Proceedings of the Second Workshop on Productivity and Performance in High-End Computing (PPHEC-05)

February 13, 2005
San Francisco, USA

Held in conjunction with the
Eleventh International Symposium on
High Performance Computer Architecture

2^{nd} Workshop on Productivity and Performance in High-End Computing
Febryary 13, 2005
San Francisco, USA

Proceedings of the Second Workshop on Productivity and Performance in High-End Computing (PPHEC-05)

Keynote Address

Mitigating the Risks Faced by Large-scale Computational Science

Douglass E. Post
Los Alamos National Laboratory

Mitigating the risks faced by large-scale computational science.

D. E. Post, Los Alamos National Laboratory
Keynote Address for the Second Workshop on Productivity and Performance in High-End Computing (P-PHEC), 11th Intl. Symposium on High-Performance Computer Architecture (HPCA-2005) San Francisco, CA, February 13, 2005

Large-scale computational science faces at least three major risks: the "Performance Risk"—the risk that the power of high performance computers will not continue to grow exponentially, the "Programming Risk"—the risk that it will become too difficult to develop useful programs for these highly complex high performance computers, and the "Prediction Risk"—the risk that the large-scale computer simulations and applications we develop to exploit this new computer power will not provide reliable and credible results.

Larger scale computers are being developed by many vendors. While the exponential increase in single processor speed has begun to saturate, massive parallelization, faster interconnects and new architectures are sustaining the growth of computer power. Computers with processing powers of over 40 Tera-Flop/s now exist, 300 Tera-Flop/s computers are planned for 2006-2007, and Peta-Flop/s class computers are being designed for 2010. Platform architectures are proliferating: LINUX-based commodity clusters, Vector architectures and proprietary clusters of shared memory processor nodes with and without rapid interconnects. Although there is some risk that one or more of these approaches will fail, it is almost certain that more than one will be successful.

The increased complexity of computer architectures that is largely responsible for much of the growth in computer power is the major cause of the Programming Risk. Developing high performance computer applications for these platforms requires attention to data communication among the processors and shared memory nodes, almost at the level of assembly language. Debugging, data visualization and analysis, and performance optimization are difficult. The software tools are generally immature and often short-lived. The typical code employs many different languages (Fortran, C, C++, Python, PERL, Unix Scripts, etc.). The high performance computing community must continue to develop better tools and methods for code development and performance optimization, and improved computational mathematical algorithms appropriate for the new computer architectures.

Computational science has the promise of making major contributions to science and society by enabling us to solve many highly complex and non-linear problems with an unprecedented degree of realism and fidelity. The Prediction Risk is that these large-scale, very complicated applications will not give correct and reliable answers to the questions they are designed to address. Many present codes have too many defects, and many of the codes are based on models that are either incomplete or inaccurate representations of the real world. Case studies of many such codes indicate that much more attention must be paid to sound software project management. These studies also show that present methods for verification and validation are inadequate, and that improved methods are urgently needed.

The DARPA High Productivity Computing Systems Program is designed to help minimize these risks and reduce the time to solution by exploring a number of approaches for the development of a Peta-flop/s class computer, by developing tools and benchmarks for high performance computers, and by conducting case studies of existing high performance computer applications to characterize those projects and develop the "lessons learned".

INFLUENCE OF WORKLOAD CHARACTERIZATION ON DoD HIGH PERFORMANCE COMPUTING MODERNIZATION PROGRAM ACQUISITIONS

Roy L. Campbell, Jr.
U.S. Army Research Laboratory
rcampbell@arl.army.mil

Larry P. Davis
DoD HPC Modernization Program Office
larryd@hpcmo.hpc.mil

Abstract

Each year the DoD High Performance Computing Program (HPCMP) conducts a mathematically rigorous assessment of bid systems with workload being at the cornerstone of its analysis. Workload characterization is a principal consideration in determining (1) which benchmarking codes are included in the HPCMP suite, (2) what test cases are constructed for each code (in terms of general size, since the problem size is fixed for each test case), (3) what attributes are incorporated for each test case (e.g., memory, I/O, and relative compute-time requirements), and (4) what percentage of the HPCMP workload is assigned to each test case (to be treated as a target metric for price per performance optimization). This paper provides an in-depth discussion on how workload characterization influences each of these determinations.

1. Introduction.

Over a period of five years, the HPCMP has crafted its annual acquisition process to objectively and mathematically assess bid systems. The process begins with the careful selection of application codes to represent the program's ten Computation Technology Areas (CTAs): Computational Structural Mechanics (CSM), Computational Fluid Dynamics (CFD), Computational Chemistry and Materials Science (CCM), Computational Electromagnetics and Acoustics (CEA), Climate, Weather, and Ocean Modeling and Simulation (CWO), Signal/Image Processing (SIP), Forces Modeling and Simulation/C4I (FMS), Environmental Quality Modeling and Simulation (EQM), Computational Electronics and Nanoelectronics (CEN), and Integrated Modeling and Test Environments (IMT). Next, input decks are constructed to create representative test cases in terms of parallel environment size, memory density (i.e., amount of memory used per processor), I/O intensity, and overall time-to-solution. Typically, a standard and large test case are generated for each code such that the standard test case can be solved in 30 minutes using a small number of processors (e.g., 64) and the large test case can be solved in one hour using a large number of processors (e.g., 384), both using the DoD's standard system, which is determined annually.

The application test cases along with a set of synthetic probes are sent to vendors for execution on each bid system (or a similar system, assuming the vendor is willing to provide binding forecasts for the bid system). Application test case and synthetic results are then submitted to the HPCMP for assessment. Individual application test case scores and an overall synthetic score are calculated per system. Non-recurring (i.e., capital investment) and recurring (i.e., maintenance, power, and specialized administration) costs are also submitted to the HPCMP for inclusion in a price per performance assessment. The underlying mathematical model takes into account application and synthetic scores for both the bid systems and the HPCMP systems in production, bid prices, and a carefully forecasted workload distribution among application test cases, culminating in a linear programming problem that often possesses a unique and optimal solution (i.e., an identification of what systems to buy along with a description of how work should be distributed among bid and production systems such that the overall workload distribution is satisfied within a certain tolerance). The workload distribution is a mapping of recorded usage (past), granted system allocation (present), and submitted user requirements (future) onto the set of application test cases, and is deduced by circumspectly accounting for the CTA and problem size characterization of each data source (past, present, and future).

Two objective recommendations are then prepared for decision makers – one based on price per performance results and the other based on pure performance. The former assumes work will be distributed among bid and production systems according to the results of the mathematical model (i.e., the distribution of work can vary dramatically from system to system as long as the overall distribution of work coincides with the target workload distribution within a given tolerance). The latter assumes work will be distributed across each bid and production system strictly according to the target workload distribution (i.e., the distribution of work is exactly the same for each system).

Therefore, an accurate understanding of the workload is paramount to the annual HPCMP acquisition assessment, as the careful (1) selection of codes, (2) development of test cases, and (3) deduction of target application test case percentages are critical to the production of accurate and objective acquisition recommendations.

2. Determination of Codes.

CTA leaders and annual requirements surveys provide an initial list of codes (Table 1) that are representative of work being performed within each computational discipline. Then, a survey of the top 100 projects (in terms of CPU-hours) is conducted to determine how relevant each identified code is to the HPCMP workload. (A survey response summary is provided in Table 2.) Relevance is determined in two ways. First, a list of all codes reported by the top 100 is compiled and ranked according to a percentage of the total expended CPU-hours reported (Table 3). Second, a best-fit mapping of the reported codes onto those in the initial list is conducted. Codes in the initial list are then ranked according to a mapped percentage of the total expended CPU hours reported (Table 4). Finally, the initial list is pruned based on the results of these two methods to yield seven or eight of the most representative codes (Table 5).

Table 1: Initial list of codes.

AERO	CFD	Used in previous year
ALEGRA	CSM	Potential addition
AVUS	CFD	Used in previous year
CTH	CSM	Used in previous year
GAMESS	CCM	Used in previous year
HYCOM	CWO	Used in previous year
NAMD	CCM	Used in previous year
OVERFLOW-2	CFD	Potential addition
SWITCH	CEA	Used in previous year
WRF	CWO	Potential addition

Table 2: Response rate to top 100 projects survey.

SET	RESPONSE
Top 100	42% (42)
Top 20	70% (14)
Top 10	90% (9)
Top 5	100% (5)
Total FY-03 Hours	57.7% (46,844,992)

Table 3: Actual use ranking.

HYCOM	CWO	14.0%
CTH	CSM	9.1%
AVUS	CFD	5.7%
GAMESS	CCM	1.1%
ALEGRA	CSM	0.5%
WRF	CWO	0.4%
AERO	CFD	0.0%
NAMD	CCM	0.0%
OVERFLOW-2	CFD	0.0%
SWITCH	CEA	0.0%

Table 4: Representative use ranking.

HYCOM	CWO	19.6%
AVUS	CFD	16.0%
CTH	CSM	15.7%
GAMESS	CCM	14.0%
OVERFLOW-2	CFD	12.7%
NAMD	CCM	4.6%
WRF	CWO	0.9%
ALEGRA	CSM	0.8%
SWITCH	CEA	0.4%
AERO	CFD	0.2%

Based on the information in Tables 3 and 4, one might concluded that the initial set of ten codes should be reduced by removing AERO (FDL3DI), SWITCH, ALEGRA, and/or WRF; however, of these four, only one was removed – ALEGRA. AERO is a serial (i.e., a single processor) code and, therefore, was spared since a notable portion of the HPCMP workload uses only one processor. (Serial examples include grid generation, pre/post-processing of input/output data, single processor interactive sessions, and multiple single processor executions launched within a parallel environment). SWITCH survived since it was the only CEA code in the list. WRF was retained since its future use within the CWO community was highly likely. Therefore, only one code of the bottom four was actually cut – ALEGRA.

Since the target size of the suite is seven to eight codes, NAMD was examined next. Given that the chemistry area within CCM contains two major thrusts – molecular dynamics (NAMD) and quantum chemistry (GAMESS), both NAMD and GAMESS were compared for relative relevance. Since, at the time, very little molecular dynamics work was being performed within the program, NAMD was removed, although it is expected that NAMD will be reinstated in subsequent suites. Table 5 lists the codes selected for the final benchmarking suite.

Table 5: New benchmarking suite.

AERO	CFD
AVUS	CFD
CTH	CSM
GAMESS	CCM
HYCOM	CWO
OVERFLOW-2	CFD
SWITCH	CEA
WRF	CWO

3. Determination of Application Test Cases.

To determine what application test case sizes should be considered, the job records for an entire fiscal year are classified according to the system fraction used by each job. For each system sixteenth, the number of GFLOP/s-years (Equation 1) is accumulated, and a percentage of the total usage is calculated, yielding a histogram of usage versus job size in terms of system fraction.

$$U_{GFLOP/s-years} = P_{speed} \cdot P_{instruction\_rate} \cdot N \cdot T_{years} \qquad (1)$$

where

- $U_{GFLOP/s\text{-}years}$=usage in GFLOP/s-years
- P_{speed}=processor speed
- $P_{instruction\_rate}$=number of instructions per cycle
- N=number of processors used by job
- T=time-to-solution of job in years

For fiscal year 2003, the usage characterization (Figure 1) demonstrated a bi-modal behavior that lead the HPCMP to adopt (or actually reconfirm) a policy of having a large and standard test case for each benchmarking code, in general.

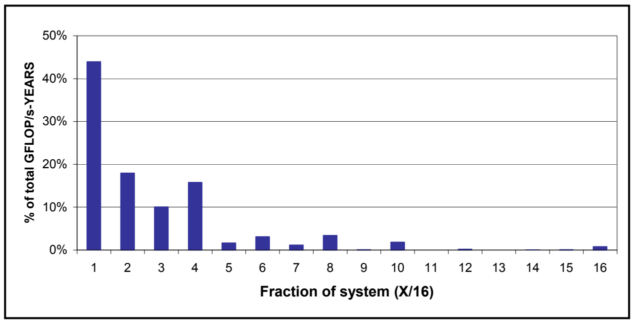


Figure 1: FY-03 usage versus system fraction.

Only two codes were granted exceptions to the rule. AERO, a serial code, was not conducive to building a large test case, and WRF proved to be an exceptionally challenging code, so for sake of time, only a small test case was developed.

Based on the location of the two major peaks in Figure 1, the target times for each standard and large test case correspond to the time-to-solution for one-sixteenth and one-fourth of the standard DoD system, respectively. This general rule is, of course, not applied to AERO.

4. Determination of Application Test Case Weights.

The utilization for the previous year, allocation for the current year, and requirements for the upcoming year are deduced per CTA. Then, a linear combination of the three is carefully mapped onto each application test case, yielding a target percentage of the workload for each. Since current and future workload characterizations contain proprietary data, only the past year's usage data is provided below (Figure 2).

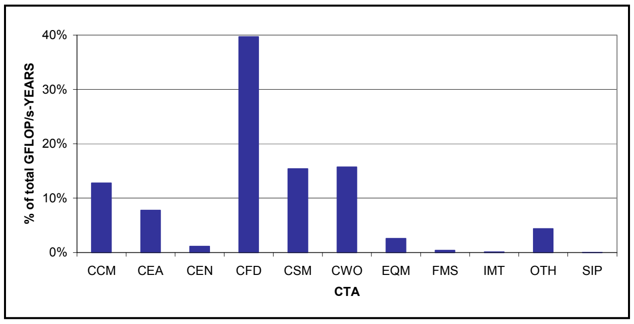


Figure 2: FY-03 usage in GFLOP/s-years per CTA.

Five CTAs (CCM, CEA, CFD, CSM, and CWO) dominate the plot above, hence the selection of codes in Table 5 that cover only these five computational disciplines. Three CTAs (CFD, CSM, and CWO) individually correspond to more than 15% of the HPCMP workload, hence the selection of multiple codes for each: CFD – AERO, AVUS, and OVERFLOW-2; CSM – CTH and GAMESS; and CWO – HYCOM and WRF.

5. Conclusions.

Given the multitude of high-end computing (HEC) attributes (e.g., bandwidths and latencies of the memory, communication, and I/O substructures, instruction retirement rate, register set structure, and processor speed), a sound assessment of bid systems requires a careful and accurate assessment of the targeted workload, since the ranking of systems with respect to these attributes varies dramatically per application. The HPCMP has, therefore, placed workload characterization at the cornerstone of its annual acquisition process.

6. Future Work.

To increase the accuracy of the HPCMP's workload characterization, automated job profiling tools (i.e., background software that automatically captures key attributes of each executed job) are being investigated. Work performed by the National Energy Scientific Computing Center is currently being reviewed as a potential source for this automation.

7. References.

[1] F.T. Tracy et al., "A survey of the algorithms in the TI-03 application benchmarking suite with emphasis on linear system solvers", IEEE Proceedings of the 2003 Users Group Conference, pp. 332-336, June 2003.

What Do Programmers of Parallel Machines Need? A Survey

Larry Votta and Susan Squires, Sun Microsystems
Walter Tichy, University Karlsruhe

Paper to be handed out at workshop

Toward The Automated Generation of Components from Existing Source Code

Daniel Quinlan
Qing Yi
Gary Kumfert
Thomas Epperly
Tamara Dahlgren
Center for Applied Scientific Computing
Lawrence Livermore National Laboratory
P. O. Box 808, Livermore CA, 94551 USA
{dquinlan,yi4,kumfert,tepperly,dahlgren}@llnl.gov

Markus Schordan
Institute of Computer Languages
Vienna University of Technology
Argentinierstrasse 8/4/13, A-1040 Vienna, Austria
markus@complang.tuwien.ac.at

Brian White
Computer Systems Laboratory
Cornell University, Frank H. T. Rhodes Hall
Ithaca, NY 14853, USA
bwhite@csl.cornell.edu

Abstract

A major challenge to achieving widespread use of software component technology in scientific computing is an effective migration strategy for existing, or legacy, source code. This paper describes initial work and challenges in automating the identification and generation of components using the ROSE compiler infrastructure and the Babel language interoperability tool. Babel enables calling interfaces expressed in the Scientific Interface Definition Language (SIDL) to be implemented in, and called from, an arbitrary combination of supported languages. ROSE is used to build specialized source-to-source translators that (1) extract a SIDL interface specification from information implicit in existing C++ source code and (2) transform Babel's output to include dispatches to the legacy code.

1 Introduction

Contemporary multi-disciplinary, multi-scale, multiphysics simulations are increasingly becoming large, composite applications consisting of new and existing components implemented in different programming languages by disparate teams. These factors present several challenges to developers of such systems that, if dealt with manually, are time-consuming and error prone. This paper addresses an automation technology for the extraction and implementation of components within the context of a component architecture tailored for scientific computing.

Component technology is industry's answer to at least two of the three major concerns plaguing large-scale componentization efforts; namely, interoperability of software written in different languages, interoperability of software running on different platforms, and maintenance and evolution of large composite systems with multiple third party dependencies. Component architectures from industry include CORBA [3], Microsoft COM [14], and Sun's Enterprise Java Beans (EJB) [5]. These architectures establish the framework in which compliant components interact. For instance, EJB assumes all components are implemented in Java (or JNI), thereby leaving the language interoperability issue to component developers to address. Unlike commercial applications, large scale numerical simulations have additional constraints unique to the scientific computing domain such as high performance, a wide variety of often one-of-a-kind computing platforms, and a need to migrate a substantial body of code to a new programming paradigm.

The Common Component Architecture (CCA) Forum [1] is working to deliver component technology suitable for large scale numerical simulations. Babel provides the Scientific Interface Definition Language (SIDL) and associated language interoperability tools that undergirds CCA-compliant frameworks. Current best practice for migrating legacy source code to CCA components does not require modifying existing code but may involve substantially rethinking the interfaces and writing additional code by hand that bridges Babel's language bindings to the legacy code.

In this paper we address the automated generation of the bridging code between C/C++ libraries and their CCA-compliant component wrappers. The work employs the ROSE compiler tools to build specialized translators that use a library's header files to generate the SIDL file and bridging code required by the CCA framework. Such an automated approach is critical to converting legacy codes to components in sufficient numbers to achieve the economy of scale that makes component technology so effective in other domains. Currently our work is limited to C and C++ libraries. Our ongoing work includes adding a FORTRAN90 frontend to the ROSE infrastructure, which will enable us to apply similar techniques to convert FORTRAN libraries to components in the future.

The major technical challenges of automatically mapping libraries to SIDL code come from two aspects. First, to allow all-to-all interoperability among its supported languages, Babel defines SIDL with a narrow intermediate type-system and inheritance model. The ROSE translators thus must extract the necessary (often implicit) information embedded in the library and map the C/C++ type system into one expressible in a proper SIDL file. Second, programming in the CCA component model has a more event-driven and less imperative feel than traditional programming due to its focus on services. For instance, a CCA component rarely explicitly creates all the lower-level components it depends on to function. Instead, it typically registers what capabilities it provides and which capabilities it depends on to the CCA framework in response to an event, which usually takes the form of a creation request, but can also be a connection or disconnection request. Then the CCA component typically is inactive until one of its provided capabilities is invoked.

Our present work has addressed the first challenge by translating C++ types into equivalent SIDL types when possible, and conservatively using the *opaque* type in SIDL (indicating no information is known about the type) if no suitable translation is available. The work to address the second challenge is still ongoing. Specifically, our future work will include techniques to automatically cluster global functions and classes into different components (our current implementation simply places all global types in a library into a single component). Further, we will automate the generation of CCA components, which are independent units of composition that implement the $gov.cca.Component$ interface. More details are provided in Section 3.4.

Although C and C++ are only two of the modern languages supported by Babel and CCA, the their type system represents one of the most complex and comprehensive type systems in existing statically-typed languages. For example, generating SIDL specifications for a C/C++ library requires the translation of overloaded functions and operators, classes with multiple inheritance, C++ templates, function pointers, and variable number of arguments, many of which can be ignored when translating smaller languages such as Java and FORTRAN, which do not have multiple inheritance or C++ templates. We thus expect that many techniques we develop for translating C/C++ libraries to CCA components will similarly apply to Java and FORTRAN as well. Further, much of the design principles we developed are language independent, and can apply in general to all modern programming languages.

2 Infrastructure

Our component generation infrastructure includes both ROSE [21, 24] and Babel [16]. ROSE is a compiler infrastructure that offers mechanisms for analyzing C++ source code and for building source-to-source translators, which in this paper are used to process library code and generate component implementations. Babel is an Interface Definition Language (IDL)-based language interoperability tool akin to CORBA but tailored for the scientific computing community. In the following two sections we describe ROSE mechanisms in simplifying the development of translators and Babel capabilities in aiding the generation of components.

2.1 ROSE

The ROSE infrastructure allows building source-to-source translators by offering a front-end for parsing C++ code and generating an Abstract Syntax Tree (AST), a mid-end for restructuring the AST representation of the source code, and a back-end to unparse C++ source code from the AST.

We use the Edison Design Group (EDG) C++ front-end [2] to parse C++ programs. After invoking the EDG parser on an input C++ program, we then translate the C-style EDG internal representation of the program into an object-oriented abstract syntax tree (AST), Sage III, which we have developed as a revision of the Sage II [13] intermediate representation. Current work includes collaboration with Rice to add F90 support to ROSE through use of the Open64 compiler infrastructure.

The mid-end supports restructuring of the Sage III AST. The programmer can add code to the AST by specifying a source string using C++ syntax, or by manually constructing subtrees of the AST. A program transformation consists of any required program analysis and a series of AST restructuring operations each of which specifies a location in the AST where a code fragment should be inserted, deleted, or replaced.

The back-end unparses the AST and generates C++ source code. Header files can either be unparsed where they are included in source files, or `#include` directives can be generated for the header files. This feature is important when transforming user-defined data types, for example, when adding compiler-generated methods.

2.2 Babel

Compared with other IDL technologies, Babel/SIDL has several features critical for scientific computing such as intrinsic support for dynamically allocated, arbitrarily strided multidimensional arrays and complex numbers. It also supports overloading method names. Whereas CORBA emphasized remote method invocation, Babel emphasizes fast, in-process language interoperability [12]. Babel also has extensive FORTRAN 77/90/95 support, even allowing Babel arrays to be manipulated as native FORTRAN 90 arrays [16]. The Babel team is currently developing remote method invocation (RMI) capabilities.

Babel includes two parts: the code generator and the runtime library. The code generator parses SIDL and generates client and/or server bindings in C, C++, FORTRAN 77, FORTRAN 90, Python, and Java. The runtime library contains base classes of the object model which are themselves defined in SIDL and additionally, bits and pieces needed to enhance portability and support interoperability.

IDLs are fundamentally different than typical programming languages. IDLs define types without providing code to implement them. Often, IDL resembles stripped down C++ header files. In SIDL (scientific IDL), users can define new types (classes and interfaces), name operations on their types, specify arguments of their operations, and designate different scopes to avoid symbol name collision. Unlike C++, each argument is explicitly annotated as `in`, `out`, or `inout` to indicate whether data is being passed as an input, produced as an output, or used as input and output for the operation. SIDL has only declarative statements and no mechanism for defining states or algorithms.

