRAN 77)

This code excerpt implements the portion of the specification appearing in Figure 2, and the revised version appears in Figure 3.

Each of the four iterations consists of the following two steps,

r = v - A u (evaluate residual)

u = u + Mk r (apply correction)

...

Start the clock before evaluating the residual for the first time, ... Stop the clock after evaluating the norm of the final residua.l

Figure 2: NAS MG timed portion (spec.)

```
do iter = 1, niter
  r = v - A(u)    ! evaluate residual
  u = u + M(r)    ! apply correction
enddo
r = v - A(u)    ! evaluate residual
L2norm = sqrt(sum(r*r)/size(r))
```

Figure 3: NAS MG timed portion (revised Fortran 90)

The evident correspondence between the two promises greater success and lower cost for code verification and maintenance.

4. ANALYSIS

The benefit (average reduction of LOC) and the cost (the average increase in execution time performance) are calculated simply as the ratio of the pretreatment quantity (LOC, execution time) divided by the post treatment quantity (LOC, execution time). Thus for the NAS BT benchmark we have 3687/4849 = 7.62 ~8. For execution time, we use the convention that the factor is always greater than 1 with the description of slower or faster to indicate more or less execution time.

5. DISCUSSION

The central result of this study, namely that HPC code can be both shrunk tenfold and dramatically clarified at the expense of a twofold performance penalty, suggests rethinking the balance between man and machine. A number of technologies offer the prospect of recovering some of the cost of the performance loss.

There are many benefits. The lifetime costs of software are known to correlate highly, and linearly, with code size. Also, the kind of code produced in this case study is much more likely to be portable, a significant factor in the lifetime cost of HPC software. Finally, the increasing importance of verification in HPC software will be well served by code that is not only smaller, but also dramatically easier to understand in relationship to the problem specifications. Experiments are needed to assess these effects more precisely.

The study validates some of the design goals for Fortran 90: high quality results were possible, levels of effort were moderate, and maintenance could often be done gradually with frequent regression testing. Other modern languages for HPC might offer similar, or better, benefits, but the costs of learning and conversion must be a necessary part of the analysis.

6. RELATED WORK

Other studies have shown significant code reduction from distributed to shared memory implementations: approximately 2x for the NAS MG benchmark [4] and an average of 1.77 from MPI to serial implementations over 8 benchmarks [9].

These results are broadly consistent with Weinberg's studies on the cost of multiple goals: "Optimization goals tend to be highly conflicting with other goals" [18]. Goals are also constraints; this case study can be seen as removing constraints on software that derive from the "accidental" rather the "essential" nature of the task, in the terminology of Fred Brooks (following Aristotle) [3]. The "accidental" in this case includes the limitations of FORTRAN 77, the demand for utmost performance, and confounding platform architectures.

7. CONCLUSIONS

At the cost of a relatively modest performance penalty at run-time, HPC software written in FORTRAN 77 can be improved through perfective maintenance with dramatic reduction in human cost (across the entire software life cycle) and reduce the growing cost of verification.

This cost-benefit equation must be explored further, not only with investigation into performance improvements, but also by expanding the scope of the data across more HPC professionals and more kinds of HPC code. This empirical data is needed to support the kind of credible analysis the HPC community will expect in order to evaluate solutions to the HPCS problem.

8. ACKNOWLEDGMENTS

We would like to thank our HPCS colleagues at Sun Microsystems and elsewhere in the HPC community for their helpful discussions and comments.

This material is based upon work supported by DARPA under Contract No.NBCH3039002.

9. REFERENCES

[1] The Accelerated Strategic Computing Initiative (ASCI), now known as Advanced Simulation and Computing (ASC) <http://www.nnsa.doe.gov/asc/>.

[2] American National Standards Institute *American National Standard Programming Language FORTRAN*. ANSI X3.9-1978, New York, NY, 1978.

[3] Brooks, F. P. Jr., No silver bullet: essence and accidents of software engineering. *Computer* 20,4 (April 1987) 10-19.

[4] Chamberlain, B. L., Deitz, S. J., and Snyder, L. A comparative study of the NAS MG benchmark across parallel languages and architectures. *Proceedings of the ACM Conference on Supercomputing* (2000).

[5] Defense Advanced Research Project Agency (DARPA) Information Processing Technology Office, High Productivity Computing Systems (HPCS) Program. <http://www.darpa.mil/ipto/programs/hpcs/>.

[6] Gustafson, J. Purpose-Based Benchmarks. *International Journal of High Performance Computing Applications: Special Issue on HPC Productivity, 18*,4 (November 2004).

[7] ISO/IEC *International Standard ISO/IEC 1539-1:1997(E) Information Technology - Programming Languages – Fortran*. Geneva, Switzerland, 1997.

[8] Judd, C. M., Smith, E. R., and Kidder, L. H. *Research Methods in Social Relations* Holt, Rinehart and Winston, Inc., sixth edition, 1991.

[9] Kepner, J., HPC Productivity Model Synthesis. *International Journal of High Performance Computing Applications: Special Issue on HPC Productivity 18*,4 (November 2004).

[10] LLNL *The ASCI sPPM Benchmark Code*. Lawrence Livermore National Laboratory, <http://www.llnl.gov/asci/purple/benchmarks/limited/sppm/>

[11] Loh, E. Van De Vanter, M.L, and Votta, L. G. *Can Software Engineering Solve the HPCS Problem*, Sun Microsystems Laboratories Technical Report, 2005 (in preparation).

[12] Mockus, A. and Votta, L.G. Identifying Reasons for Software Changes Using Historical Databases. *Proceedings of the International Conference on Software Maintenance – ICSM2000*, San Jose, California (October 2000) 120-130.

[13] NASA *The NAS Parallel Benchmarks (NPB)*. NASA Advanced Supercomputing Division, <http://www.nas.nasa.gov/Software/NPB/index.html>.

[14] OpenMP, <http://www.openmp.org/>.

[15] Saphir, W. C., et al. New implementations and results for the NAS Parallel Benchmarks 2. *8th SIAM Conference on Parallel Processing for Scientific Computing*, Minneapolis, MN, (March 14-17, 1997).

[16] Post. D. E., and Votta, L. G. Computational Science Requires a New Paradigm. *Physics Today*, **58**(1): p. 35-41.

[17] Swanson, E.B. The Dimensions of Maintenance. *Proceedings of the 2nd International Conference on Software Engineering*, San Francisco, California (1976) 492 – 497.

[18] Weinberg, G. M., Schulman, E. L., Goals and Performance in Computer Programming. *HUMAN FACTORS 16*,1 (1974) 70-77.

[19] Yin, R. K. *Case Study Research: Design and Methods*. Sage Publications, second edition, 1994.

APPENDIX A

Comments and blank lines are excluded from line counts (LOC). Performance was measured on a Sun Fire 6800 server with 48 Gbyte of memory and 24 UltraSPARC III+ CPUs at 900 MHz with 8 Mbyte of L2 cache each. The Fortran 95 compiler from the Sun ONE Studio 8 Compiler Suite was used with typical switches:

- -fast (common performance-oriented switches)
- -xarch=v9b (UltraSPARC-III settings, for 64-bit binaries)
- -parallel –reduction (in cases where autoparallelization was used)