Babel's main purpose is to enable scientific library designers to make their code language independent and thus reach a broader audience [18]. *Babelizing* an existing library typically involves writing a SIDL interface specification, running the Babel code generator to generate implementation bindings (called *Impls* for short) in the same language as the library, and hand coding the empty Impls to dispatch to the existing software. Though Babelizing code requires manual programming, customers find it easier than generating a single language wrapper by hand. Some even welcome the opportunity to craft a modern object-oriented interface over their legacy procedural code.

More details about Babel-generated Impls are needed for discussion in later sections. Recall that implementation details of software are intentionally inexpressible in SIDL. Therefore, when Babel first generates Impls, the bodies of the methods, member functions, subroutines, or procedures (depending on what programming language Babel's generating Impls for) are empty. The library developer needs to fill in the Impls with an actual implementation or code to dispatch their existing code. Since Impls are generated code that contain hand-edited fragments, these fragments are located in *splicer blocks*. Contents of splicer blocks are preserved across multiple runs of Babel as SIDL specifications evolve. Automatically generating contents for the splicer blocks is one of the challenges for this joint work.

3 Automated Code Generation, Transformations, and Analysis

The processing steps for automatically translating a library into components are summarized in Figure 1. Essentially, the steps are

1. SIDL and C++ implementation information extraction.
   This step involves using ROSE Translator T1 to process the *library example* to generate SIDL code and *Non-SIDL C++ Information* for ROSE Translator T2.

2. Stub and Impl generation.
   Babel is called using the SIDL file from step 1 to generate all stubs for clients in all supported languages as well as the corresponding C++ *Impl* files with empty *splicer blocks*.

3. Implementation bridge generation.
   The ROSE Translator T2 is called using *Non-SIDL C++ Information* and the C++ *Impl* files from step 2 to insert dispatching code into the initially empty *splicer blocks*.

To simplify the process each of these steps can be fully contained within a single program.

The input to the ROSE Translator T1 in Figure 1 includes two objects: a target library and a simple *library example* program. The *library example* program is constructed by hand and must include all the header files required to define the target library's implementation. Only the library's header files must be seen, although more sophisticated analysis is possible by processing the entire library as described in Section 3.3. The *library example* program can be as simple as a one line file that includes a single header file. For example, a file containing the line `#include <A++.h>` is a sufficient *library example* program for processing the A++ library.

Given the target library and an example program, the ROSE Translator T1 generates two outputs: SIDL interface files and Non-SIDL C++ information. The SIDL interface

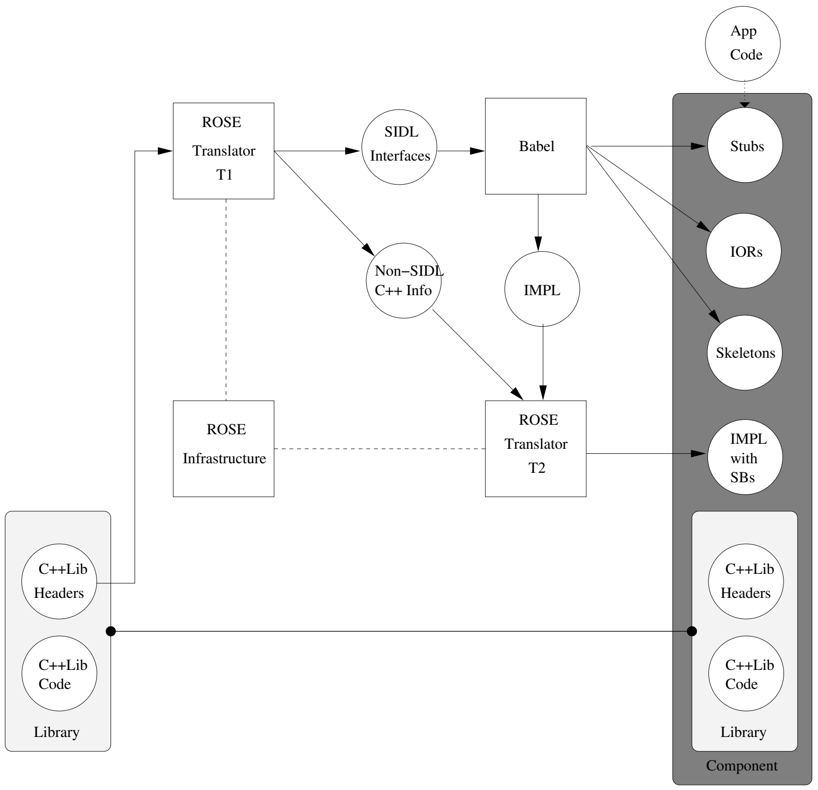


Figure 1. Component Generation Process with ROSE-Babel infrastructure. Circles represent (generated) files and arrows show the data-flow of these files. The dashed lines show that the translators T1 and T2 use the ROSE infrastructure. The library is not modified and becomes part of the generated component.

files are later used as inputs to Babel, which then generates components specifications and Impl files with empty splicer blocks. In greater detail, the ROSE Translator T1 performs the following substeps of step 1:

1.a. Constructing AST.
The AST at this point represents all library declarations. Only classes, structs, functions and member functions are of particular interest, but the AST also contains all comments, pragmas, variables, typedefs, etc.

1.b. Collecting information about classes and functions

Class definitions. Builds the list of library classes, each of which is translated into SIDL classes.

Member functions. This step builds a list of all member functions, each of which is put into its associated SIDL class previously constructed (preceding step).

Non-member functions. Builds a list of all non-member function, each of which is put into a SIDL class called "Global".

1.c. Generating *Non-SIDL C++ Information*
A file containing the list of `#include` directives is generated. The file is specified on the command-line as library specific data to be read by the ROSE Translator T2.

The above steps preserve the original structure of the library. The generated SIDL code does not re-organize the library other than presenting a list of global functions, classes, and member functions. To add more structures to the generated code, the following information can be used: directory and filename information, function name prefix information, pragmas in the library headers to specify mapping of functions to SIDL interface classes, and an alternative external annotation mechanism for specifying the mapping of functions to SIDL interface classes. These heuristics are part of our ongoing research.

The ROSE Translator T2 requires two inputs: the Impl files from Babel and the non-SIDL C++ information from ROSE Translator T1. Since the SIDL language doesn't permit the specification of `#include` directives, the non-SIDL information includes the list of `#include` directives present in the library example program processed by T1. These directives are required for declaring the library interface in the new Impl files generated by The ROSE Translator T2, which performs the following substeps of step 3:

3.a. Inserting the list of `#include` directives into the Impl file's appropriate splicer block.

3.b. Inserting code to map each input parameter of each Impl function to the appropriate parameter of the library's function call.

3.c. Inserting library function calls into the appropriate Impl function's splicer block.

Through library annotations or analysis, we can exploit SIDL specific features that are not present in C++ (e.g., specification of side-effects to function parameters via `in`, `out`, and `inout`), see Section 3.3. A third ROSE Translator could automate analysis of the library, using side-effect analysis to verify the correctness of parameter annotations or to use an in or out specification in lieu of the default inout. This narrower specification of parameters enables subsequent compiler optimization.

3.1 Generation of SIDL

Because the set of C++ features is much larger than those present in SIDL, mapping from C++ to SIDL requires some complex translation. Much information could be lost in this process, although it could conceivably be saved in the *Non-SIDL C++ Information* and used within the marshaling of function parameters between the Impl functions (generated by Babel) and the target library's function calls. The following issues have been considered in the existing translation of C++ code to SIDL:

C++ overloaded functions. Additional information is required within SIDL to support overloaded operators.

C++ overloaded operators. All overloaded operators are given unique names within the generation of SIDL. These names are mapped back to the respective overloaded operators within the transformation of the Impl files.

C++ function pointers. These are handled using a SIDL opaque. Some function pointers will be replaced by a SIDL interface.

Multiple inheritance. SIDL supports only single inheritance for classes and multiple inheritance of interfaces (similar to Java and Objective C). Through a level of indirection (supported in the interface parameter marshaling), multiple inheritance models in C++ can be reduced to single inheritance models appropriate for representation in SIDL.

C++ templates. There is no C++-like templating mechanism available as part of the SIDL interface. However, each template instantiated internally in the target library is represented as a *template-instantiation class* within ROSE, which can be translated to any non-template class within SIDL. This permits the use of templates within C++ libraries so long as they are instantiated over a closed set of parameterized types. This detail requires that the *library example* program triggers instantiate (uses) all templates.

SIDL support for arrays. SIDL supports arrays of specific types, but functions passing pointers to data and an integer describing its length can skip the use of the SIDL array abstractions. This avoids a translation ambiguity.

Variable arguments. C++ methods with variable numbers of arguments, using the ellipsis . . . in their declaration must be converted to a method with a SIDL array containing a generic argument base class.

SIDL's opaque type is necessary for low level routines with application programming interfaces (APIs) that require address pointers. For example, an opaque would have to be used for the ANSI C routine `signal` which requires a function pointer as an argument because our tool cannot change the underlying implementation to use a functor approach. A routine such as ANSI C's `malloc`, would need to use opaque as a return value. Certain device drivers might also require particular addresses as arguments.

3.2 Transformation of Impl files

Babel generates both stubs for other languages to call and Impl files to invoke the implementation of the library functions. Instead of generating new Impl files, which is handled by Babel, we transform the Impl files generated by Babel by inserting calls to the associated library functions and marshaling all parameters.

3.3 Library Analysis

An optional step is to process the target library implementation and analyze each function in the library to determine the side-effects upon their parameters. The side-effect analysis has been implemented as a result of collaborations with Cornell and will permit a verification of (`in`, `out`, and `inout`) annotations and or the generation of such annotations. The correct classification of interface function parameters is mostly a performance issue. The current conservative default classification is to classify all function parameters as `inout`. An additional processing step using ROSE could automate much of the classification of function parameters. In some cases, lack of sufficient program analysis, in particular pointer alias analysis, may require the process be semi-automatic rather than fully automatic. For example, side-effect analysis could signal that a potential alias between a locally-modified variable v and a parameter p prevents declaring p as an in parameter rather than inout. This ambiguity could be resolved by an annotation, specified through ROSE's annotation language, which declares that v does not alias p.

3.4 Future Extension: CCA Componentization

The CCA component framework has two requirements: *components* and *ports*. A component is an independent unit of composition that must implement the `gov.cca.Component` interface. Ports are capabilities, or services, of a component that must be specified in SIDL as extensions of `gov.cca.Port`.

Our translator will need to generate a set of classes implementing `gov.cca.Component`. This interface has one method, `setServices`, that must notify the framework about which ports the component can provide (known as *provides ports*) and which ports the component requires (known as *uses ports*). Identifying the mapping from the original set of C++ classes to SIDL classes implementing `gov.cca.Component` is a major challenge. In some cases, it might be best to treat each concrete C++ class as a component, and in other cases, the whole multi-class library should be considered a single component.

Our translator will also need to generate a set of ports based on the C++ classes in the original API. The first step will be to create a port for each C++ class involved. Determining better methods for choosing which C++ classes should be included in each type of port will need to be explored. *Provides ports* are basically services provided to the library's user; hence, they can be gleaned from the interfaces of a class. However, a *uses port* indicates services provided by the component's client that are needed by the component. Designating something as a *uses port* of a component means that the component needs exactly one instance of the port corresponding to the initial C++ class, which is very challenging to determine from the C++ header files. The underlying code may be able to handle zero through many instances.

4 Related Work

Our aim is to automate the generation of components from legacy scientific applications. This process includes two phases: extracting component interfaces and producing implementations that bind the interface specifications with the original software.

A number of research efforts have aimed at extracting components from existing software. Specifically, many clustering techniques [19, 20, 15] have been developed to analyze the function calls within a library system and to identify reusable components within the library. We currently focus on making all individual classes in a library reusable. Our work can be combined with the clustering techniques to provide better component interfaces for libraries. Beck and Eichmann [10] have also explored the extraction of interfaces from source code. They have focused on reducing the interfaces (and code) to only those methods that are actually needed by a user. Their solution is language-specific; whereas we focus on extracting language-neutral specifications and automating the library bindings.

To automate the second phase of generating components, Babel provides the translation from the interface specification to implementation stubs. Our work then generates dispatching code that fills in these stubs. Prior efforts have developed several systems that support automatic bridging of pairs of different languages. For example, SWIG [6, 9], a wrapper and interface generator, supports automatic bindings between C/C++ and common scripting languages such as Tcl, Python and Perl. In contrast, we leverage Babel's intermediate representation, so component developers do not have to be concerned with providing a point-to-point mapping for their users. Furthermore, Babel's RPC-like mechanism will enable future remote access to the libraries wrapped by our infrastructure. Chasm [22, 23], another point-to-point adapter generator, employs static compiler analysis to automatically connect C++ applications to FORTRAN 90 libraries. Our work, on the other hand, automates the connection of C++ libraries with applications written in a variety of scientific computing programming languages. By leveraging SIDL, which has been adopted as the specification language for scientific components by the Common Component Architecture Forum [1, 8], we enable the automatic generation of CCA-compliant components from existing libraries.

Similar to our work, Rational Rose [4], a commercial general-purpose graphical modeling tool, supports round-trip engineering from user applications to both CORBA and COM specifications [11, 17] using the Unified Modeling Language (UML) [7] as the intermediate representation. Our work, on the other hand, does not require the translation from yet another intermediate language. By virtue of using Babel/SIDL, we also have an intermediate layer that is tailored for scientific computing.

5 Conclusions and Future Work

This paper presents work on the automated generation of components. The work is incomplete presently but shows both the automation of the SIDL description and the connection of the generated code (from SIDL) back to the library. Both pieces are essential to automate the connection to an arbitrary library. To enable automatic generation of CCA components, additional analysis and transformations of the resulting SIDL objects is necessary to properly define and designate ports and implement the CCA required `setServices()` method.

All known "automatic" language wrapping tools require some degree of hints, pragmas, structured comments, or the like just to enable a one way connection from the calling language to the existing code. To "automatically" Babelize an existing C++ code is even more challenging, but offers more capabilities if successful. Taking the entire body of Babelized code and packaging it up as a component actually involves analysis of how the code is used and creation of new functions in the interface. The ultimate goal is to break any large existing code up into useful constituent CCA components.

Although the current ROSE infrastructure is limited to C and C++, the essential motivations are language independent. Analogous reverse mappings of FORTRAN to SIDL would require FORTRAN-specific analysis and techniques.

Specific details to C and C++ are addressed separately. It is conceivable that all C++ language feature could be mapped to SIDL without extension, but with some library specific translation support. Still, such tools could be made easy to build in the future, perhaps even automatically generated.

6 Acknowledgments

This work was performed under the auspices of the U. S. Department of Energy by the University of California, Lawrence Livermore National Laboratory under Contract No. W-7405-Eng-48. UCRL-CONF-208403

References

[1] Common Component Architecture forum. http://www.cca-forum.org.

[2] Edison Design Group. http://www.edg.com.

[3] Object Management Group's CORBA Component Model. Available online from the OMG http://www.omg.org/.

[4] Rational Rose UNIX. http://www-3.ibm.com/software/awdtools/developer/rose/unix/.

[5] Sun Microsystems' Enterprise JavaBeans Downloads and Specifications. Available online from Sun http://java.sun.com/products/ejb/docs.html.

[6] SWIG. http://www.swig.org.

[7] Unified Modeling Language. http://www.uml.org.

[8] R. Armstrong, D. Gannon, A. Geist, K. Keahey, S. Kohn, L. McInnes, S. Parker, and B. Smolinski. Toward a common component architecture for high-performance scientific computing. In *Proceedings of the 8th High Performance Distributed Computing*, 1999.

[9] D. M. Beazley. SWIG: An easy to use tool for integrating scripting languages with C and C++. In *Proceedings of the 4th Annual Tcl/Tk Workshop*, Monterey, CA, July 1996.

[10] J. Beck and D. Eichmann. Program and interface slicing for reverse engineering. In *Proceedings of the 15th International Conference on Software Engineering (ICSE '93)*, pages 509–518, May 1993. Baltimore, Maryland.

[11] N. Bereny. Rose 101: Component Modeling with Rose 98. *Rose Architect*, January 1999.

[12] D. E. Bernholdt, W. R. Elwasif, J. A. Kohl, and T. G. W. Epperly. A component architecture for high-performance computing. In *Proceedings of the Workshop on Performance Optimization via High-Level Languages and Libraries*, New York, NY, June 2002.

[13] F. Bodin, P. Beckman, D. Gannon, J. Gotwals, S. Narayana, S. Srinivas, and B. Winnicka. Sage++: An object-oriented toolkit and class library for building fortran and C++ restructuring tools. In *Proceedings. OONSKI '94*, Oregon, 1994.

[14] D. Box. *Essential COM*. Addison-Wesley Professional, 1997.

[15] Y. Chiricota, F. Jourdan, and G. Melancon. Software components capture using graph clustering. In *Proceedings of the 11th International Workshop on Program Comprehension (IWPC 2003)*, pages 217–225, May 2003.

[16] T. Dahlgren, T. Epperly, and G. Kumfert. *Babel User's Guide*. Lawrence Livermore National Laboratory, Livermore, CA, 0.8.8 edition, 2003.

[17] J. Hammond. CORBA and Rational Rose – An Insider's View. *Rose Architect*, April 1999.

[18] S. Kohn, G. Kumfert, J. Painter, and C. Ribbens. Divorcing language dependencies from a scientific software library. In *10th SIAM Conference on Parallel Processing*, Portsmouth, VA, March 2001.

[19] R. Koschke and T. Eisenbarth. A framework for experimental evaluation of clustering techniques. In *Proceedings of the 8th International Workshop on Program Comprehension (IWPC 2000)*, pages 201–210, June 2000.

[20] B. S. Mitchell and S. Mancoridis. Modeling the search landscape of metaheuristic software clustering algorithms. In *Proceedings of the 7th Annual Genetic and Evolutionary Computing Conference (GECCO '03)*, pages 2499–2510, July 2003.

[21] D. Quinlan, M. Schordan, B. Miller, and M. Kowarschik. Parallel object-oriented framework optimization. *Concurrency and Computation: Practice and Experience*, 2003.

[22] C. E. Rasmussen, K. A. Lindlan, B. Mohr, and J. Striegnitz. CHASM: Static analysis and automatic code generation for improved Fortran 90 and C++ interoperability. In *Proceedings of the Los Alamos Computer Science Symposium 2001 (LACSI'01)*, October 2001. Santa Fe, New Mexico.

[23] C. E. Rasmussen, M. J. Sottile, S. S. Shende, and A. D. Malony. Bridging the language gap in scientific computing: the Chasm approach. (in submission).

[24] M. Schordan and D. Quinlan. A source-to-source architecture for user-defined optimizations. In *JMLC'03: Joint Modular Languages Conference*, volume 2789 of *Lecture Notes in Computer Science*, pages 214–223. Springer Verlag, Aug. 2003.

Improving Time to Solution with Automated Performance Analysis

Shirley Moore, Felix Wolf, and Jack Dongarra
Innovative Computing Laboratory
University of Tennessee
{shirley,fwolf,dongarra}@cs.utk.edu *

Bernd Mohr
Zentralinstitut für Angewandte Mathematik
Forschungszentrum Jülich
b.mohr@fz-juelich.de +

Abstract

The complex architectures of HEC systems present difficult challenges for performance optimization of scientific applications. Tools are needed that collect and present relevant information on application performance in a scalable manner so as to enable developers to easily identify and determine the causes of performance bottlenecks. This paper describes KOJAK, a suite of performance analysis tools that collect and analyze runtime data from high performance applications. Performance data are collected automatically using a combination of source code annotaions or binary instrumentation and hardware counters. The analysis tools use pattern recognition to convert the raw performance data into information about performance bottlenecks relevant to developers. Such automated approaches to performance instrumentation and analysis promise to increase programmer productivity and reduce time to solution by reducing both development and execution time.

1. Introduction

High performance computing is playing an increasingly critical role in advanced scientific research as simulation and computation are becoming widely used to augment and/or replace physical experiments. However, the gap between peak and achieved performance for scientific applications running on high-end computing (HEC) systems has grown considerably in recent years. The complex architectures and deep memory hierarchies of HEC systems present difficult challenges for performance optimization of scientific applications. Tools are needed that collect and present relevant information on application performance in a scalable manner so as to enable developers to easily identify and determine the causes of performance bottlenecks. According to the Report of the High-End Computing Revitalization Task Force (HECRTF) [1], the single most important metric for high-end system performance is *time to solution* for the scientific applications of interest. Time to solution includes not only execution time, but also development time. Portable, easy-to-use, effective performance tools aim to reduce both development and execution time.

In order to collect performance data, the application must be instrumented in some manner. To be most useful for performance tuning, the data should be collected at routine or even basic block or loop granularity. For developers of large-scale applications to implement this level of instrumentation manually is too time-consuming and thus not feasible. Automated instrumentation techniques are needed that can collect the relevant data with a minimum of effort.

Developers of scientific applications for HEC systems are not necessarily experts in high performance computing architectures and performance analysis. For this reason, performance data at the level of un-interpreted hardware counter data or communication statistics or traces may not be useful to these developers. Higher level abstractions that identify various types of performance problems, such as inefficient use of the memory hierarchy or excessive synchronization delay for example, and that map these problems to the relevant application source code, will be much more useful and allow performance tuning to be done with much less time and effort.

The amount of performance data collected for

*Work by these authors has been supported by the DOE SciDAC program under grant number DE-FC02-01ER25490, by the DOE MICS program under grant number DE-FG02-01ER25510, and by the DoD HPCMP PET program under Contract No. N62306-01-7110.

+Work by this author has been supported by the Helmholtz Research Programme "Scientific Computing" in the Research Field "Key Technologies" and by the European IST working group APART under Contract No. IST-2000-28077.

applications running on HEC systems can be overwhelming. While analysis of event tracing has proved to be a superior technique to identify performance problems at a high level of abstraction, it usually suffers from scalability problems associated with trace-file size. Even collecting detailed profiling data for large numbers of processes can be unwieldy. Current display tools are limited in their representation of large-scale performance data.

This paper describes our efforts at addressing the above problems as part of the KOJAK project [2,3].

2. Automated Instrumentation

Figure 1 gives an overview of KOJAK's architecture and its components. The KOJAK analysis process is composed of two parts: a semi-automatic multi-level instrumentation of the user application followed by an automatic analysis of the generated performance data.

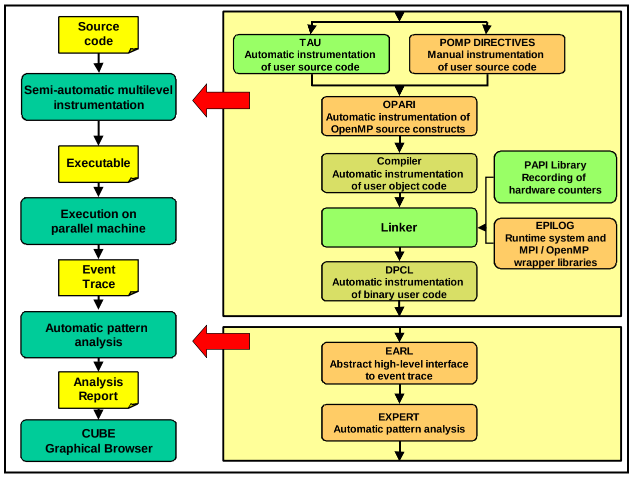

Source code
Semi-automatic multilevel instrumentation
Executable
Execution on parallel machine
Event Trace
Automatic pattern analysis
Analysis Report
CUBE Graphical Browser
TAU Automatic instrumentation of user source code
POMP DIRECTIVES Manual instrumentation of user source code
OPARI Automatic instrumentation of OpenMP source constructs
Compiler Automatic instrumentation of user object code
PAPI Library Recording of hardware counters
Linker
EPILOG Runtime system and MPI / OpenMP wrapper libraries
DPCL Automatic instrumentation of binary user code
EARL Abstract high-level interface to event trace
EXPERT Automatic pattern analysis

Figure 1. KOJAK Architecture

The event traces generated by EPILOG capture MPI point-to-point and collective communication as well as OpenMP parallelism change, parallel constructs, and synchronization. In addition, data from hardware counters accessed using the PAPI library [4,5] can be recorded in the event traces. To make measurements with the EPILOG system, the user's application must be instrumented at specific important points, or *events*, to activate EPILOG library calls. Events of interest include sending and receiving messages, user function entries and exits, entering and exiting OpenMP regions, and synchronization operations such as acquiring and releasing locks. Automated instrumentation is supported by compiler instrumentation on the following platforms:

- Linux clusters using the PGI compilers
- Hitachi SR-8000
- Sun Solaris (Sun Fortran90 compilers only)
- NEC SX

The instrumentation of user function entries and exits on the above systems is based on undocumented and unsupported compiler options. Discussions are underway with additional vendors to provide similar instrumentation hooks. Ideally these compiler instrumentation hooks will become fully supported in the future. On the above systems, all necessary instrumentation of user functions, MPI functions, and OpenMP constructs is handled by the "kinst" command. In the commands to build the application (e.g., in a makefile), the user need only precede all compile and link commands with "kinst". For example, instead of the command

```
% mpif90 myprog1.f90 myprog2.f90 –o myprog
```

the command

```
% kinst mpif90 myprog1.f90 myprog2.f90 –o myprog
```

would be executed.

For platforms on which compiler instrumentation using kinst is not supported, the users may manually instrument the desired functions and regions of their application by inserting POMP instrumentation directives and then using the "kinst-pomp" command in the same way as described above for "kinst". POMP instrumentation directives are supported for Fortran and C/C++ and are replaced by the necessary instrumentation calls by our source-to-source transformation tool OPARI [6]. In the case of OpenMP programs, OPARI also automatically instruments all OpenMP constructs and OpenMP run-time library calls by inserting calls to the POMP monitoring API [7]. An advantage of using POMP instrumentation directives is that the instrumentation is ignored during normal compilation. An INST BEGIN/INST END pair can be used to mark any user-defined sequence of statements, again with a single argument giving a name for the code region. At least the main program function must be instrumented in this way, and in addition, an INST INIT directive must be inserted as the first executable statement of the main program.

While fairly straightforward, such manual instrumentation of a large program is time-consuming and has an adverse effect on time to solution. In addition, the manual instrumentation must be redone with every new version of the program Fortunately, the TAU performance analysis system [8] provides an automated source code instrumentation mechanism that can be used with the EPILOG library. TAU is a cross-platform tool that supports a wide variety of HEC platforms. To use TAU's automated source code instrumentation, the user should first configure and build

TAU with the desired options. To use EPILOG, TAU should be configured with the –TRACE and –epilog options. Then only two changes need to be made to the application makefile. First, a makefile stub with the necessary TAU definitions, which was created when the appropriate library was built, should be included. Then the user need only precede all compile and link commands with $(TAU_COMPILER). All MPI functions, user functions, and OpenMP constructs will then be instrumented with EPILOG library calls. TAU also uses KOJAK's OPARI system [6] to automatically instrument OpenMP constructs. Although TAU currently supports automated source code instrumentation only down to the routine level for non-OpenMP codes, there are plans to extend the automated instrumentation capability to the basic block and loop level.

An alternative to source code instrumentation is to use automatic binary instrumentation. KOJAK supports binary instrumentation on IBM systems where the optional DPCL (Dynamic Probe Class Library) package [9] has been installed. The user need only precede compile and link commands with "kinst-dpcl" and launch the resulting program using the "elg-dpcl" command. TAU supports binary instrumentation using the Dyninst[10,11] library. The *tau_run* tool dynamically loads the specified TAU instrumentation library and instruments the application at runtime. All user and MPI functions are instrumented.

If EPILOG has been built with hardware counter support enabled, then hardware counter data can be recorded as part of the event records. To request the measurement of certain counters, the user must set the environment variable ELG_METRICS to a colon-separated list of counter names. EPILOG uses the PAPI library [4,5] to access the hardware counters. All of the PAPI standard metrics are supported for data collection although not all are currently supported for automated analysis.

Any of the instrumentation methods described above will cause an EPILOG trace file to be produced when the application is run. The per-process trace files generated during the execution will be automatically merged into a single trace file when execution ends. The resulting trace file can be analyzed using KOJAK's automated performance analysis as explained below.

3. Automated Performance Analysis

Large-scale applications running on HEC systems can produce extremely large trace files. Visualization tools such as Vampir and Intel Trace Analyzer [12], Jumpshot [13], and Paraver [14] can provide a graphical view of the state changes and message passing activity represented in the trace file, as well as provide statistical summaries of communication behavior. However, it is difficult and time-consuming for even expert users to identify performance problems from such a view or from large amounts of statistical data. Spending large amounts of time analyzing performance data manually has a negative effect on time to solution. KOJAK's EXPERT tool is an automatic trace analyzer that attempts to identify specific performance problems. Internally, EXPERT represents performance problems in the form of execution patterns that model inefficient behavior. These patterns are used during the analysis process to recognize and quantify inefficient behavior in the application.

The performance problems addressed by EXPERT include inefficient use of the parallel programming model and low CPU and memory performance. Internally patterns are specificed as C++ classes that provide callback methods to be called upon occurrence of specific event types in the event stream. The pattern classes are organized in a specialization hierarchy, as shown in Figure 2. There are two types of patterns: 1) simple profiling patterns based on how much time or some other metric (e.g., cache misses) is spent in certain MPI calls or code regions, and 2) patterns describing complex inefficiency situations usually described by multiple events – e.g., late sender in point-to-point communication or synchronization delay before all-to-all operations. Recent work has taken advantage of the specialization relationships to obtain a significant speed improvement for EXPERT and to allow more compact pattern specifications [15]. Each pattern calculates a (call path, location) matrix containing the time spent on a specific behavior in a particular (call path, location) pair, where a location is a process or thread. Thus, EXPERT maps the (performance problem, call path, location) space onto the time spent on a particular performance problem while the program was executing in a particular call path at a particular location. After the analysis has been finished, the mapping is written to a file and can be viewed using the CUBE display tool.

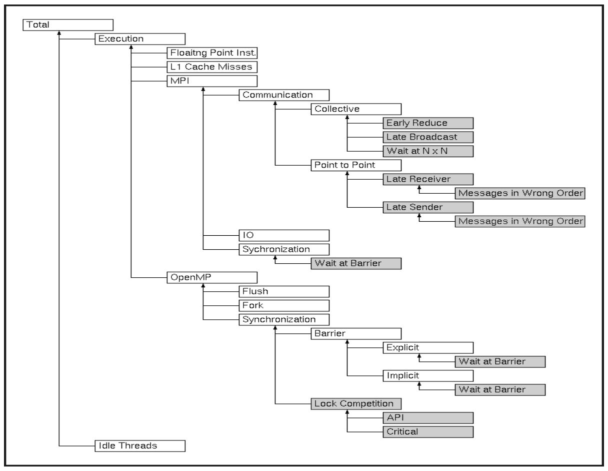


Figure 2. EXPERT pattern specialization hierarchy

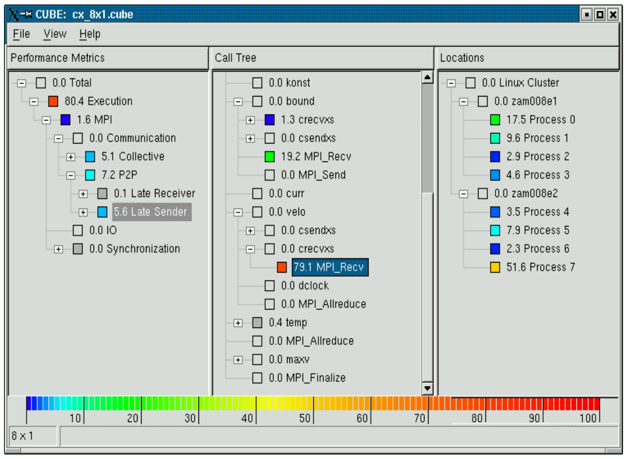


Figure 3. CUBE display showing metric, call tree, and location dimensions

The CUBE display for a crystal growth simulation [16] run on eight processors of a Linux cluster is shown in Figure 3. The display consists of three coupled tree browsers, representing the metric, the program, and the location dimensions from left to right. The user can switch between a call tree and a flat profile view of the program dimension, with the default being the call-tree view. The nodes in the metric tree represent performance metrics, the nodes in the call tree represent call paths, and the nodes in the system tree represent machines, nodes, processes, and threads. A user can perform two types of actions: selecting a node or expanding/collapsing a node. At any given time, there are two nodes selected, one in the metric tree and one in the call tree. Each node is labeled with a severity value. A value shown in the metric tree represents the sum of a particular metric for the entire program, that is, across all call paths and all locations. A value shown in the call tree represents the sum of the selected metric across all locations for a particular call path. A value shown in the location tree represents the selected metric for the selected call path and a particular location. All numbers may be displayed either as absolute values or as percentages. To help identify metric/resource combinations with a high severity, values are ranked using colors. The color legend shows a numeric scale mapping values to colors. Note that all hierarchies in CUBE are inclusion hierarchies, meaning that a child node represents a subset of a parent node. The severity value in CUBE follows the principle of *single representation* – that is, within a tree each fraction of the severity is displayed only once. The purpose of this strategy is to have a particular problem appear only once in the tree and thus help identify it more quickly.

Performance optimization of parallel applications usually involves multiple experiments to compare the effects of different code versions, different execution configurations, or different input data. In addition, hardware characteristics may limit the availability of certain performance data, such as hardware counter data, in a single run, requiring multiple experiments to obtain a full set of data. A user may also wish to combine the results obtained using different monitoring tools that cannot be applied simultaneously. Finally, results of analytical modeling or simulation may need to be compared with experimental data. The traditional method of comparing different experiments is to put multiple single-experiment views side by side or to plot overlay diagrams. Previous research on multi-experiment analysis described in [17] uses an operator to calculate a list of resources showing significant discrepancies between different experiments. However, this difference operator maps from its input space containing entire experiments into a smaller representation consisting of a list of resources. A repeated application is not possible, and further processing would require a logic or display different from the one suitable for the original input data. With our approach the output of multi-experiment analysis can be represented just like its input, allowing us to use the same set of tools to process and display it. The CUBE performance algebra can be used to compare, integrate, and summarize performance data of message-passing and/or multithreaded applications from multiple experiments including results obtained from simulations and analytical modeling. The algebra consists of a data model to represent the data in a platform-independent fashion plus arithmetic operations to subtract, merge, and average the data from multiple experiments., All operations are closed in that their results are mapped into the same space, yielding an entire "derived" experiment including data and metadata. Figure 4 shows the differences between two versions of a nano-particle simulation, with raised reliefs indicating performance improvements and sunken reliefs indicating performance degradations. The differences are broken down along the various dimensions. The CUBE performance algebra is described in further detail in [18].

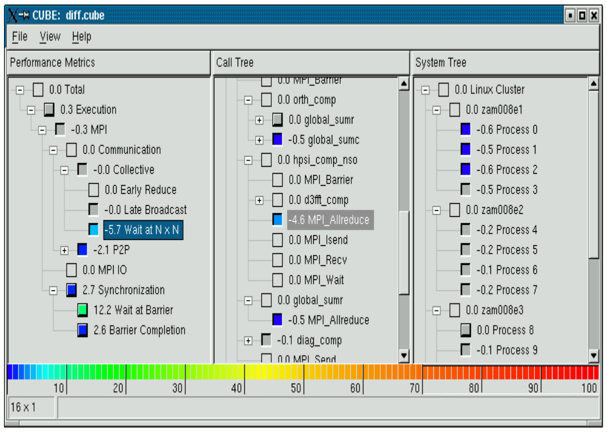


Figure 4. Intuitive display of differences between two code versions

4. Scalability and Portability Issues

While analysis of event tracing has proved to be a superior technique to identify performance problems at a high level of abstraction, it usually suffers from scalability problems associated with trace-file size. If the automated instrumentation techniques described in section 2 are applied non-discriminately, they can instrument all user and MPI functions and OpenMP constructs with trace library calls, resulting in a large amount of trace data being collected. Although KOJAK's approach of reducing the trace data to a higher-level, more compact representation using EXPERT results in a much smaller data file, the initial very large raw trace file can be problematic. Fortunately TAU provides a filtering mechanism available to reduce the instrumentation. An initial profiling run can be conducted to identify routines that are called a very large number of times and for which trace data do not contribute much useful information. These routines can then be excluded from the automated instrumentation by specifying them in an exclude list. The *tau_reduce* tool can be used to generate the exclusion list automatically, and work is underway to add this capability to KOJAK's module that handles automatic user function instrumentation via compiler switches.

Even with careful filtering, large-scale applications can still produce very large trace files. In view of present and future architectures consisting of thousands of processors and in view of applications running on all or at least a major fraction of the available CPUs, KOJAK's current approach will become increasingly constrained by the potentially enormous size of the resulting event traces. The current approach of collecting a large trace file for an entire parallel program execution in a centralized location and then processing and reducing this single trace file, such as the approach used by the EXPERT trace analysis tool in KOJAK, will not scale to thousands of processors. Although recent improvements have made an order of magnitude improvement in EXPERT's efficiency [15], our future research in this area will focus on applying parallel and distributed processing approaches to the processing, reduction, and filtering of large-scale trace data.

KOJAK's current CUBE display will be unwieldy for representing HEC systems with thousands of processors. We plan to develop a highly optimized version of the CUBE display that replaces the current tree representation of processes and threads with a much more scalable multi-dimensional topology display reflecting the virtual topology of the application and/or the physical topology of the machine. As an integral part of parallel programming deals with choosing the right virtual topology, that is, the mapping of processes and threads onto the problem domain, a topology display will not only be much more scalable but can also provide more intuitive guidance in analyzing the influence of physical or logical communication structures.

KOJAK is portable across a wide range of UNIX platforms including Linux on IA-32, IA-64, and Opteron; IBM POWER 3 and 4; SGI MIPS and IA-64; SUN Sparc; HP Alpha; Cray T3E and X1; IBM BG/L; NEC SX; and
Hitachi SR8000. Since the analysis performed by KOJAK is based on standardized characteristics of the programming models MPI and OpenMP, the platform-dependent part of KOJAK's implementation is very small. Porting KOJAK to a new platform requires essentially the provision of time-measurement routines, information on local file systems, and platform and node names. Portable access to hardware counters is achieved by using the PAPI library [4,5]. Portable source code instrumentation is achieved using PDToolkit and Opari [6]. Portable binary instrumentation is achieved using the Dyninst [11] and DPCL [9].

5. Conclusions and Future Work

Automated approaches to performance instrumentation and analysis promise to increase programmer productivity and reduce time to solution by reducing both development time and execution time. Performance tuning is often a neglected part of application development because the amount of effort invested does not yield an adequate return in reduction of execution time. By automatically and accurately pinpointing the most severe performance problems, the amount of effort can be reduced while achieving greater performance gains.

Although some significant results have been obtained already, the pattern analysis used by EXPERT could be considerably improved. We have only begun to scratch the surface on the specification of patterns based on hardware counter data and on correlating these data with other events and with program data structures. The pattern search could also be made more accurate by applying it at the loop level. Scientific applications frequently contain computationally intensive nested loop structures, the tuning of which is critical to achieving good performance. We expect the combination of automated instrumentation at the loop level and the specification of patterns for analyzing nested loop performance to provide a powerful mechanism for achieving substantial performance gains with a minimum of effort.

To be most useful to a developer in tuning an application, information about cache and memory behavior should be presented in a way that relates it to program data structures at the source code level. KOJAK's EXPERT analyzer and CUBE display tool support post-mortem analysis of trace and/or profile data with a display that allows the user to interactively explore a similar three-dimensional performance space with the metric, call tree, and location dimensions displayed by coupled tree browsers. In order to enable performance analysis to focus specifically on memory hierarchy performance as it relates to data structures used by an application, we plan to extend the search space to include an explicit data structure dimension. This dimension will include various levels of data structures that may be distributed across multiple memories in a parallel system. Explicit representation of this dimension will better specification of patterns that represent inefficient memory system performance, as well as hyper-linking detected memory performance problems to entities in the other dimensions of the performance search space, such as the specific call path that is generating the particular memory performance problem.

Future HEC systems may require the use of new parallel programming paradigms. We have prototyped an extension of the KOJAK toolset that is able to instrument, record, and analyze MPI-2 one-sided communication and synchronization features. This work can be extended easily to handle vendor-specific one-sided communication such as SHMEM or LAPI. Work is also underway to analyze Co-Array Fortran [19] applications using KOJAK.

In Section 4, we have already mentioned planned future work on improving the scalability of trace-based automated performance analysis.

Our claim that automated performance analysis can improve time to solution of scientific applications on HEC systems needs to be verified with experimental evidence. The amount of effort actually involved in using the tools needs to be measured and the performance gains obtained quantified under controlled conditions. We plan to investigate programmer productivity metrics and apply them to measure the effectiveness of the KOJAK approach.

References

[1] Federal Plan for High-End Computing, Report of the High-End Computing Revitalization Task Force (HECRTF), Executive Office of the President, Office of Science and Technology Policy, May 2004, http://www.itrd.gov/pubs/2004_hecrtf/20040702_hecrtf.pdf.

[2] KOJAK web site, http://icl.cs.utk.edu/kojak/

[3] Wolf, F. and B. Mohr, Automatic performance analysis of hybrid MPI/OpenMP applications. *Journal of Systems Architecture*, November 2003, pp. 421-439.

[4] PAPI web site, http://icl.cs.utk.edu/papi/

[5] Browne, S., et al., A Portable Programming Interface for Performance Evaluation on Modern Processors. *International Journal of High-Performance Computing Applications* 14(3), 2000, pp. 189-204.

[6] B. Mohr, A. Malony, S. Shende, F. Wolf, Towards a Performance Tool Interface for OpenMP: An Approach Based on Directive Rewriting, *Proceedings of the Third European Workshop on OpenMP - EWOMP'01*, Barcelona, Spain, September 2001.

[7] B. Mohr, A. Malony, H.-Ch. Hoppe, F. Schlimbach, G. Haab, S. Shah, A Performance Monitoring Interface for OpenMP, *Proceedings of the fourth European*

Workshop on OpenMP - EWOMP'02, Rome, Italy, September 2002.

[8] TAU web site, http://www.cs.uoregon.edu/research/paracomp/tau/

[9] DPCL web site, http://www-124.com/developerworks/opensource/dpcl/

[10] Buck, B.R. and J.K. Hollingsworth, An API for Runtime Code Patching. *International Journal of High Performance Computing Applications* 14(4), 2000, pp. 317-329.

[11] Dyninst web site, http://www.dyninst.org/

[12] Intel Cluster Tools web site, http://www.intel.com/software/products/cluster/index.htm

[13] Jumpshot web siute, http://wwwunix.mcs.anl.gov/perfvis/software/viewers/index.htm

[14] Paraver web site, http://www.cepba.upc.es/paraver/

[15] Wolf, F., et al. Efficient Pattern Search in Large Traces through Successive Refinement, in *European Conference on Parallel Computing (Euro-Par)*. Pisa, Italy. August-September, 2004.

[16] Mihelcic, M., H. Wenzl, and H. Wingerath, Flow in Czochralski Crystal Growth Melts. Forschungszentrum Jülich Technical Report Jül-2697, 1992.

[17] Karavanic, K. L. and B. P. Miller, Experiment Management Support for Performance Tuning, in *SC'97*, San Jose, California, November 1997.

[18] Song, F., et al. An Algebra for Cross-Experiment Performance Analysis, in *International Conference on Parallel Processing (ICPP)*. Montreal, Canada. August, 2004.

[19] Numrich, R. W. and J.K. Reid, Co-Array Fortran for Parallel Programming, *ACM Fortran Forum* 17(2), pp 1-31, 1998.

The FG Programming Environment: Reducing Source Code Size for Parallel Programs Running on Clusters

Elena Riccio Davidson [*]
Thomas H. Cormen[†]

Dartmouth College Department of Computer Science
{laney, thc}@cs.dartmouth.edu

Abstract

FG is a programming environment designed to reduce the source code size and complexity of out-of-core programs running on clusters. Our goals for FG are threefold: (1) make these programs smaller, (2) make them faster, and (3) reduce time-to-solution. In this paper, we focus on the first metric: the efficacy of FG for reducing source code size and complexity. We designed FG to fit programs, including high-end computing (HEC) applications, for which hiding latency is paramount to designing an efficient implementation. Specifically, we target out-of-core programs that fit into a pipeline framework. We use as benchmarks three out-of-core implementations: bit-matrix-multiply/complement (BMMC) permutations, fast Fourier transform (FFT), and columnsort. FG reduces source code size by approximately 14–26% for these programs. Moreover, we believe that the code FG eliminates is the most difficult to write and debug.

1. Introduction

In this paper, we demonstrate that our programming environment, called ABCDEFG (FG for short) [9], reduces source code size for out-of-core implementations of bit-matrix-multiply/complement (BMMC) permutations, fast Fourier transform (FFT), and columnsort. Replacing each of these C and C* programs by a comparable program written with FG saves 468, 1322, and 2004 lines of source code, respectively. These reductions amount to percentage decreases of 14.6%, 17.4%, and 25.6% of the source-code lines, respectively.

The high-end computing (HEC) applications on which we focus are *out-of-core* programs running on clusters. In an out-of-core program, the amount of data exceeds the capacity of main memory, and therefore data must reside on disk. Performing disk I/O is a high-latency operation, and so in order to achieve a high-performance implementation, it is essential to hide latency in these programs. We take two separate but related approaches to hide latency. First, we must overlap work. Since we often use disk I/O and interprocessor communication, we can overlap these two types of operations with computation on the CPU. Second, we must use buffers to access data. A buffer is simply a block of memory; in our programs, we read into and write from buffers in order to amortize the cost of transferring data among levels of the memory hierarchy. The pairing of writing asynchronous code to overlap work and using buffers to access data effectively hides latency in HEC parallel programs. We call the code for creating asynchrony and managing buffers *glue*.

Each of the three programs that we focus on in this paper fits into a pipeline framework. For example, Figure 1 illustrates the pipeline structure we use for our implementation of out-of-core columnsort. The pipelines in each of the three programs contain a stage that reads from disk, a stage that writes to disk, a stage that performs interprocessor communication, and one or more stages that perform computation. To introduce asynchrony into our programs, we overlap work by running the stages of each pipeline concurrently. Buffers travel from stage to stage; every stage may be working on a distinct buffer simultaneously.

Every time a buffer travels the length of the pipeline, we say that one *round* of execution has completed. Since we are in an out-of-core setting, we expect that the number of rounds demanded by a program far exceeds the number of buffers that can fit in memory. Therefore, we must reuse buffers after they travel the length of the pipeline. We use a global pool for buffers; we store free buffers in the pool after we initially allocate them, and we return each buffer to the pool whenever it completes a round.

Hiding latency by way of creating asynchrony and man-

[*]Supported in part by DARPA Award W0133940 in collaboration with IBM.
[†]Supported in part by DARPA Award W0133940 in collaboration with IBM and in part by National Science Foundation Grant IIS-0326155 in collaboration with the University of Connecticut.

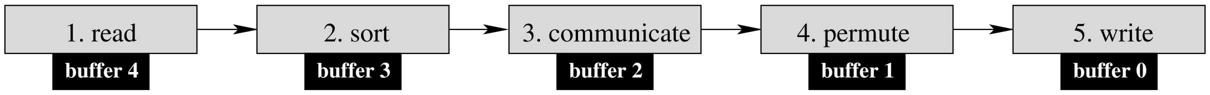


Figure 1: An implementation of out-of-core columnsort, represented as a pipeline. The first stage reads data from disk into a buffer. The second stage performs a local sort. The third stage performs interprocessor communication. The fourth stage performs a local permutation. The final stage writes data from the buffer to disk. The stages run concurrently so that, at any moment, each stage may be working on a distinct buffer.

aging buffers is a difficult task. Without FG, it means the programmer must produce a considerable amount of glue in addition to the code required to implement the algorithm. We define as *base code* any code that is not the glue; essentially, the base code is what the program would be without any attempt to overlap. The base code does not change significantly between FG and non-FG programs. With FG, however, the programming environment makes it much easier to incorporate the glue, thus reducing the source code size and complexity of out-of-core programs.

The remainder of this paper is organized as follows. Section 2 discusses two former methods for writing pipeline-structured programs: the ViC* system and C programming with threads. Section 3 is a brief description of the FG programming environment. Section 4 presents the three out-of-core applications we use as benchmarks for comparing FG and non-FG code. Section 5 analyzes the reductions in code size and complexity that we have achieved with FG. Finally, Section 6 offers some concluding remarks.

2. Previous approaches

In this section, we present two prior approaches that researchers at Dartmouth have taken to writing out-of-core programs that fit into a pipeline framework. ViC* [3] was a software system, developed during the period 1992–2001, that adapted C* programs for massive datasets. ViC* used static scheduling to introduce asynchrony, overlapping I/O with communication and computation. It was too difficult, however, to overlap communication with computation in the ViC* framework. Starting in 2001, we moved to programming pipeline-structured out-of-core applications in C with threads [1]. Threads use dynamic scheduling, and so we were able to overlap all three of communication, computation, and I/O. The programmer was responsible for coordinating all the actions associated with threads, however. With both approaches, the programmer was responsible for writing the code that managed buffers.

2.1. ViC*

ViC* was a compiler and run-time system, and it was the focus of out-of-core programming at Dartmouth starting in 1992. We implemented two significant out-of-core programs in ViC*, namely bit-matrix-multiply/complement (BMMC) permutations [5, 7] and fast Fourier transform (FFT) [8].

ViC* overlapped only I/O with other operations, and it used static scheduling to do so. That is, in order to enjoy even the partial overlapping of asynchronous I/O, the programmer had to produce the code that scheduled the I/O operations. Writing asynchronous I/O is far more complex than writing synchronous I/O. Figure 2 illustrates a simplified example of using asynchronous and synchronous I/O within an out-of-core permutation. The asynchronous version is more efficient than the synchronous version, because it uses a blocking wait to perform the in-core permutation while waiting for the disk I/O to complete, whereas the synchronous version first reads, then performs the in-core permutation, then writes. It is clear, however, that the synchronous version is much simpler to code.

Furthermore, in ViC*, the programmer was responsible for all aspects of buffer management, a task made more complicated by the presence of asynchronous I/O. The programmer had to allocate and deallocate buffers, store them in a global pool, keep track of which buffers were free, and recycle buffers that had traversed the entire pipeline. In addition to these primary buffers, the programmer was responsible for allocating and maintaining secondary buffers, used for in-core permutations. Although any permutation can be done in-place, it is often simpler to use distinct source and target buffers. With ViC*, the programmer had to allocate designated secondary buffers to use as target buffers, ensure that a particular one was free to use in a permutation, and, after the permutation, release the secondary buffer so that it could be used again.

2.2. Threaded programming

After the ViC* project, the focus of out-of-core computing at Dartmouth turned to writing C code using threads. We used standard POSIX threads [10] to overlap I/O as well as communication and computation, and so we were able to take advantage of the dynamic scheduling inherent in the pthreads package. With dynamic scheduling, any thread that is ready can run when the CPU becomes available. Moving from ViC* to threads meant that overlapping work for asynchrony no longer necessitated writing large amounts of code for statically scheduled asynchronous I/O.

```
$b = 0$
**start** read into buffer [$b$, 0]
**while** some read has not been started **do**
    wait for read into buffer [$b$, 0]
    **if** not first or second time through
        **then** wait for write to complete from target buffer [$b$, 1]
    **if** not working on the final buffer
        **then** start read into buffer [$1 - b$, 0]
    permute (in-core) from read buffer [$b$, 0] into target buffer [$b$, 1]
    **start** write of target buffer [$b$, 1]
    $b = 1 - b$
wait for writes to complete from target buffers [0, 1] and [1, 1]
```

(a)

```
**while** some read has not been started **do**
    read into buffer 0
    permute (in-core) from read buffer 0 into target buffer 1
    write target buffer 1
```

(b)

Figure 2: An out-of-core permutation using asynchronous and synchronous I/O operations. **(a)** Using asynchronous I/O. While waiting for a read or write to complete, we can begin the in-core permutation, but we must schedule it statically. **(b)** Using synchronous I/O. It is much simpler, but much less efficient, than its asynchronous counterpart.

It introduced a different kind of glue, however—all the code associated with spawning and coordinating the actions of the threads. Also, the programmer's burden in terms of buffer management was no different than with ViC*.

We implemented an out-of-core version of Leighton's columnsort algorithm [11] using this approach [1, 2]. In the threaded C code, we represented each stage of a pipeline as a thread. Since threads run concurrently, it was up to the programmer to ensure that they operated on buffers in order. The programmer spawned threads and coordinated among them using semaphores. Each stage had to wait for a signal from its predecessor before operating on a particular buffer; each stage also had to signal its successor after it finished working on the buffer. The structure of the pipeline, therefore, was tied to the operations within threads. The programmer had to write code to ensure that the threads signaled each other appropriately.

As with ViC*, the programmer was entirely responsible for buffer management in threaded code. The programmer had to allocate, deallocate, and store buffers, and recycle them from the global pool when necessary. Moreover, the programmer had to allocate and store additional buffers for stages whose work could not be done in place.

3. The FG environment

In this section, we present a simplified description of FG. The central job of FG is to provide the glue for HEC applications that fit a pipeline structure. To create asynchrony, FG represents each step of work as a pipeline stage and maps it to a thread. FG also manages the buffers to amortize the cost of transferring data among levels of the memory hierarchy. By shouldering both of these tasks, FG hides latency in such programs.

FG uses pthreads to overlap work in the pipeline. The programmer does not write any code associated with pthreads but instead has the simpler task of creating FG-defined objects. FG spawns all the threads, coordinates the semaphores for communication among threads, and kills the threads after the pipeline has completed. The programmer maps one or more functions to each thread; these functions, however, are completely synchronous. The programmer need not take overlap into consideration when writing code with FG. In fact, a programmer with only a rudimentary knowledge of threads can easily produce threaded code in FG.

FG also assumes all aspects of buffer management. It allocates buffers at the start of execution and deallocates them at the end. The programmer need only specify the number and size of buffers. FG also recycles buffers appropriately, so that the programmer need not write code to establish or maintain a global pool of buffers. Moreover, FG introduces a new kind of buffer that we call an *auxiliary buffer*. An auxiliary buffer does not traverse the pipeline, but is simply a block of memory that can be requested by any stage. We have seen that it is sometimes necessary to use a second buffer in a stage, such as one that performs a permutation, and FG supplies auxiliary buffers for this purpose. Finally, FG ensures that buffers traverse the pipeline in sequential order. A stage does not have knowledge of its successor and predecessor stages. Instead, each stage simply calls FG-supplied functions to accept buffers from its predecessor and convey buffers to its successor.

After writing simple, synchronous C or C++ stages, there is little a programmer must do to put together an FG program. FG provides a class to describe a thread, so that the programmer need not interact directly with the pthreads interface, as well as a class to describe a stage. Both of these classes are easy to use. The programmer creates the FG-defined stages and threads necessary for the pipeline. Then the programmer simply assigns the appropriate functions to the stages and maps each stage to a thread. All that is left is to create the pipeline, another FG-provided class. We will show in Section 5 that setting up and running the FG pipeline is quite a bit simpler for the programmer than setting up and running a pipeline with ViC* or with threads.

The preceding description does not tell the whole story of FG. Its capabilities extend well beyond the linear pipeline structures that our three benchmark applications fit. Additional features of FG include multistage threads, multistage repeat, buffer swapping, macros, hard barriers, soft barriers, and implicit threads. Additionally, programmers can incorporate fork-join constructs and directed acyclic graphs into an FG pipeline. FG also uses time-balance strategies to reduce the execution time of a pipeline on the fly. Although we touch on some of these features in Section 5, the details are beyond the scope of this paper.

4. Benchmark applications

In this section, we present our benchmark applications: out-of-core implementations of BMMC permutations, FFT, and columnsort. We implemented the first two programs in ViC* and the third in C code with threads; we implemented all three in FG for comparison. In Section 5, we will show the reductions in source code size and complexity that FG affords for the three programs.

4.1. BMMC permutations

A BMMC permutation is specified by an $n \times n$ characteristic matrix A whose entries are drawn from $\{0, 1\}$ and that is nonsingular over $GF(2)$. That is, multiplication is replaced by logical-and, and addition is replaced by exclusive-or. The following are examples of BMMC permutations: matrix transpose when all dimensions are powers of 2, shuffle and unshuffle permutations, Gray-code permutations, and bit-reversal permutations.

For our BMMC-permutation pipeline, the stages are as follows: a read stage, a first permute stage, a communicate stage, a second permute stage, and a write stage. The read and write stages perform disk I/O. The communicate stage performs interprocessor communication across the cluster. The two permute stages work only within each node's CPU and local memory. We can overlap the five stages, therefore, because the CPU is idle during the read, write, and communicate stages, and it is busy during the two permute stages.

Let us explore the path of a buffer through this pipeline. First, the read stage reads a portion of the data into the buffer from disk. Each item i to be permuted initially belongs to some processor $P(i)$ and has a destination processor $P'(i)$. The first permute stage rearranges the data on each processor so that the items mapped to each target processor are contiguous in local memory. The communicate stage performs interprocessor communication so that each item i moves from $P(i)$ to $P'(i)$. The second permute stage rearranges the data locally on each processor. Finally, the write stage writes the data from buffer to disk.

4.2. FFT

The FFT is a computationally efficient algorithm for computing the discrete Fourier transform of an N-element vector. First, the input undergoes a bit-reversal permutation. Then a butterfly graph of $\lg N$ stages is computed. (We use $\lg N$ to mean $\log_2 N$.) In the sth stage of the butterfly graph, elements whose indices are 2^s apart participate in a butterfly operation [6, Chapter 30].

Figure 3 illustrates our out-of-core FFT implementation. We start with a bit-reversal permutation, for which we use our out-of-core BMMC permutation pipeline as a subroutine. Then there are $\lg N / \lg F$ superlevels, where F is the buffer size. Each superlevel consists of N/F separate "mini-butterflies" (on F elements and with depth $\lg F$) followed by a particular type of BMMC permutation on the entire vector. Each pass of the FFT implementation, therefore, consists of a pipeline with a read stage, a mini-butterfly stage, and a write stage, followed by a subroutine that performs a BMMC permutation. See [8] for details.

4.3. Columnsort

We implemented an out-of-core version of Leighton's columnsort algorithm in C code with threads. Columnsort sorts N items, which are treated as an $r \times s$ mesh. When columnsort completes, the mesh is sorted in column-major order. Columnsort proceeds in eight steps. Steps 1, 3, 5, and 7 are identical: they sort the columns of the mesh. Each of the even-numbered steps performs some fixed permutation on the mesh, but the fixed permutation differs from stage to stage.

Our columnsort implementation makes four separate passes over the data. Each pass performs two of the eight steps of the columnsort algorithm. Each pass also includes a read stage, a write stage, and a communicate stage. Figure 1 illustrates a pass of columnsort. It represents the gen-

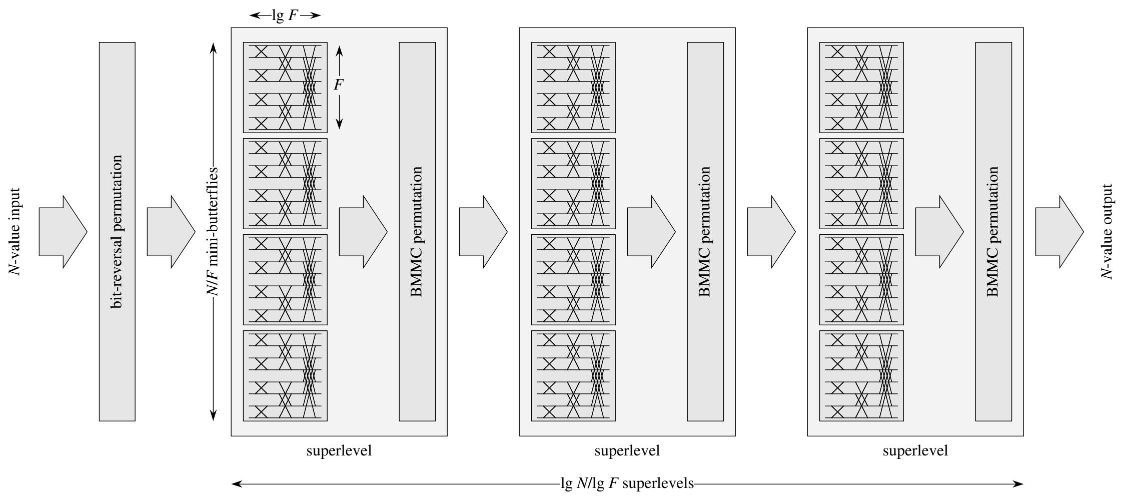


Figure 3: The structure of the out-of-core FFT algorithm. After a bit-reversal permutation, we perform $\lg N/\lg F$ superlevels. Each superlevel consists of N/F mini-butterflies on F values, followed by a BMMC permutation on the entire vector.

	Seconds: 8 GB/proc				Code size			
Program	non-FG	FG	difference	improvement	non-FG	FG	difference	improvement
BMMC	1844	570	1274	69.1%	3204	2736	468	14.6%
FFT	4245	1638	2607	61.4%	7612	6290	1322	17.4%
Columnsort	1893	1862	31	1.6%	7824	5820	2004	25.6%

Table 1: Running times and code size reductions for our three benchmark programs with and without FG. We show the running times, in seconds, and the lines of source code. We also show quantitative differences and percentage improvements. Each time shown is the average of three runs.

eral structure of a pass, although the details of the stages vary with each pass. In our columnsort implementation, the buffer size is equal to the size of one column of the input mesh; every time we read from or write to disk, we transfer exactly one column. Let us explore the path of a buffer through the pipeline. First, the read stage reads one column of the input mesh from disk into the buffer. The sort stage sorts each column. As with BMMC permutations, each item i initially belongs to some processor $P(i)$ and has a destination processor $P'(i)$. Therefore, the communicate stage transmits items among processors so that each item i moves from $P(i)$ to $P'(i)$. The permutation stage permutes the data locally on each processor. Finally, the write stage writes the data from the buffer to disk.

5. Reducing code size and complexity

In this section, we present the reductions in source code size and complexity that FG affords. The authors have shown previously [4] that using FG speeds the execution time of the BMMC permutations, FFT, and columnsort implementations. Table 1 summarizes the running times for the three programs using 8 GB of data per processor on a 16-node cluster. Due to disk-space limitations, 8 GB per processor was the largest problem size that we could test. The authors presented more detailed running-time results in [4], but the focus of the present paper is on code size and not experimental results, and so it suffices to show a representative problem size here.

Table 1 also summarizes the differences in source code size between FG and non-FG programs. For BMMC permutations, using FG reduces source code size by 468 lines, or approximately 14%. For FFT, FG reduces source code size by 1322 lines, or approximately 17%. Finally, for columnsort, FG reduces source code size by 2004 lines, or approximately 25%.

Where do these reductions come from? FG lessens the size and complexity of source code in ViC* programs by eliminating the need for writing asynchronous I/O code. In threaded programing, FG eliminates the need for writing any code associated with the threads. Furthermore, FG takes on all buffer management, a common component of both ViC* and threaded programming.

For our benchmark programs, we separate the code into two parts: glue and base code. Figure 4 illustrates the breakdown between glue and base code in FG and non-FG programs. In FG, the glue does not disappear completely. Instead, the glue is code that we use for setting up, running, and dismantling pipelines, as well as for accepting and conveying buffers. As the figure shows, however, the glue in

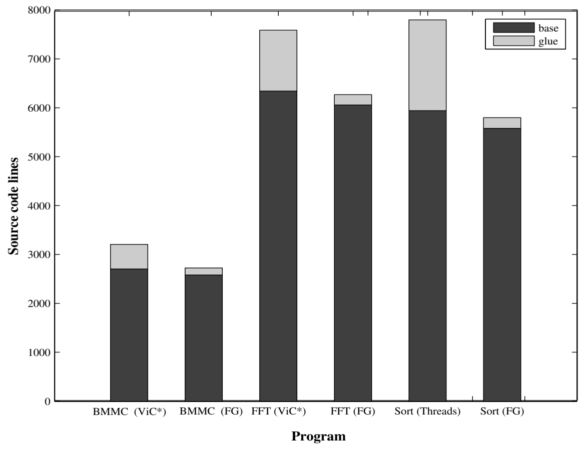


Figure 4: Lines of source code dedicated to glue and to base code for BMMC permutations, FFT, and columnsort, with and without FG. Without FG, the BMMC permutation program uses 502 lines of code for glue, FFT uses 1249 lines, and columnsort uses 1861 lines. With FG, these programs require only 143, 212, and 220 lines of code for glue, respectively.

the FG programs accounts for substantially fewer lines of code than in the non-FG programs.

With the two ViC* programs, the glue is devoted mostly to asynchronous I/O and buffer management. As we show in Figure 4, ViC* requires 502 lines of glue for BMMC permutations and 1249 lines for FFT. These lines amount to 15.59% of the total code for BMMC permutations and 16.41% for FFT. The corresponding FG programs, on the other hand, need only 143 lines of glue for BMMC and 212 lines for FFT, respectively, 5.23% and 3.37% of the total code.

With the threaded program, most of the code reduction comes from setting up and coordinating the threads as well as from managing buffers. In the threaded implementation of columnsort, there is a considerable amount of code devoted to spawning threads and coordinating the concurrent actions among them. Figure 4 shows that, in columnsort implemented with threading, the glue accounts for 1861 lines of code, which is 23.77% of the total. With FG, on the other hand, the glue is reduced to setting up, running, and shutting down the pipeline, which requires only 220 lines of code, or 3.78% of the total.

Reducing source code size is not the only benefit of FG; it lessens the complexity of the code as well. We cannot quantify this claim, but in our experience we have found that the glue FG provides is particularly difficult to write and debug. Without FG, the programmer must not only implement the algorithm itself, but also write the code to make the implementation run efficiently in an HEC environment. With FG, the programmer writes little glue, and the functions are straightforward and synchronous. In our experience, the great majority of base code is far simpler to write than the code to overlap operations. Writing the code for buffer management is onerous as well. Furthermore, we have found that it is especially difficult to debug the glue. Particularly in the threaded programs, for which standard debugging tools are not reliable, finding errors in the glue often proves to be a substantial burden.

FG also allows for easy structural experimentation. When writing HEC programs, a programmer often searches for small changes to improve performance. Reducing running time by even a small percentage can be important, and altering the structure of a pipeline can reduce running time. For example, a programmer might map more than one stage to a single thread—mapping a read stage and a write stage to a single I/O thread since the two operations serialize at the disk anyway. Without FG, replacing a one-to-one mapping of stages to threads by a many-to-many mapping entails considerable time and effort. Figure 5 shows that, with FG, it requires only a few lines of code to make the change. Moreover, it is just as easy to revert to the former mapping if the change does not prove effective. FG can simplify the use of threads even further—it is possible to write a program in FG without explicitly creating threads at all. A programmer can simply create the stages of a pipeline, and FG creates a one-to-one mapping from the stages to threads.

FG also has functionality to find performance improvements. Since the best performance generally comes from a time-balanced pipeline, FG monitors the progression of buffers from stage to stage to determine whether any one stage processes buffers more slowly or more quickly than others. FG searches for any stage that becomes a *bottleneck stage*—it has more buffers in its queue than other stages—and replicates it in another thread. It also searches for any stage that becomes a *spewing stage*—it processes buffers more quickly than other stages—and lowers the priority of

```
FG_pipeline_thread_helper io_thread = new FG_pipeline_thread_helper_info();
FG_pipeline_stage_helper
  read_stage = new FG_pipeline_stage_helper_info(io_thread),
  write_stage = new FG_pipeline_stage_helper_info(io_thread);
```

Figure 5: Mapping a read stage and write stage to one single I/O thread in FG. To split the stages between two threads, we would simply need to create one new thread and make a one-to-one mapping—a change that would require about one additional line of code.

```
stage1->replicate();
stage2->lower_priority();
```

Figure 6: Using stage replication and thread priority adjustments in FG. These two lines of code can yield performance improvements of up to approximately 4%.

its thread. We have found that the speed gains from these run-time techniques are modest—up to 4% at best. For such a small gain, it may not be worth the time and effort to implement them by hand. As Figure 6 shows, however, with FG, it requires little effort on the part of the programmer.

6. Conclusion

We conclude with a discussion of related work and our future plans for FG.

6.1. Related work

StreamIt [13] is a high-level language for stream programs that provides an abstraction for manipulating streams of word-size entities. One of the goals of the project is to simplify the programming of these streaming applications. To enable simpler code, it represents an algorithm as a hierarchical network of filters. It has a graphical editor to represent the hierarchy of components in a user-friendly way. With the StreamIt graphical editor, the programmer initially sees the top level components, and upon clicking on a component, it expands into its subcomponents. Another click causes the hierarchy to contract. FG and StreamIt share some common structures, and both projects attempt to simplify source code, but StreamIt is strictly for streaming applications.

StreamBit [12] is an optimizing compiler for StreamIt that targets bit streaming applications such as cryptography. It enables the programmer to produce a piece of code simply by sketching it. A sketch is a partial specification of the full implementation, and StreamBit derives the missing details. StreamBit uses this sketching capability to improve productivity for transforming a functional specification (written by a domain expert) to an optimization specification (written by a system expert). The domain expert writes an algorithm in a high-level domain-specific language, and the system expert optimizes it for the specific system. Although FG also simplifies programming, it does not use a sketch of code as StreamBit does. Rather, the programmer writes C or C++ code, and FG hides the complexity inherent in making the code run efficiently.

6.2. Future work

We have shown in the past that FG speeds execution time for the three benchmark programs presented here. In our future work, we plan to conduct usability studies to investigate FG's third goal: reducing time-to-solution. We plan to hold a programming case study with Dartmouth undergraduates who are familiar with threads. Each of the subjects will receive the same threaded programming assignment. Half of them will code with FG, and the other half will code with threads explicitly. We will use this setup to measure the time-to-solution for FG and non-FG programs.

References

[1] Geeta Chaudhry and Thomas H. Cormen. Getting more from out-of-core columnsort. In *4th Workshop on Algorithm Engineering and Experiments (ALENEX 02)*, pages 143–154, January 2002.

[2] Geeta Chaudhry, Thomas H. Cormen, and Leonard F. Wisniewski. Columnsort lives! An efficient out-of-core sorting program. In *Proceedings of the Thirteenth Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 169–178, July 2001.

[3] Alex Colvin and Thomas H. Cormen. ViC*: A compiler for virtual-memory C*. In *Proceedings of the Third International Workshop on High-Level Parallel Programming Models and Supportive Environments (HIPS '98)*, pages 23–33, March 1998.

[4] Thomas H. Cormen and Elena Riccio Davidson. FG: A framework generator for hiding latency in parallel programs running on clusters. In *Proceedings*

of the 17th International Conference on Parallel and Distributed Computing Systems (PDCS-2004), pages 137–144, September 2004.

[5] Thomas H. Cormen and Melissa Hirschl. Early experiences in evaluating the Parallel Disk Model with the ViC* implementation. *Parallel Computing*, 23(4–5):571–600, June 1997.

[6] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. The MIT Press and McGraw-Hill, second edition, 2001.

[7] Thomas H. Cormen, Thomas Sundquist, and Leonard F. Wisniewski. Asymptotically tight bounds for performing BMMC permutations on parallel disk systems. *SIAM Journal on Computing*, 28(1):105–136, 1999.

[8] Thomas H. Cormen, Jake Wegmann, and David M. Nicol. Multiprocessor out-of-core FFTs with distributed memory and parallel disks. In *Proceedings of the Fifth Workshop on I/O in Parallel and Distributed Systems (IOPADS '97)*, pages 68–78, November 1997. Also Dartmouth College Computer Science Technical Report PCS-TR97-303.

[9] Elena Riccio Davidson and Thomas H. Cormen. *Asynchronous Buffered Computation Design and Engineering Framework Generator (ABCDEFG): Tutorial and Reference*. Dartmouth College Department of Computer Science. Available at http://www.cs.dartmouth.edu/FG/.

[10] IEEE. Standard 1003.1-2001, Portable operating system interface, 2001.

[11] Tom Leighton. Tight bounds on the complexity of parallel sorting. *IEEE Transactions on Computers*, C-34(4):344–354, April 1985.

[12] Armando Solar-Lezama and Rastislav Bodik. Templating transformations for bitstream programs. In *First Workshop on Productivity and Performance in High-End Computing (P-PHEC)*, pages 27–37, February 2004.

[13] StreamIt Language Specification, Version 2.0. http://www.cag.lcs.mit.edu/streamit/papers/streamit-lang-spec.pdf, February 2003.

A Productive Programming Environment for Stream Computing

Kimberly Kuo, Rodric M. Rabbah, Saman Amarasinghe
Computer Science and Artificial Intelligence Laboratory
Massachusetts Institute of Technology
Cambridge, MA 02139
{kkuo, rabbah, saman}@csail.mit.edu

Abstract

This paper presents StreamIt and the StreamIt Development Tool. The development tool is an IDE designed to improve the coding, debugging, and visualization of streaming applications by exploiting the ability of the StreamIt language to naturally represent streaming codes as structured, hierarchical graphs. The StreamIt Development Tool aims to emulate the best of traditional debuggers and IDEs while moving toward hierarchical visualization and debugging concepts specialized for streaming applications. As such, it provides utilities for stream graph examination, tracking of data flow between streams, and deterministic execution of parallel streams. These features are in addition to more conventional tools for creating and editing codes, integrated compilers, setting breakpoints, watchpoints, and step-by-step program execution.

A user study evaluating StreamIt and the development tool was held at MIT during which participants were given erroneous programs and asked to resolve the programming errors. We compared the productivity of the users when using the StreamIt Development Tool and its graphical features to those who were restricted to line-oriented debugging strategies, and we found that the former produced ten more correct solutions compared to the latter set of users. Furthermore, our data suggests that the graphical tool chain helped to mitigate user frustration and encouraged participants to invest more time tracking and fixing programming errors.

1 Introduction

The last few years have witnessed the rebirth of supercomputing as computer scientists and engineers realize that current monolithic architectures and conventional von Neumann programming styles are at their limits in terms of deliverable performance to the end-user. Thus as architects, compiler engineers, and application developers look into the future, there is a concerted effort to develop processors and programming paradigms that can deliver significantly better performance, and more so, to deliver high performance more productively. This is especially important since the complexity of applications continues to increase, and compilers are more heavily burdened with the extraction of parallelism and the efficient mapping of computation to physical substrate. What's more is that the architectures of the future will tend toward distributed resources in an effort to manage the complexity of centralized architectures with respect to power and wire delay. Thus, research labs in industry and academia alike are investigating ideas and methodologies to address the computing challenges of the future with an eye toward delivering high performance and to do so productively. This goal translates to (i) relieving application developers from architecture details and allowing for natural expression of applications, (ii) lessening the burden for heroic compilers that extract parallelism, (iii) developing scalable architectures that are powerful yet easier to verify and assemble.

The Computer Architecture Group at MIT has for the last several years conducted research to address all of the aforementioned objectives. This paper focuses on the productivity of application developers. Specifically, the paper will briefly describe StreamIt [11] a novel language for the prevalent application class of stream computing. StreamIt provides high-level stream abstractions that improve programmer productivity and program robustness. The language is architecture independent, and it features several characteristics (such as parameterization and modularity) geared toward large scale program development. Furthermore, this paper will also describe a unique development environment that leverages the language features to deliver a tool chain for the rapid verification and debugging of StreamIt programs.

StreamIt represents a program as a hierarchical graph of concurrent filters that operate on streams of data and communicate via FIFO queues. The language exposes the parallelism and communication patterns that are inherent in many streaming programs which include software radio, real-time encryption, network processing, graphics, and multimedia editing consoles. Because of the abundance of parallelism in such applications, they are especially challenging to program, and worse, to debug. This is due to the multitude of factors that an application developer must consider when implementing a streaming program, such as for example how to exploit the parallelism on a target architecture. The marriage of implementation to a specific processor results in both algorithmic changes and code transformations that make porting difficult—since the transformations depend on the architecture details.

By contrast, application developers using StreamIt focus on specifying the functional behavior of their programs and verify correctness using high level abstractions that result in clean and portable implementations. The task of optimizing the code and efficiently mapping it to target processors is left to the compiler which can automate many powerful domain specific optimizations to deliver high performance [5, 9]. This paper will not discuss the StreamIt compiler technology; the interested reader can visit the StreamIt web page [10] for more information on the topic.

In addition to the language and compiler effort, we have engineered and developed a programming environment that graphically represents the hierarchical nature of streaming codes with an eye toward the productivity of the application engineer. The StreamIt Development Tool (SDT) provides an elaborate prototyping and debugging environment that can interpret and visually represent streaming computation. The key distinguishing features of the SDT are its ability to track the flow of data between streams, and the deterministic execution of parallel streams. The latter leverages an intuitive concept of time in StreamIt that is tied to the flow of data in distributed programs. A significant portion of this paper is dedicated to evaluating the SDT and its impact on programmer productivity. Toward quantifying productivity, we organized a user study at MIT. The study involved a number of students who were given a set of "buggy" applications and asked to fix the codes according to corresponding functional specifications. Some of the study participants were allowed to use the graphical debugger and its distinguishing features, whereas others were restricted to line-oriented debugging strategies. The results of our study provide evidence that the SDT was instrumental in helping the users track down and repair programming errors. The evidence is particularly strong in cases where the applications were large, with many streams and non trivial communication topologies.

As we analyzed the data from the user study, we made a somewhat surprising observation. First, it was evident that the SDT did not make users faster. In fact, the mean time to solution (i.e., a program where all of the bugs are fixed) was longer for participants using the graphical debugger. Perhaps this is to be expected since the participants did not have prior experience with the language or the IDE, and indeed our post-study interviews and feedback support this theory. Second, the data suggested that the power of the SDT is in mitigating the frustration factor of the participants, especially in the later portions of the study. That is, the participants who were restricted to line-oriented debugging strategies gave up more often, and did so sooner, compared to their counterparts using the graphical debugger. This led us to conclude that users tend to be more productive when they trust the tools at their disposal. In other words, one might believe their probability of success is reasonably high if they are confident that the tools they are using are adequate, and therefore they are more likely to invest their time objectively.

In the following Section we describe the StreamIt programming language, and in Section 3 we describe the StreamIt development environment. Section 4 describes our user study and reports our results and analysis. Section 5 summarizes related work and Section 6 concludes the paper.

2 The StreamIt Programming Language

StreamIt is an architecture-independent language for streaming applications. It adopts the Cyclo-Static Dataflow [1] model of computation which is a generalization of Synchronous Dataflow [7]. StreamIt programs are represented as graphs where nodes represent computation and edges represent FIFO-ordered communication of data over tapes.

The basic programmable unit in StreamIt is a filter. Each filter contains a work function that executes atomically, popping (i.e., reading) a fixed number of items from the filter's input tape and pushing (i.e., writing) a fixed number of items to the filter's output tape. A filter may also "peek" at a given index on its input tape without consuming the item; this makes it simple to represent computation over a "sliding-window". The push, pop, and peek rates are declared as part of the work function, thereby enabling the compiler to construct a static schedule of filter firings [6].

StreamIt provides three hierarchical structures for composing filters into larger stream graphs (see Figure 1). The *pipeline* construct composes streams in sequence, with the output of one connected to the input of the next. The *splitjoin* construct distributes data to a set of parallel streams, which are then joined together

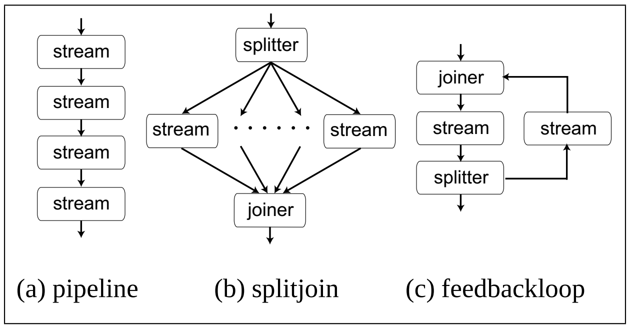
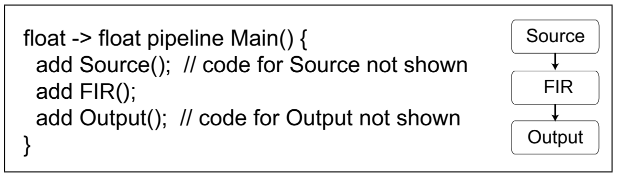


Figure 1. StreamIt containers.



```
float -> float pipeline Main() {
  add Source();  // code for Source not shown
  add FIR();
  add Output();  // code for Output not shown
}
```

Figure 2. Example pipeline with FIR filter.

in a round robin fashion. The *feedback loop* provides a mechanism for introducing cycles in the graph. An example of a pipeline appears in Figure 2. It contains a single FIR (Finite Impulse Response) filter which could be implemented as follows:

```
float->float filter FIR (int N, float[] weights)
{
  work push 1 pop 1 peek N {
    float sum = 0;
    for (int i = 0; i < N; i++) {
      sum += peek(i) * weights[i];
    }
    pop();
    push(sum);
  }
}
```

The filter can now serve as a module that is incorporated into stream graphs as necessary, for example as part of an acoustic beam former. A filter is akin to a class in object oriented programming with the work function serving as the main method. A filter may also declare a constructor function to initialize the filter state before any other method is invoked. The implementation of the work function in StreamIt obviates the need for explicit buffer management. The application developer instead focuses on the hierarchical assembly of the stream graph and its communication topology.

3 Development Environment

The StreamIt Development Tool (SDT) features many aspects of an IDE, including a text editor and a debugger. For example, the SDT debugger supports line and method breakpoints, watchpoints, program suspension, code stepping, variable inspection and value modification to list a few.

Moreover, the SDT offers features tailored to the StreamIt language. The SDT graphically represents StreamIt programs, and preserves hierarchical information to allow an application engineer to focus on the parts of the stream program that are of interest. In addition, the SDT can track the flow of data between filters, and most importantly, it provides a deterministic mechanism to debug parallel streams.

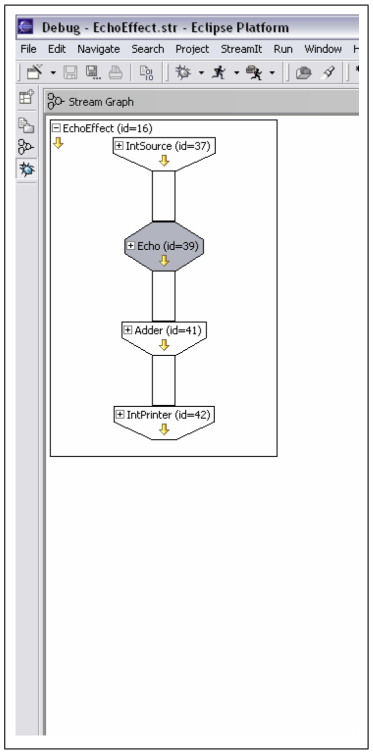
The SDT is implemented in Java as an Eclipse [3] plug-in. The Eclipse universal tools platform is an extensible development environment. We leverage the built-in user interfaces for editing and viewing files, the resource management system, the documentation infrastructure, and the runtime support of launching, running and debugging programs.

3.1 Hierarchical Graphs

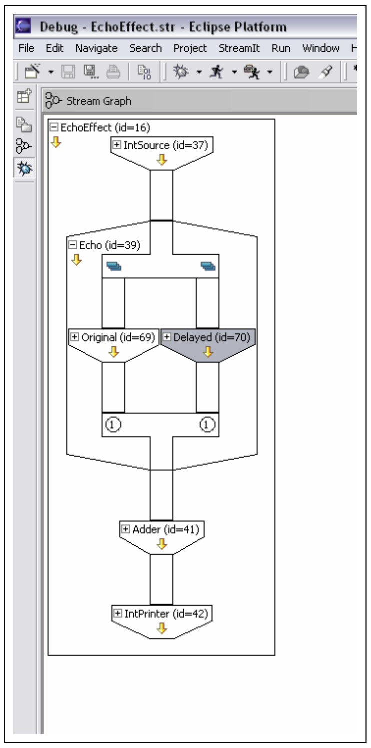
As seen in Figure 3, a StreamIt program can be visually depicted as a hierarchical directed graph of streams, with graph nodes representing streams and graph edges representing tapes or channels. The containers are rendered according to the code declarations, and the visualization tools in the SDT allow the user to selectively collapse and expand containers. This is useful in large streams where the application developers are only interested in visualizing a particular subset, for example to verify the interconnect topology of the graph. In Figure 3(a), we show a screen shot of the SDT for a simple StreamIt program which consists of a filter that generates input data (`IntSource`), a splitjoin (`Echo`) that operates on the data produced by the source and whose data is in turn consumed by an `Adder`. Lastly, a filter (`IntPrinter`) reads and prints the computed values to the screen. In Figure 3(b), the splitjoin is expanded to reveal to parallel streams: `Original` and `Delay`. The former is simply an identity filter, whereas the later shifts its input data one position in time (i.e., at time t it outputs data consumed at time $t + 1$. The splitter in this example is a duplicate splitter, meaning that the input stream is duplicated to all of its siblings. The joiner is a roundrobin joiner which collects one data item from the left stream followed by an item from the right stream. This particular stream program simulates how echos are added to sound waves.

3.2 Data Flow

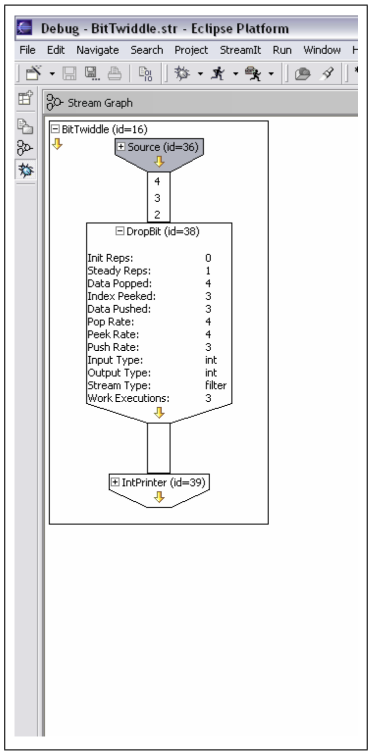
An important distinguishing characteristic of the SDT is its ability to track the flow of data between


Debug - EchoEffect.str - Eclipse Platform
File Edit Navigate Search Project StreamIt Run Window
Stream Graph
EchoEffect (id=16)
IntSource (id=37)
Echo (id=39)
Adder (id=41)
IntPrinter (id=42)

(a) Collapsed pipeline.



(b) Expanded splitjoin.


Debug - BitTwiddle.str - Eclipse Platform
File Edit Navigate Search Project StreamIt Run Window
Stream Graph
BitTwiddle (id=16)
Source (id=36)
4
3
2
DropBit (id=38)
IntPrinter (id=39)

```
Init Reps:          0
Steady Reps:        1
Data Popped:        4
Index Peeked:       3
Data Pushed:        3
Pop Rate:           4
Peek Rate:          4
Push Rate:          3
Input Type:         int
Output Type:        int
Stream Type:        filter
Work Executions:    3
```

(c) Example pipeline with data in transit.

Figure 3. Hierarchical stream graph views.

streams. This is illustrated in Figure 3(c) which shows the data that is live between two filters `Source` and `DropBit`. This particular program generates a sequence of numbers at its source, and the `DropBit` filter removes the third element of the sequence with every execution of its main function. In the figure, the values 2, 3, and 4 are queued on the input tape to `DropBit`, and from the expanded filter node, we can see that the filter requires four queued data items before the work function can execute (i.e., the declared pop rate is 4). The expanded filter node also displays other information such as the input and output types of the stream, as well as profiling information that is useful for debugging.

The SDT also allows the user to highlight and automatically track data items as they propagate between streams. The user can also modify values on a tape, much like a conventional debugger allows users to modify variables and registers.

The flow of data is especially helpful in splitjoins where sequential data streams are distributed to parallel streams, and parallel streams assembled into a single stream. The visualization allows the user to readily verify that splitters and joiners implement the desired functionality. Also, the visualization allows users to quickly pinpoint unexpected outputs (e.g., a filter pushing NaN's).

3.3 Debugging Parallel Streams

Perhaps the most important feature of the SDT is its support for debugging parallel streams. In StreamIt, the streams in a splitjoin are independent, and can execute when their corresponding data are queued. Thus, the SDT can execute parallel streams in a deterministic order using a single program counter and machine state; this is in contrast to a multi-threaded program where a user has to cope with multiple program counters and a scheduling order that may appear non-deterministic and subject to the host operating system. Furthermore, by exposing the flow of data and the communication in a stream graph, StreamIt provides a natural way to reason about time in a distributed system—thereby greatly simplifying the task of debugging parallel streaming programs.

The SDT also features a unique capability that allows a user to set instance-breakpoints. This features is useful in splitjoins with many parallel streams or in long pipelines which contain multiple instances of a single filter. As with conventional debuggers, the program executes until the designated instance of a filter is encountered—in which case control is transfered to the user for further input.

4 Productivity Study

We designed and carried out a user study to assess the extent the SDT helps in debugging StreamIt programs. The goals were two fold. First, we aimed to identify difficulties in using the SDT and toward this end we used questionnaires and automatic action logging. The second goal was to gather data to support the hypothesis that the SDT can improve a programmer's ability to debug StreamIt applications.

We provided participants with a set of "buggy" StreamIt programs, along with verbal descriptions of the programs. The participants were asked to find and fix the errors and to record their experience using various forms and questionnaires. The participants were divided into different groups, some of which used the SDT and its graphical debugging features whereas others did not. Our results and analysis are reported in the following sections.

4.1 Target Population

We solicited participants for the user study by advertising it to MIT students majoring in computer science. We favored students who specialize in communications, signal processing, computer systems and architecture, and who are experienced in popular imperative languages (e.g., C, C++, Java). The nature of study was not explicitly divulged in our solicitation; this served to prevent potential users from learning about StreamIt and becoming familiar with the SDT prior to the study. The participants were awarded a small monetary gift upon completion of the study.

4.2 Methodology

Each participant in the user study was presented with a set of documents that described the tasks of the study and which served to record information from the participants during the study. The documents were:

1. Pre-Study Questionnaire: This document was designed to gather information on the participant's programming background and skill level. Questions such as year in school, major, degree being sought, area of computer science concentration, relevant classes, language proficiency, application development experience, and background in DSP, IDE, and the SDT were asked.

2. StreamIt Language Tutorial: This written presentation was intended to give a cursory introduction to the StreamIt language. It described and illustrated the syntax and semantics of the StreamIt language. Furthermore, example toy applications and tips on the most common mistakes new StreamIt programmers are likely to make were included.

3. SDT Tutorial: Another written presentation, this document was aimed at informing users of the essential features of the SDT. The first part of the tutorial described the functionality of the StreamIt editor and debugger. The second part of the document contained step-by-step instructions on how to compile, run, and debug a sample application.

4. User Tasks: This document instructed users to debug nine StreamIt applications in a specific order. Each of the nine programs contained one or more bugs. As the users moved from one program to the next, they were asked to record their start and end times, the debugging methods they used (e.g., code inspection, print statements, graphical debugger), and a short diagnosis of the program bugs they uncovered.

5. Description of Applications and Code: This document contained a description of each application (numbered 1 through 9), a code listing, a sample buggy output, and a sample correct output. The applications are summarized in Table 1.

6. Post-Study Questionnaire: This document was designed to gather data pertaining to the participant's experience, such as the perceived difficulty of each problem, a general description of how the user debugged each application, user satisfaction, ability to learn and recall various features of the SDT, etc.

In order to minimize biased effects on a programmer's debugging ability, and to ensure internal validity, users were grouped into four categories. All users were asked to debug application 1 without the SDT's graphical features. The participants were then asked to debug application 2 using the SDT and its graphical features. These "control" experiments served to create a baseline

Table 1. Applications used in the productivity study.

1. Bit Twiddle	Removes every third bit from a 96 bit stream.
2. Fib	Generates a Fibonacci sequence using a feedback loop.
3. Echo Effect	Simulates how echos are introduced into sound waves. Uses a splitjoin with two parallel streams.
4. Merge Sort	Implements a merge sort algorithm using 16 parallel streams.
5. Cornerturn	Implements a matrix transpose using a splitjoin to exchange the rows and columns. Stresses the visual tracking of data.
6. Echo Effect2	Alternate implementation of Echo Effect using a feedback loop.
7. Bubble Sort	Implements a bubble sort algorithm. This is a conceptually difficult implementation that stresses the visualization features of the debugger.
8. Bit Reverse	Sorts a sequence of 16 consecutive numbers in bit-reversed order. This is an adaptation of the bit-reversal stage in FFT.
9. Overflow	A synthetic benchmark with a substantial number of hierarchies, filters, and parallel streams. Stresses the visual tracking of data, and the instance breakpoint capabilities of the debugger.

reference for meaningful comparison later on[1]. Moreover, the control applications were designed to bolster the user's confidence. Next, half of the users (group A) were told to debug applications 3, 4, and 5 with the SDT and 6, 7, and 8 without the SDT (i.e., using the graphical features of the SDT then without the graphical features). Meanwhile, the other half (group B) were told to debug 3, 4, and 5 without the SDT and 6, 7, and 8 with the SDT. Due to this grouping structure, applications 3 and 6, 4 and 7, and 5 and 8 were designed to be of comparable difficulty. For application 9, half of group A (A1) and half of group B (B1) were asked to debug with the SDT, while the other halves (A2 and B2) were asked to debug without the SDT. Cross-sectioning the groups was aimed at ensuring external validity.

The study was divided into three sessions over a three-day period. We estimated that each session would last for two hours (with 45 minutes spent on the tutorials and the rest of the time dedicated to debugging), but in reality the sessions spanned an average of four hours. Many users were either unable or did not have enough time to debug certain applications. Participants were asked to complete the set of documents at their own pace, and upon completion, they were individually interviewed and received a $40 gift certificate. During the study, users were encouraged to ask questions although particulars relating to the problems and the SDT were not revealed.

[1]The control experiments served mainly to filter data. We did not use data attributed to participants who did not complete the control experiments.

4.3 Results and Analysis

Even though 25 users were scheduled to participate (5 people for sessions 1 and 2 and 15 people for session 3), cancellations reduced the participation to 20 users and led to uneven groupings. There were 6 people in A1, 5 in A2, 4 in B1, and 5 in B2. Of the 20 participants, there were 4 juniors, 2 seniors, 8 masters, and 6 Ph.D. students, all majoring in Electrical Engineering and Computer Science. None of the users had prior StreamIt or SDT experience.

Figure 4 summarizes the study in terms of the number of solutions reported for each of the applications in the study. In the figure, the bars labeled "solved with the SDT" represent the number of participants that fully debugged the corresponding applications using the SDT and its graphical features. Similarly, the bars labeled "solved without the SDT" represent the number of participants that fully debugged the corresponding applications without using the graphical debugger. The bars that are labeled "unsolved" represent the number of participants whose applications remained buggy. For example, for the application `EchoEffect` there were two users who were allowed to used the SDT and were unable to debug the code properly. There was also one other participant who did not debug `EchoEffect` although this user was not allowed to use the SDT's graphical features.

Because the groupings are uneven as previously mentioned, the numbers seen in the figure are weighted depending on which group is lacking users. The percentage above each quadruple of columns represents the percentage increase or decrease in debugged applications due to the SDT (and its graphical features). For example, the graphical debugger did not particularly help in appli-

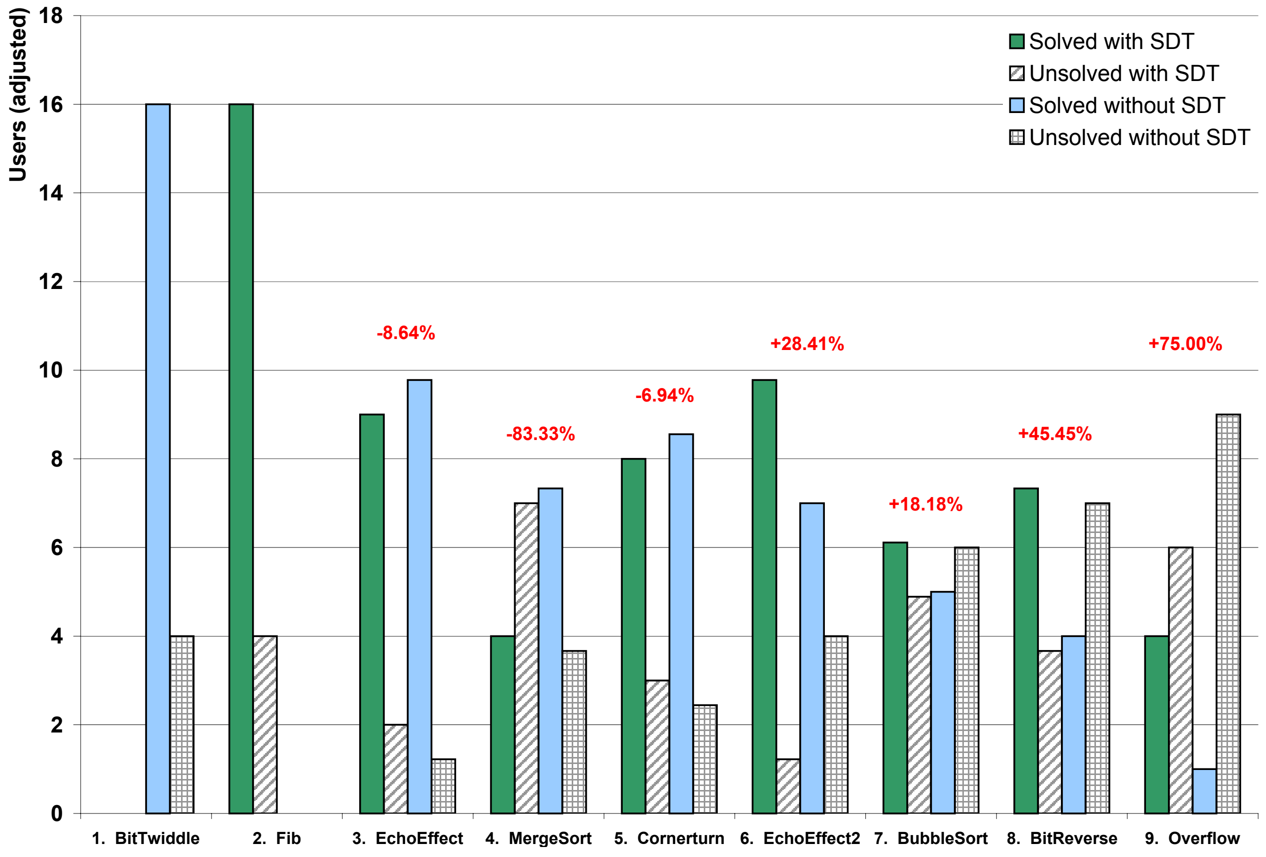


Figure 4. Summary of results.

cations 3, 4, and 5, but did help in debugging the others. On average, 1.56 fewer participants fully debugged applications 3, 4, and 5 when using the graphical debugger, and 2.56 more users debugged applications 6, 7, 8, and 9 when using the graphical debugger.

Figure 5 compares the average time spent debugging each application. The percentage above each set of columns represents the percentage improvement or deficiency in time caused by using the SDT. On average, users took 7.78 (36.48%) more minutes to debug applications 3, 4, 5, 6, 7, and 9 when using the graphical features of the debugger, compared to participants using more traditional debugging means. Furthermore, participants who were allowed to use the SDT and its graphical features spent an average of 16.96 more minutes debugging applications 6, 7, and 8, compared to an average of 10.67 minutes invested by the participants who could not use the graphical debugger. In both cases the participants did not fully debug their respective applications.

Summarizing the results, we found that more participants were able to successfully debug their applications when using the SDT and its graphical features. However, we also observed that the SDT increased the "time to solution" as users had to navigate through a user interface they were not familiar with. Interestingly, we can also observe that the SDT may have mitigated user frustration. As noted earlier, users generally spent much more than the two hours allotted to complete the study, and as such, users became frustrated and may have rushed with the later applications. Correspondingly, this might have caused users to spend less time and debug fewer applications as users progressed through the study. Although this pattern is true for participants who did not use the SDT, the opposite occurs for participants who used the SDT: 41.76% more users were able to debug applications 6, 7, 8, and 9 using the SDT. Furthermore, users spent 83.35% more time tracking down bugs in applications 6, 7, and 8 when using the graphical debugger. These results suggest that users are willing to spend more time and work on more problems when using a tool that they felt more certain would lead them to a solution.

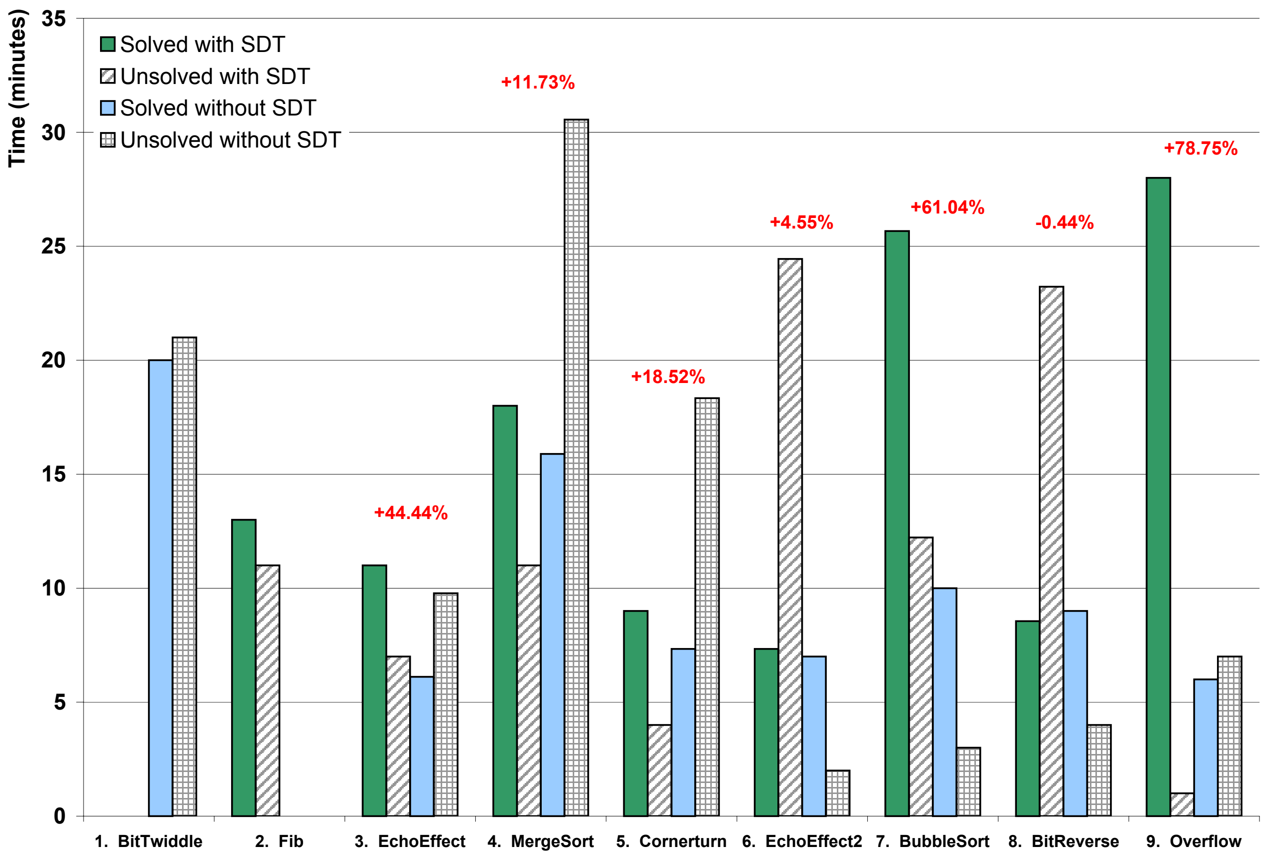


Figure 5. Summary of results.

4.4 User Feedback

Comments obtained from the post-study questionnaires were quite helpful toward finding new feature ideas and problems with functionality, performance, reliability, and usability.

Summarizing the most notable feedback, users largely commented on their difficulties using Eclipse and navigating the StreamIt code. Ten participants reported that the time alloted to learn how to navigate Eclipse was too short, and that the many windows, menus, and options made it difficult to find vital information quickly. Five participants noted that they were uncomfortable or unaccustomed to thinking in terms filters and streams, while an equal number also found it overwhelming to remember some of the language syntax and concepts.

In general however, participants rated the SDT a 3.85 on average (on a scale of 1 to 10, from easy to hard), praising many aspects of the stream graph viewer (e.g., hierarchical representation). Those who rated the SDT as helpful, stressed that it was most useful for graphically visualizing the flow of data in the programs, especially when the applications were large.

4.5 Discussion

Many problems and issues arose in running the study itself. One of the major problems was the time allotted for users to complete the study. As previously mentioned, the slowest user spent twice the budgeted amount of time. The timing negatively impacted users in several ways, all of which contributed to incomplete or unreliable data: Users became frustrated and overwhelmed by the amount of information presented to them; Users were unable to complete the study due to time constraints; Users did not properly fill out the post-study questionnaire, etc.

We believe that better screening can help bridge the gap between between participants, although the biggest lesson learned centers on the method of compensating the participants. Specifically, a multi-level pay scale for compensation may have alleviated some of the above problems and lead to more conclusive results. A graded pay scale would allow each participant to judge whether they can or are willing to complete the study. In other words, each user is rewarded according to their investments. Nonetheless, the expertise gap between participants in a user study is a well-documented issue: Usability studies have found that the best users are often ten times better than the worst users, and the fastest quartile of users are twice as fast as the slowest quartile of users [2, 8]. However, because increasing the number of users in a study only narrows the standard deviation of the mean by the square root of the number of users[2], the improvement in results and reliability becomes an expensive and time-consuming task. For example, in order to double accuracy, the number of participants in this study would have to be quadrupled to 80 users, which would cost an additional $2400 and 24 man-hours.

5 Related Work

Numerous debuggers and program visualization tools exist for DSP applications written in C/C++ and assembly. The majority of these tools are targeted at specific hardware platforms, offering traditional debugging features (i.e., program suspension, breakpoint stepping, watchpoints, local variable and output display, etc.) combined with assembly code, memory register, and signal plot display.

In recent years, some movement in the streaming domain has been made toward OOP languages such as C++ or Java, which introduce abstractions that improve the portability and reusability of code. The introduction of conceptual abstractions empowers the design, debugging, visualization, and analysis tools created for OOP based streaming applications to introduce hierarchical, modular structures while hiding unnecessary details from the programmer. On top of the traditional debugging features previously mentioned, all three of the tools described next use some variation on the theme of signal processing blocks that are connected, displayed, and navigated graphically.

Simulink is a modeling, simulation, and analysis tool for control, signal processing, and communications system design. This tool imposes OOP conventions on Matlab, C, Fortran, and Ada programmers by allowing its users to insert their code into the methods of predefined blocks or to use application-specific standard block libraries. Furthermore, hierarchically block navigation at both the design and debugging stages is offered: command-line Simulink Debugger enables breakpoint stepping of the currently executing method which is simultaneously displayed on its associated block. Additional information, such as block state, inputs, and outputs, are visible in other windows.

Process-Level Debugger (PDG) is designed for a graphical parallel programming environment for concurrent applications called GRAPE. The PDG models processes as black boxes that interact with each other. Like Simulink, programmers build their applications by creating and connecting black boxes hierarchically (i.e., each black box may be composed of sub-boxes–subprocesses–and displayed in a graphical view). As an application is debugged, the PDG shows the application's behavior in a window and allows a programmer to zoom down on suspicious process blocks in the hierarchy. This top-down debugging method can eventually find the associated erroneous code.

The MULTI Integrated Development Environment is designed for multiprocessor, distributed systems and embedded applications using C, C++, Ada, Fortran, and assembly. Besides standard editing and debugging functionality, this IDE conveys program control flow with perusable static and dynamic call graphs and class hierarchies.

Much like other language efforts, StreamIt addresses many software engineering concerns by embracing concepts such as modularity, parameterization, hierarchical composition, and portability. Furthermore, the language automates several tedious tasks such as circular buffer management that is common in streaming codes. StreamIt also facilitates the verification of program via inductive reasoning since simple components are assembled to create large and complex graphs. Moreover, the language treats communication and parallelism as "first-class citizens", and by naturally exposing the flow of data in a program, the StreamIt Development Tool can help application engineers in their debugging and verification tasks.

A more thorough treatment of related work is available [4] for review by the interested reader.

6 Concluding Remarks

This paper presents StreamIt and the StreamIt Development Tool. The SDT is an IDE designed to improve the coding, debugging, and visualization of streaming applications by exploiting the StreamIt language's ability to naturally represent these applications as structured, hierarchical graphs. Although industry and academia have devoted much effort to tools for developing and debugging software, the SDT aims to emulate the best of traditional debuggers and IDEs while moving toward hierarchical visualization and debugging concepts specialized for streaming applications. As such, it provides utilities for stream graph examination and navigation, and detailed tracking of data between streams, as well as deterministic execution of parallel streams. These features are in addition to program creation and code editing, program compilation and launch support, and general debugging and help support.

A user study evaluating the SDT uncovered several problems and areas of improvement that need to be addressed before this tool can fully realize its goals. From the user study however, we have empirical evidence to suggest that the SDT improved the ability of users to find and repair programming errors. The user study also provided key insights that suggest that application developers and engineers are more likely to invest their time tracking bugs and enhancing their applications if they are confident they have adequate tools at their disposal. In our user study, several subjects using the StreamIt graphical debugger spent considerable more time debugging applications in the latter parts of the study, compared to subjects who were restricted to line oriented debugging.

For more information on StreamIt, or to download the StreamIt compilation and development infrastructure, please visit the project web page [10].

ACKNOWLEDGMENTS

Special thanks to William Thies who suggested and wrote many of the applications in the study and supplied advice on improving and enhancing the SDT and the study. Both David Maze and William Thies modified the StreamIt Java library to interface with the SDT. Jasper Lin, Juan Carlos, Sitij Agarwal, Jeremy Wong, Michael Gordon, and Michal Karczmarek all participated in in-house evaluations of the SDT. Finally we thanks the MIT students who participated in this study, without whom this paper would not be possible. This work was funded in part by NSF, IBM under the PERCS project, and DARPA under contract No. NBCH30390004.

References

[1] G. Bilsen, M. Engels, R. Lauwereins, and J. Peperstraete. Cyclo-static dataflow. *IEEE Transactions on Signal Processing*, pages 397–408, February 1996.

[2] Controlled Experiments. http://graphics.lcs.mit.edu/classes/6.893/F03/lectures/-L13.pdf.

[3] The Eclipse Universal Tools Platform. http://www.eclipse.org/.

[4] K. Kuo. The StreamIt Development Tool: A programming environment for StreamIt. Master's thesis, Massachusetts Institute of Technology, June 2004.

[5] A. Lamb. Linear analysis and optimization of stream programs. Master's thesis, Massachusetts Institute of Technology, May 2003.

[6] E. Lee and D. Messershmitt. Static Scheduling of Synchronous Data Flow Programs for Digital Signal Processing. *IEEE Trans. on Computers*, C-36(1):24–35, January 1987.

[7] E. A. Lee and D. G. Messerschmitt. Static scheduling of synchronous data flow programs for digital signal processing. *IEEE Transactions on Computers*, January 1987.

[8] Research Issues. http://pages.cpsc.ucalgary.ca/saul/681-/1997/jas/issues.html.

[9] A. Solar-Lezama and R. Bodik. Templating transformations for bitstream programs. In *Proceedings of the HPCA Workshop on Productivity and Performance in High-End Computing (P-PHEC)*, Madrid, Spain, 2004.

[10] The StreamIt Project. http://cag.csail.mit.edu/streambit/.

[11] W. Thies, M. Karczmarek, and S. Amarasinghe. StreamIt: A Language for Streaming Applications. In *Proceedings of the International Conference on Compiler Construction*, Grenoble, France, 2002.

X10: an Experimental Language for High Productivity Programming of Scalable Systems
(Extended Abstract)

Kemal Ebcioğlu   Vijay Saraswat   Vivek Sarkar

IBM Research
T.J. Watson Research Center
P.O. Box 704, Yorktown Heights, NY 10598

{kemal,vsaraswa,vsarkar}@us.ibm.com

Abstract

It is well established that application development productivity is a significant bottleneck in the time to solution for obtaining production applications on High-End Computing (HEC) systems. Previously, we introduced a simple model for defining application development productivity in the presence of multiple expertise levels, and used this model to motivate the programming model and tools solution being pursued in the IBM PERCS project [8]. In this paper, we describe X10*, an experimental language that embodies a new parallel programming model serves as the foundation for multiple productivity-improving technologies in PERCS ranging from visualization and refactoring tools to static and dynamic optimizing compilers.*

1 Introduction and Motivation

The key challenges faced by current and future-generation large-scale systems are 1) *Scalability:* the ability to effectively utilize multiple levels of available parallelism in a high system, such as clusters, SMPs, multiple cores on a chip, co-processors, SMT, and SIMD levels, and 2) *Non-uniform data access:* the ability to support a global data model in the presence of severe nonuniformities in latency, bandwidth and interfaces for accessing data in different parts of the system. It is now common wisdom that the ongoing increase in hardware complexity of large-scale parallel systems to address these challenges has been accompanied by a *decrease in software productivity* for developing, debugging, and maintaining applications for such machines [10]. This is a serious problem because current trends for next generation systems, including SMP-on-a-chip and tightly coupled "blade" servers, indicate that these complexities will be faced not just by programmers for large-scale parallel systems, but also by mainstream application developers.

In the area of scientific computing, the programming languages community responded to these challenges with the design of several programming languages, including Sisal, Fortran 90, High Performance Fortran, Kali, ZPL, UPC, Co-Array Fortran, and Titanium. The ultimate challenge facing this community is supporting *high-productivity, high-performance programming*: that is, designing a programming model that is simple and widely usable (so that hundreds of thousands of application programmers and scientists can write code with felicity) and yet efficiently implementable on current and proposed architectures without requiring "heroic" compilation efforts. This is a grand challenge, and past languages, while taking significant steps forward, have fallen short of this goal either in the breadth of applications that can be supported or in the ability to deliver the underlying performance of the target machine. MPI still remains the most common model used in obtaining high performance on large-scale systems, despite the productivity limitations inherent in its use.

During the same period, significant experience has also been gained with the widespread adoption of object-oriented languages, such as JAVA and C#, that are executed on *virtual machines* and *managed runtime environments*. These languages, along with their accompanying libraries, frameworks and tools, have enjoyed much success in improving *productivity* for commercial applications.

X10 is an experimental new object-oriented language for high performance computing that is currently under development at IBM in collaboration with academic partners.

The X10 effort is part of the IBM PERCS project (Productive Easy-to-use Reliable Computer Systems) whose goal is to design adaptable scalable systems for the 2010 timeframe. The PERCS technical agenda is focused on hardware-software co-design that combines advances in chip technology, computer architecture, operating systems, compilers, programming environments and programming language design. The main role of X10 is to simplify the programming model so as to increase the programming productivity for future systems like PERCS, without degrading performance. Combined with the PERCS Programming Tools agenda [8], the ultimate goal is to use a new programming model and a new set of tools to deliver a 10$\times$ improvement in development productivity for large-scale parallel applications by 2010.

To manage concurrency and distribution, X10 introduces constructs that are expected to be amenable to automatic static and dynamic optimizations by 2010. Specifically, X10 introduces *atomic sections* in lieu of locks, *clocks* in lieu of barriers, and *asynchronous operations* in lieu of threads. To increase performance transparency, X10 integrates new constructs (notably, *places*, *regions* and *distributions*) to model hierarchical parallelism and non-uniform data access.

X10 is a strongly typed language that emphasizes the static expression of program invariants (e.g. about locality of computation). Such static expression improves both programmer productivity (in documenting design invariants) and performance. The X10 type system supports generic type-abstraction (over value and reference types), is place- and clock-sensitive and guarantees the absence of deadlock (for programs without conditional atomic sections), even in the presence of multiple clocks. X10 specifies a rigorous, clean and simple semantics for programming constructs independently from a specific implementation.

In the remainder of this extended abstract, we present an overview of the X10 design, and use example programs to illustrate some of the individual features in X10.

2 The X10 language design

This section provides a brief summary of the X10 language, focusing on the core features that are most relevant to locality and parallelism. A number of other features in X10 are not mentioned here due to space limitations. These include generic interfaces, generic classes, type parameters, sub-distributions, array constructors, exceptions, place casts and the nullable type constructor.

A central concept in X10 is that of a *place*. A place is a collection of resident light-weight threads (called *activities*) and *data*, and is intended to map to a data-coherent unit in a large scale system such as an SMP node or a single co-processor. It is intended to contain a bounded, though perhaps dynamically varying, number of activities and a bounded amount of storage. Cluster-level parallelism can be exploited in an X10 program by creating multiple places.

There are four storage classes in an X10 program:

1. *Activity-local* — this storage class is private to the activity, and is located in the place where the activity executes. The activity's stack and thread-local data are allocated in this storage class.

2. *Place-local* — this storage class is local to a place, but can be accessed coherently by all activities executing in the same place.

3. *Partitioned-global* — this storage class represents a *unified* or *global address space*. Each element in this storage class has a unique place that serves as its home location, *references* to the element can be manipulated by both local activities (activities in the same place as the element) and remote activities (activities in a different place from the element). However, as discussed below, *accesses* to the element can only be performed by local activities.

4. *Values* — Instances of *value classes* (value objects), are immutable and stateless in X10, following the example of Kava[6]. Such value objects are in this storage class. Since value objects do not contain any updatable locations, they can be freely copied from place to place, and also lend themselves to more extensive compiler optimizations than mutable objects [4]. The choice of when to clone, cache or share value objects is left to the implementation. In addition, methods may be invoked on such an object from any place.

X10 activities operate on two kinds of *data objects*. A *scalar* object has a small, statically fixed set of fields, each of which has a distinct name. The mutable state of a scalar object is located at a single place. It is also worth noting that commonly-used basic types such as `int`, `float`, `complex` and `string` are defined as *value classes* in the `x10.lang` standard library, rather than as primitive types in the language.

An *aggregate* (array) object has many elements (the number may be known only when the object is created), uniformly accessed through an index (e.g. an integer) and may be distributed across many places. Specifically, an X10 array specifies 1) a set of indices (called a *region*) for which the array has values, 2) a *distribution mapping* from indices in this region to places, and 3) the usual array mapping from each index in this region to a value of the given base type (which may itself be an array type). Operations are provided to construct regions (distributions) from other regions (distributions), and to iterate over regions (distributions). These operations include standard set-based operations such as unions and intersections, some of which are available in modern languages such as ZPL [9].

Activities represent lightweight threads in X10. An activity is created in a given place and remains in that place for its lifetime, but each place may have several activities executing in parallel. An activity can recursively spawn additional activities at places of its choosing. Throughout its lifetime an activity executes at the same place, and has direct access only to data stored at that place. Remote data can only be accessed by spawning asynchronous activities at the places at which data is resident. Any attempt by an activity to directly access a non-local datum is made manifest either as a type-checking error during compilation or as a `BadPlaceException` during execution.

The X10 type system is used to catch many common cases

Asynchronous activities have two forms — statements and expressions. The expression form of an asynchronous activity is called a *future*, and is discussed further below. The statement form of an asynchronous activity is `async (P) S` where `S` is a statement and `P` is a place expression. Such a statement is executed by spawning an activity at the place designated by `P` to execute statement `S`. As a convenient means of identifying the place of a datum in the partitioned-global storage class, when the expression `P` specifies an array element or object, it evaluates to the place containing that array element or object. `(P)` can also be omitted, in which case, it is inferred to be the place of the data accessed by statement `S` (provided that a single place can be unambiguously inferred).

For example, the X10 statement,

```
async (A[99]) { A[99] = k }
```

creates a new activity at the place containing element `A[99]` of a global distributed array `A`. The values of local variables such as `k` are passed as implicit parameters to this activity. We believe that the use of implicit parameters aids in productivity, since it relieves the programmer of the burden of encapsulating remote activities as procedure calls with explicit parameters. As an additional productivity aid, X10 also supports an *implicit syntax* for async statements and other constructs e.g., the above example could simply be written as `A[99] = k;`, which denotes the same asynchronous activity to be executed at the place containing `A[99]`. This example illustrates how an `async` statement can be used to accomplish a remote store operation. However, `async` statements can be used as the foundation for many other common programming idioms in HEC application development including fine-grained threads, asynchronous DMA operations, message send (for an active or passive message), and scatter operations.

In addition to the *async* statement, the *foreach* construct serves as a convenient mechanism for spawning local activities across a specified index set (*region*) and the *ateach* (pronounced “at each”) construct serves as a convenient mechanism for spawning activities across a set of local/remote places or objects.

X10 provides five mechanisms for the coordination of activities — *clocks*, *force operations*, *finish operations*, *atomic sections*, and *conditional atomic sections* — which are summarized below.

Clocks Clocks are a generalization of barriers, which have been used as a basic synchronization primitive for MPI process groups and in other SPMD programming models. X10 *clocks* are designed to offer the functionality of multiple barriers in the context of dynamic, asynchronous, hierarchical networks of activities, while still supporting determinate, deadlock-free parallel computation.

A clock is defined as a special value class instance, on which only a restricted set of operations can be performed. At any given time an activity is *registered* with zero or more clocks. The activity that creates a clock, is automatically registered with this clock. An activity may register other activities with a clock, or may un-register itself with a clock. At any given step of the execution a clock is in a given *phase*. The first phase of the clock starts when the clock is created. The clock *advances* to its next phase only when all its currently registered activities have quiesced (either by performing a `next` operation, or by terminating), and all statements scheduled for execution in the current phase have terminated. In this manner, clocks serve as a generalization of *barriers* for a dynamically varying collection of activities. From an activity’s viewpoint, when it performs a `next` operation, it quiesces on *all* the clocks it is registered with, and suspends until all of them have advanced to their next phase.

Force Operations When an activity A executes the statement, `F = future (P) E`, it asynchronously spawns an activity B at the place designed by `P` to evaluate the expression `E`. Execution of the expression in A terminates immediately, yielding a *future* [3] in `F`, thereby enabling A to perform other computations in parallel with the evaluation of `E`. A may also choose to make the future stored in `F` accessible to other activities. When any activity wishes to examine the value of the expression `E`, it invokes a `force` operation on `F`. This operation blocks until B has completed the evaluation of `E`, and returns with the value thus computed. Like `async` statements, `future`’s can be used as the foundation for many other common programming idioms in HEC application development including fine-grained threads, asynchronous DMA operations, message send receive, and gather operations.

Finish Operations When an activity A executes the statement, `finish S`, where `S` is a statement, it is guaranteed that the finish statement will not be completed till all activities that are (recursively) spawned by `S` have terminated. Therefore, `finish` is a convenient operation that can be used to enforce global termination.

Unconditional Atomic Sections A statement block or method that is qualified as `atomic` has the semantics of being executed by an activity as if in a single step, during which all other activities are frozen[1]. An atomic section is a generalization of user-controlled locking, so that the `X10` programmer only needs to specify that a collection of statements should execute atomically and can leave the responsibility of lock management and other mechanisms for enforcing atomicity to the language implementation. Primitives such as fetch-and-add, updates to histogram tables, updates to a bucket in a hash table, airline seat reservations arriving at an online data base, and many others, are a natural fit for coordination using atomic sections. `X10` also requires that each access to shared mutable data (*i.e.*, mutable data that can be accessed by multiple activities) must occur in an atomic section, thereby easing the constraints imposed by the memory consistency model.

Consider the following atomic section as a concrete example:

```
atomic { node = new Node(data, head);
         node.next = head; head = node; }
```

By declaring the statement block as atomic, the programmer is able to maintain the integrity of a linked list data structure in a multithreaded context, while still giving the `X10` system the flexibility of using fine-grained synchronization or even non-blocking implementations.

From a scalability viewpoint, it is important to avoid including long-running or blocking operations in an atomic section. In addition, we call an atomic section *analyzable* if the locations and places of all data to be accessed in the atomic section can be computed on entry to the atomic section. Analyzability of atomic sections is not a language requirement, but serves as an important special case for which optimized implementations of atomic sections can be developed [7].

Conditional atomic sections Conditional atomic sections in `X10` are akin to conditional critical regions [2], and have the form `when (c) S`. If the guard `c` is false in the current state, the activity executing the statement blocks until `c` becomes *true*. Otherwise, as far as any other concurrently executing activity is concerned, the statement is executed *in a single step* which begins with the evaluation of `c` = *true*, and ends with the completion of statement `S`. This implies that `c` is not allowed to change between the time it is detected to be true and the time `S` begins execution. `X10` currently does not permit the statement `S` to contain or invoke a nested conditional atomic section.

A conditional atomic section for which the condition `c` is statically true is considered to be equivalent to an unconditional atomic section.

3 RandomAccess Example

Figure 1 outlines one possible implementation for the RandomAccess HPC Challenge benchmark in `X10`. The group of statements labeled (1) is used to allocate and initialize `table` as a global block-distributed array. Note the definitions of `region` *r* and `distribution` *d*, which provide the foundation for allocating the `table` array. Since the index variable used in the `ateach` construct has the same distribution as the `table` array, it is guaranteed that each access to `table[i]` will be performed by a local activity *i.e.*, by an activity located in the same place as `table[i]`. The use of the `finish` operator ensures that all initialization activities spawned in the `ateach` construct must be completed before execution moves to group (2).

Next, the group of statements labeled (2) is used to allocate and initialize `ranStarts` as a "unique-distributed" array *i.e.*, an array with exactly one element per place, and the group of statements labeled (3) is used to allocate and initialize a *value* array named `smallTable`.

The group of statements labeled (4) defines the core computational kernel of RandomAccess, with one activity per place that executes a long running sequential loop, (5). Each iteration of the loop performs an `async` statement on the place containing `table[j]` group of statements labeled (2) , and the async statement performs an atomic read-exor-write operation on `table[j]`.

Finally, the statement labeled (6) performs a sum reduction on `table[]`, and compares the sum value with an expected result.

4 Jacobi Example

Figure 2 outlines one possible implementation for the Jacobi example program in `X10`. The group of statements labeled (1) is used to create a block distribution, `D`, a second distribution, `D_inner`, that contains only the interior elements of `D`, and a third distribution, `D_boundary`, that contains all the remaining elements. Next, the group of

[1]The implementation may of course allow concurrent execution of atomic sections, using techniques such as non-blocking algorithms and optimistic concurrency, as long as atomic sections are made to appear to execute in a "single step" to the rest of the program.

```
public boolean run() {
    // (1) Allocate and initialize table as a block-distributed array
    final region r = new region(0,TABLE_SIZE-1);
    final distribution d = distribution.block(r);
    ranNum[d] table = new ranNum[d];
    finish ateach(int i:d) {table[i]=new ranNum(i);}

    // (2) Allocate and initialize ranStarts as a unique-distributed array
    // with one random number seed for each place
    final distribution d2= distribution.unique(place.places);
    ranNum[d2] ranStarts = new ranNum[d2];
    finish ateach(int i:d2) {ranStarts[i]=new ranNum(...);}

    // (3) Allocate a small immutable table that can be copied on all processors
    // and is used in generating the update values
    final region r3=new region(0,SMALL_TABLE_SIZE-1);
    final place valuePlace=(1).place;
    final distribution d3=distribution.constant(r3,valuePlace);
    value ranNum[d3] smallTable = new ranNum[d3];
    foreach(int i:r3) {smallTable[i]=new ranNum(i*SMALL_TABLE_INIT);}

    // (4)In all places in parallel, repeatedly generate random table indices
    // and perform atomic read-modify-write operations on corresponding table elements
    finish ateach (point p : ranStarts.distribution) {
       long ran = nextRandom(ranStarts[p]);
       // (5) Sequential loop
       for (int count=1; count<=N_UPDATES_PER_PLACE; count++) {
           final int j = f(ran);
           final long k = smallTable[g(ran)];
           async(table.distribution[j]){atomic{table[j]^=k};}
           ran = nextRandom(ran);
       }
    }

    // (6) Return true iff sum of elements in table[] matches expected result
    return table.reduce(ranNum.add,0)==EXPECTED_RESULT;
}
```

Figure 1: RandomAccess example in X10

```
/**
  * Jacobi iteration
  *
  * At each step of the iteration, replace the value of a cell with
  * the average of its adjacent cells in the (i,j) dimensions.
  * Compute the error at each iteration as the sum of the changes
  * in value across the whole array. Continue the iteration until
  * the error falls below a given bound.
  *
  */

 public class Jacobi  {
     . . .

     // (1) Create distributions D, D_inner and D_boundary
     final region R=new region(new region(0,N+1),
                               new region(0,N+1));
     final region R_inner=new region(new region(1,N),
                                     new region(1,N));
     final distribution D = distribution.block(R);
     final distribution D_inner = D.restriction(R_inner);
     final distribution D_Boundary = D.difference(D_inner);

     public boolean run() {
         int iters = 0;
         // (2) Initialize array b
         double[D] b= new double[D];
         finish ateach(point [i,j]:D_inner) {b[i,j]=(double)i*N+j;}
         finish ateach(point [i,j]:D_boundary) {b[i,j]=0.0;}
         while(true) {
             // (3) Create array temp, and overlay it with array b
             final double[D_inner] temp = new double[D_inner];
             finish ateach(point [i,j]:D_inner) {
                 temp[i,j]=(b[i+1,j]+b[i-1,j]+b[i,j-1]+b[i,j+1])/4.0;}
             if ( b.restriction(D_inner).lift(double.sub,temp).lift(double.abs)
                   .reduce(double.add,0.0) < epsilon) break;
             b = b.overlay(temp);
             iters++;
         }

         // (4) Validate correctness of the run
         return b.reduce(double.add,0.0)==EXPECTED_CHECKSUM &&
                iters==EXPECTED_ITERS;
     }
 }
```

Figure 2: Jacobi example in X10

statements labeled (2) is used to allocate and initialize array `b`. Note the use of different initialization statements for the inner and boundary elements.

The statements in (3) creates a new array, `temp[]`, that is used to compute the new values of the interior elements of `b[]`. The *overlay* operator is used to merge in temp[] values into `b[]`.

Finally, the statement labeled (4) performs a sum reduction on `b[]`, and compares the sum value with an expected result.

5 Conclusion

`X10` is considerably higher-level than thread-based languages in that it supports dynamically spawning very lightweight activities, the use of atomic operations for mutual exclusion, and the use of clocks for repeated quiescence detection of a data-dependent set of activities. Yet it is much more concrete than languages like HPF in making explicit the distribution of data objects across places. In this, the language reflects the designers' belief that issues of locality and distribution cannot be hidden from the programmer of high-performance code in high-end computing. A performance model that distinguishes between computation and communication must be made explicit and transparent. At the same time we believe that the interaction between the concurrency constructs and the place-based type system (including first-class support for type parameters) will enable much of the burden of generating distribution-specific code and coordination of activities to be moved from the programmer to the underlying implementation.

In future work we plan to extend `X10` along two dimensions. First we plan to develop an *implicit syntax* which allows the programmer to elide certain details. The compiler will automatically fill-in these details based on type information. For instance, the programmer may specify an assignment `l = x` where `x` is not known to be local; the compiler may automatically introduce a `force/future` combination to read the remote value synchronously and store it in `l`. Several simplifications to the `X10` syntax are possible in this fashion.

Second we plan to develop mechanisms to support library developers writing place- and clock-generic code and their own high-level domain-specific abstractions. For instance, it should be possible for library developers to write code for hierarchically tiled arrays [1], and for distributed data-structures [5]. It should be possible for such developers to use the equivalent of `foreach/ateach` over their own distributed data-structures.

We plan to evaluate the effectiveness of the `X10` language by designing and running *productivity trials*. These trials will primarily be designed to evaluate the ease of developing new code in the HPC domain using `X10`. We plan to target developers in the HPC domain who are focused on developing performance-efficient library code, as well as developers interested in rapidly prototyping new applications (that must use high degrees of concurrency). Once a performance-efficient implementation of `X10` is available we also plan to evaluate the performance of `X10`, for a range of benchmark programs.

Acknowledgments

`X10` is being developed in the context of the IBM PERCS (Productive Easy-to-use Reliable Computing Systems) project, which is supported in part by DARPA under contract No. NBCH30390004. We are grateful to the following people for their feedback and contributions to the design and implementation of `X10`: David Bacon, Bob Blainey, Philippe Charles, Perry Cheng, Julian Dolby, Guang Gao, Christian Grothoff, Allan Kielstra, Robert O'Callahan, Filip Pizlo, Christoph von Praun, V.T. Rajan, Lawrence Rauchwerger, Mandana Vaziri, and Jan Vitek. We are also grateful to John McCalpin, Mootaz Elnozahy and Lisa Spainhower for helpful discussions.

References

[1] Gheorge Almasi and Luiz de Rose and Jose Moreira and David Padua. Programming for Locality and Parallelism with Hierarchically Tiled Arrays.

[2] C.A.R. Hoare. Monitors: An operating system structuring concept. *CACM*, 17(10):549–557, October 1974.

[3] R. Halstead. MULTILISP: A language for concurrent symbolic computation. *ACM Transactions on Programming Languages and Systems*, 7:501–538, 1985.

[4] Igor Pechtchanski and Vivek Sarkar. Immutability Specification and its Applications. *Proceedings of the ACM 2002 Java Grande/ISCOPE Conference*, October 2002.

[5] Steven Saunders and Lawrence Rauchwerger. Armi: an adaptive, platform independent communication library. pages, 230–241. ACM Press, 2003.

[6] David Bacon. Kava: A Java dialect with a uniform object model for lightweight classes. *Concurrency – Practice and Experience*, 15:185–206, 2003.

[7] V. Sarkar and G. R. Gao. Analyzable atomic sections: Integrating fine-grained synchronization and weak consistency models for scalable parallelism. Technical report, CAPSL Technical Memo 52, February 2004.

[8] Vivek Sarkar and Clay Williams and Kemal Ebcioğlu. Application development productivity challenges for high-end computing. In *Proceedings of Workshop on Productivity and Performance in High-End Computing (P-PHEC)*, February 2004. http://www.research.ibm.com/arl/pphec/pphec2004-proceedings.pdf.

[9] Bradford L. Chamberlain and Sung-Eun Choi and Steven J. Deitz and Lawrence Snyder. The high-level parallel language ZPL improves productivity and performance. In *Proceedings*

of the IEEE International Workshop on Productivity and Performance in High-End Computing, 2004.

[10] HPL Workshop on High Productivity Programming Models and Languages, May 2004. http://hplws.jpl.nasa.gov/.

Understanding Productivity through Non-intrusive Instrumentation and Statistical Learning

Nicholas A. Nystrom, John Urbanic, and Christina Savinell
Pittsburgh Supercomputing Center
{nystrom,urbanic,savinell}@psc.edu

Abstract

Advances in parallelism now yield systems of theoretical peak performance nearing 100 TFlop/s. However, realizing an acceptable fraction of that potential, either for ports of legacy applications or for development of new ones, is generally achieved only after "heroic" efforts. Designing systems to deliver sustained petascale performance will require understanding the elements of architecture, programming models and environments, languages, hardware, and software that enhance or hinder productivity. To arrive at that understanding, we have developed SUMS, the Standardized User Monitoring Suite, which applies techniques of data mining and statistical learning to objective, fine-grained observations of software development processes. Instrumentation is with consent but non-intrusive, improving the accuracy of sampled data. A modular, layered architecture allows for rapid, independent development and exploration of new ideas. Lightweight data acquisition components facilitate installation at new sites and applicability to arbitrary systems and experiments. The SUMS database aggregates data from multiple experiments, facilitating analysis both currently and retrospectively. We present the design and implementation of SUMS, together with results from two initial experiments.

1. Introduction

Computational science is now recognized as a vital complement to theory and experiment. Realistic simulations involving detailed models, high resolution, and sophisticated algorithms are necessary for advancement and to sustain competitiveness in physical and life sciences, engineering, climate and environment, and national security [1]. Today's high-end computing systems are largely evolutionary, providing thousands to tens of thousands of commodity processors coupled with various interconnects and programmed using a relatively small number of established programming models and languages. Productive use of those resources as capability systems is possible but not without challenge, and scaling complex software to run efficiently even on only thousands of processors requires specialized expertise and remains arduous. Demands for ever more realistic simulation and for progress in new, challenging problem domains continue to motivate development of more powerful capability-oriented and leadership-class computing facilities. At the high end, for example, applications requirements were recently identified for full global climate modeling and plasma fusion simulation that extend to the zettaflop (10^9 teraflops) range [2]. Systems now being envisaged [3] entail scales and revolutionary aspects that may require very different programming models, languages, and development environments if they are to produce desired results in acceptable timeframes. Improved understanding of the factors that enable or hinder productivity is required to guide the design of future systems.

The need to understand productivity has fostered several efforts to quantify software development in the context of parallel computing. Recently, the DARPA HPCS Productivity Team [4] initiated a series of productivity experiments and has begun collecting a set of hypotheses, known with the community as "tribal lore", for testing based on the results of those experiments. Instrumentation for the HPCS "development time" experiments conducted to date has used Hackystat [5], which provides a sensor-based framework for collecting data from software developers.

The current work describes SUMS, the Standardized User Monitoring Suite, which addresses the essential problem of understanding factors which promote or hinder productive use of high-end computing resources. SUMS differs from other initiatives in two fundamental ways. First, we believe that the software development process must be instrumented in an objective, comprehensive, non-intrusive manner that supports systems on which actual work must be accomplished using today's systems and programming models, languages, and techniques. Truly meaningful data will not be accessible through designed productivity studies alone. Instrumentation must also be easily extensible to new systems, languages, and models as the field rapidly evolves. Second, we believe that data mining [6], in which we include statistical learning and knowledge discovery, is a powerful and natural approach to understanding complex relationships inherent in data that will necessarily span many experiments conducted with disparate subject populations and environments over the span of years. This mandates objective, well-defined, fine-grained data acquisition.

SUMS acquires data from programming activities in very general contexts. Those contexts can include designed productivity experiments, but they can also include classroom and workshop settings and active research groups. That generality allows SUMS to address populations ranging from novices to experts working on software ranging from easily implemented algorithms through complex research codes. Of special interest is the software development process for actual research codes, which maps poorly to short-duration, designed experiments. Lightweight, easily deployed, non-intrusive acquisition that work with diverse systems, programming models, and languages encourage widespread use of SUMS by avoiding barriers to acceptance. Data acquisition within SUMS supports paradigms needed by today's practitioners, such as C, Fortran, C++, MPI, OpenMP, and CHARM++ [7], as well as those of emerging interest, such as UPC [8], Co-Array Fortran [9], Titanium [10], TCE [11], and X10. The data gathered is objective, fine-grained, and highly detailed. The SUMS database provides a powerful and efficient core, efficiently bridging comprehensive development data and high-level analysis tools.

SUMS addresses the high dimensionality of the feature set—both systems and experiments can represent an abundance of possibilities—in the analysis and discovery layer through techniques of data mining and statistical learning. For example, in an experiment using UPC, cluster analysis of source code changes and compiler output as related to subject population might reveal one subgroup adept in C syntax and another that is less adept (perhaps coming from a Fortran background). Recognizing those clusters would guide separate analyses and generate potentially different inferences, each with higher confidence, for each of the subpopulations.

Central to the SUMS approach are techniques of unsupervised and supervised learning. Rather than test a finite set of hypotheses, which may be incomplete, insufficiently general, or evolving less rapidly than the systems, SUMS instead non-obtrusively (and with consent) monitors developers' progress. Productivity analysts can mine the data immediately as well as retrospectively, as additional experimental data add statistical weight and as new questions arise.

2. Architecture and Implementation

The SUMS architecture (Figure 2) features distinct layers of components: acquisition components, deployed at productivity experiments to acquire raw data; analysis and discovery components, which implement techniques of data mining and machine learning to cluster, recognize patterns in, and draw inferences from the SUMS data; and presentation components, which provide a user interface through which performance analysts explore and interact with the data. Coupling the architecture, the SUMS database efficiently and securely bridges data acquisition and analysis layers.

2.1. Design goals

Practical design goals shape the SUMS system. SUMS must address the full range of system architecture, programming models, languages, software, and hardware of interest both today and in the future. It must obtain robust data from practitioners as they work, not in contrived settings, and therefore must be portable, general, objective, and non-intrusive. Through appropriate choices for generality and easy extensibility, SUMS supports systems relevant to current productivity experiments, and it will also support systems that are not yet ready for those deployments, e.g. emerging programming languages. We seek to complement and interoperate with the DARPA HPCS Productivity Team. To that end, we are integrating the Productivity Team's metrics, workflows, and benchmarks [12] into analysis and discovery components. To address the full range of productivity concerns, SUMS aggregates empirical data over large and diverse populations and development approaches, which will achieve statistical significance as the volume of data acquired from productivity experiments continues to expand. Choosing a componentized, extensible framework promotes rapid implementation for short-term analysis as well as future discovery, encouraging incremental additions and increases in sophistication. Extensibility is necessary to provide ongoing value and broad applicability; indeed, the methodology embodied in SUMS is not specific to high-end computing, but is equally applicable to software engineering in general. SUMS must interoperate with diverse development environments, both open and proprietary, ranging from command line interfaces (e.g. *f90, make)* to integrated development environments (e.g. Eclipse). To avoid biasing incoming data, SUMS must integrate transparently with participants' workflows, not perturb their work patterns, and not depend on subjective input. To foster widespread deployment, SUMS Data Acquisition Components feature support for relevant systems and convenient installation and uninstallation. Finally, we leverage third-party software, especially in the analysis and discovery layer.

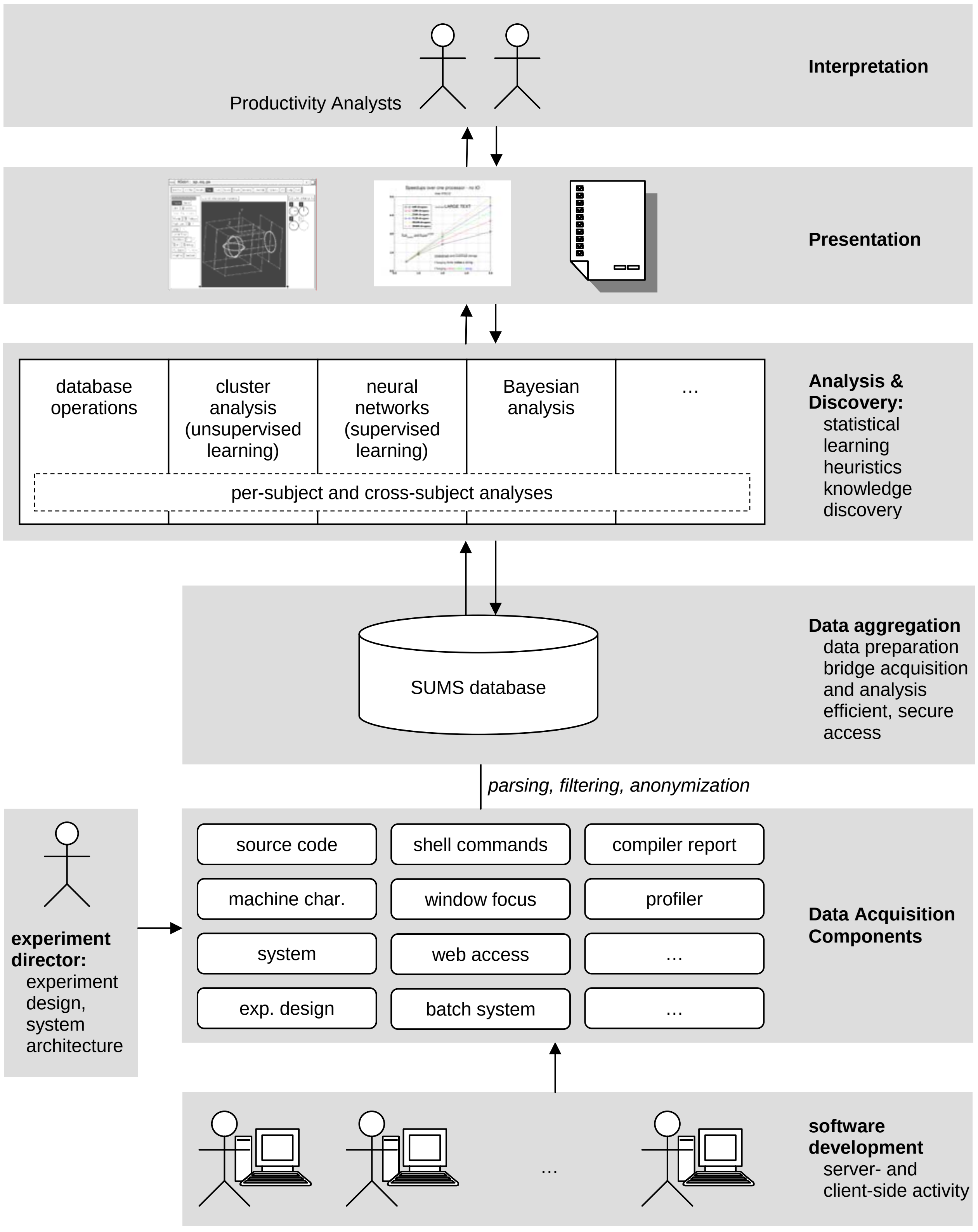

Productivity Analysts
Interpretation
Presentation
database operations
cluster analysis (unsupervised learning)
neural networks (supervised learning)
Bayesian analysis
…
per-subject and cross-subject analyses
Analysis & Discovery: statistical learning heuristics knowledge discovery
SUMS database
Data aggregation data preparation bridge acquisition and analysis efficient, secure access
parsing, filtering, anonymization
experiment director: experiment design, system architecture
source code
shell commands
compiler report
machine char.
window focus
profiler
system
web access
…
exp. design
batch system
…
Data Acquisition Components
…
software development server- and client-side activity

Figure 1. SUMS architecture.

2.2. Data Acquisition Components

SUMS acquisition components are lightweight, independent but interoperating tools that instrument actions involved in software development. Acquisition components are designed to be portable where possible, with system-specific versions where necessary. Client-side support includes Linux and Window, with possible extension to Mac OS X, pending demand. Server-side support currently includes Linux- and UNIX-like systems. In this model, developers work on a client (for example, a PC), remotely developing parallel code for execution on a supercomputer or a cluster.

Finally, and perhaps most importantly, only through non-intrusively capturing practitioners' actual development process can we hope to glean, without introducing bias, the factors that influence their productivity in real-world settings. For example, we choose not to interrupt a subject with an inquiry of "Why did you recompile your code?", obtaining an answer that may or may not be accurate. Instead, by assembling data on recent shell commands, source code revisions, and compiler invocations and results, one can instead deduce that the reason for recompilation, whether to debug, to tune performance, or as a step in initial code development.

Acquisition component implementations include program wrappers, cron scripts, small C utilities, and easily deployed third-party software. SUMS acquisition components currently include:

sums_source records statistics describing source files, including total lines of code and the number of lines added, deleted, and changed for each file. The sums_source component executes as a cron process, examining saved files every 60 seconds (tunable), allowing use of arbitrary editors and additional a posteriori analysis. For example, language-specific analysis tools are being developed to instrument measures of complexity, e.g. the number of entry points and variables, and also indicators of explicit parallelization, e.g. MPI calls, OpenMP directives, and parallel language constructs.

sums_web obtains timestamps, usernames, URLs, and access methods from Squid [13] web proxy cache logs. Output from the *sums_web* component contains data necessary for identifying accesses to documentation. *Sums_web* data is also useful for identifying non-development time, which can be deduced from the URLs visited and corroborated by output, or lack thereof, from other acquisition components. Prior to database insertion, *sums_web* output is pruned to eliminate extraneous information, e.g. retrieval of images associated with other html or text requests.

sums_shell records each command executed from a shell. When integrated with IDEs, analogous functionality can be provided for spawned subprocesses.

sums_window notes the name and auxiliary information for the subject's active window, providing information on subjects' instantaneous activity in a windowed desktop environment. The *sums_window* component is currently implemented for both Windows and X11.

sums_compiler records each compiler invocation along with its command line, exit status, number of errors, and number of warnings, and output.

sums_batch records output from batch software, through which supercomputers are typically accessed. Output from the *sums_batch* component includes the number of processors, wall-clock execution time, and when available, other job-specific parameters such as memory requirements.

sums_profiler records invocations of code profilers and their results.

Additional acquisition components are planned.

2.3. SUMS Database

The SUMS database aggregates instrumentation and system data associated with each experiment, providing an efficient engine to manage multiple views, augment raw data with derived information and dimension-reduced forms, and interface to analysis and discovery components. The design of the database allows for analyses and re-interpretation through complementary "component" and "history" views.

The component view preserves all raw data from each data acquisition component, enabling component-specific interpretation of highly detailed statistics. Preservation of raw data allows future analyses of increased sophistication and targeting new lines of inquiry.

Supplementary fields in the component view, as well as in auxiliary tables, support data preparation including normalization, conversion to alternate representations, preliminary parsing, dimension reduction, and designation of outliers. For example, categorizing window focus records into shell, browser, debugger, performance tool, IDE, and system classes allows recognition of activity patterns without extending definitions of each instance of each class to higher-level tools. The secondary data structures accommodate information generated by synthetic components, which fuse data from separate acquisition components to add meaning. They also allow for feedback from data mining and statistical tools, providing scaffolding for additional analyses and method validation.

The history view facilitates tracking all events for any given user on a timeline basis. Operating on the history view, queries and other analysis tools can immediately reconstruct an ordered timeline of any subject's development activities. The history view contains summaries of the raw data, to increase the efficiency of complex analyses, as well as links to detailed raw data, to allow arbitrarily deep inquiries. Because all raw data is preserved, the history can be regenerated at will as improved summarization procedures are implemented. As in the component view, the history view supports derived data, e.g. assignment of phases Development, Debugging, Performance Optimization, Testing, and Production to a given subject's timeline.

The SUMS database is implemented in MySQL [14], in which to subjects' data is controlled through a username/ password/host model. A web interface will be provided for uploading SUMS data collected from new experiments.

2.4. Analysis and Discovery

Analysis and Discovery components operate on data from the SUMS database, augmenting the database, drawing inferences, and providing tools for unsupervised and supervised machine learning to identify correlations between independent variables with various measures of productivity and workflows. Following the SUMS philosophy, the Analysis and Discovery layer includes independent, interoperating tools for data mining, cluster analysis, pattern recognition, neural networks, inference generation, and other statistical methods. Leveraging selected, mature, open-source software, our focus is on efficiently applying a range of techniques to detailed productivity measurements, discovering salient and perhaps subtle trends in the empirical data. Inferences generated will guide future experiments and, as the weight of data collected achieves statistical significance, we hope they will also guide development of environments and systems to increase productivity.

Examples of the types of inquiries we expect to be able to probe inquiries include questions such as the following: What clusters of total development times and application performance emerge for given applications being developed in MPI vs. OpenMP vs. Co-Array Fortran vs. UPC vs. X10 vs. hybrid models? For applications that yield favorable scaling, where are developers spending most of their time, and hence, where would improvements yield the greatest benefit? What effect, quantitatively, do IDEs (integrated development environments, for example, Eclipse) have on productivity? How do correlations differ for porting large, legacy applications vs. developing small, specialized, codes? How does choice of programming model and language affect ongoing maintenance and enhancement of production applications? What effect do various factors, for example programming model and interconnect design, have on inception-to-result productivity? If an architectural feature were changed in a specified way, what would be the effect on productivity?

Attaining those goals will require gathering of adequate productivity data, together with careful mining of the collection. The data must be sufficiently rich to support cross-validation through multiple methods, validating against measurable quantities wherever possible.

A few aspects of analysis can be implemented purely as database operations. For example, statistics regarding development times and execution times as related to different programming models or different architectures can be implemented directly in SQL. Similarly, many data preparation tasks are also best expressed in the database layer, and a library of useful utilities is steadily growing.

We have begun preliminary analyses using a variety of data mining techniques, to be detailed in a future paper. Tools include a suite of database scripts to filter, combine, and classify raw component data, as well as sophisticated and relatively mature systems for information visualization [15] and machine learning [16]. Initial emphasis is on cluster analysis, which together with assignment of development phases to timelines, will significantly improve the confidence of inferences drawn from otherwise disparate subject populations and systems. Those analyses, together with knowledge of experimental conditions, fit naturally with HPCS workflows [12] identified by Koester, et al. Similarly, HPCS productivity metrics can be integrated into the SUMS analysis tools. Explicitly addressing the HPCS productivity metrics and workflows will provide a common language through which the community can interact and advance.

Supervised learning techniques, in particular neural networks and genetic algorithms, are of interest for their potential to relate perturbations to known systems to changes in productivity. To explore this possibility, we plan to train selected algorithms using subsets of the SUMS data. Their results, expressed for example in the parlance of productivity metrics, will be validated against the remainder and also against other techniques. As new data continue to arrive, we will revisit the collection as a whole, periodically redrawing inferences, increasing scope and sophistication of analyses and knowledge discovery tools, and improving the confidence of predictions.

3. Productivity Experiments

SUMS was deployed in two productivity experiments to date, summarized in Table 1. Additional productivity studies are planned, addressing a range of programming models and subject backgrounds.

3.1. PSC Workshop (MPI)

SUMS was first deployed at the Pittsburgh Supercomputing Center's *Introduction to Terascale Code Develop-*

ment workshop on September 13-14, 2004. Acquisition components were deployed on both terminal and supercomputer systems. Terminals were dual-boot Windows/ Linux PCs in the PSC's training center. Individual logins ensured accurate reporting even though participants could use different systems as the workshop progressed. Exercises were conducted on the NSF Terascale Computing System (TCS), a 3000-processor HP AlphaServer SC (Compaq Tru64 UNIX V5.1A (Rev. 1885); Compaq AlphaServer SC TS2.5; dual-rail fat tree Quadrics Elan3 interconnect; 1.0GHz Alpha EV6.8 processors with 4GB RAM per 4-processor node; PSC-modified OpenPBS scheduler).

The two-day workshop focused on MPI using C and Fortran. Participants developed a parallel solution to Laplace's equation using point Jacobi iteration, chosen because its conceptual simplicity allows individuals from a broad range of backgrounds to focus on the parallel programming principles rather than the algorithm. The *sums_web*, *sums_shell*, *sums_compiler*, *sums_window*, and *sums_source* acquisition components were installed and executed, and subjects progressed at normal rates through their exercises, as judged by the instructors who taught the same exercises on numerous prior occasions.

Accumulation of data from similarly opportunistic productivity studies, working within constraints imposed by experimental environments, will result in increasingly significant masses of data. Furthermore, that is the only model available for the intensely interesting target group of expert developers working on research applications.

3.2. University of Pittsburgh class (MPI, OpenMP)

A second experiment was conducted on November 29 and December 2, 2004, drawing on undergraduates in a *Parallel Computing* course at the University of Pittsburgh. Goals for this experiment included deployment of data acquisition components at an external site, development and use of an experimental design conducive to comparing programming models, and gathering of productivity data specific to OpenMP and MPI across a relatively uniform population.

Iterative solution of a tridiagonal system using Jacobi's algorithm was selected as being well-suited to the short time (80 minutes per session) available for the classes. (We also developed a closely related example using successive over-relaxation that would be excellent for more protracted experiments, as the dependence within the inner loop poses a greater challenge for parallel implementations.) The algorithm was described verbally, and then a handout was distributed that included pseudocode, input specification, and an example of correct output. Pseudocode was chosen to minimize bias not related to message-passing or threaded programming models. The input was designed to scale from small rank (e.g. 8), for development and debugging, to large rank (e.g. 400) to measure parallel efficiency, which is one component of productivity.

The class was conducted in two sessions. In session 1, five of the ten subjects attempted the exercise in MPI,

Table 1. Summary of productivity experiments 1 and 2, conducted from September-December, 2004.

experiment	subjects	programming model(s)	programming exercise	milestones
1. Workshop: *Introduction to Terascale Code Development*	4 undergraduates 7 graduate students 2 postdocs 2 faculty 4 research staff 19 total	MPI; C & Fortran	Solution of Laplace's equation assigned from serial reference codes C: 72 SLOC; Fortran: 82 SLOC	SUMS proof-of-concept Successful deployment of data acquisition components Successful instrumentation of development activity: 19,717 actions recorded
2. Class: *Parallel Programming*	10 undergraduates	MPI, OpenMP; C	Iterative Jacobi solution of a tridiagonal system assigned as pseudocode serial reference implementation: 45 SLOC	Successful deployment at an external institution Designed experiment: 5 subjects performed the exercise with OpenMP while the other 5 used MPI. The groups then switched to the other programming model, addressing discrimination of learning effects.

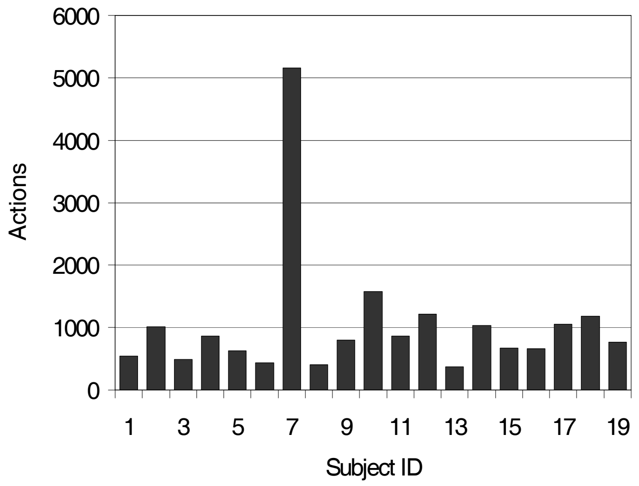


Figure 2. Total actions logged for each of the subjects who participated in Experiment 1. The obvious outlier, actions for subject_id=7, is dominated by web accesses. Even this apparently troublesome case can be effectively mined, however, as URLs associated with web access provide indications of development time (for example, accessing documentation or course materials) versus non-development time (reading webmail).

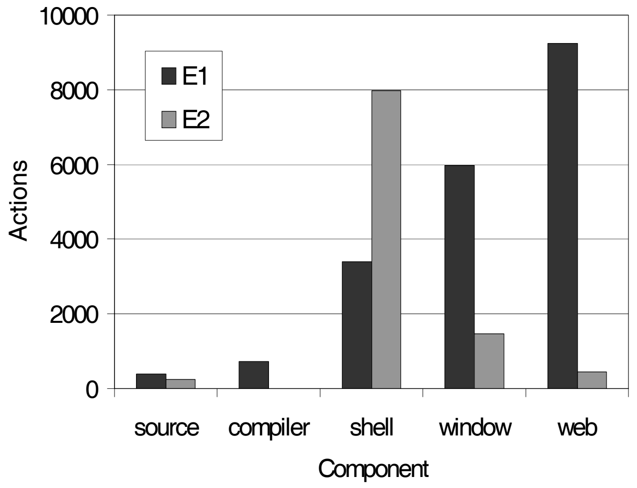


Figure 3. Actions logged per component for Experiments 1 and 2. The differing acquisition rates from E1 to E2 reflect a combination of factors including experiment format, subjects' background, motivation, and priorities, and systems, programming models, languages, and algorithms being considered. "Source", "compiler", "shell", "window", and "web" denote the components from which the corresponding actions were obtained.

while the other half attempted it in OpenMP. All subjects programmed in C, consistent with prior class policy, and all had completed two prior programs in both OpenMP and MPI. The choice of initial programming model was voluntary.

Only one subject completed the exercise in the time available. Allowing subjects to continue their first day's activity in session 2, seven completed the OpenMP version, five completed the MPI version, and only one completed both. This underscores the importance of selecting straightforward exercises for short experiments and relatively inexperienced programmers, as well as the importance of complementing classroom settings with more less constrained productivity studies.

3.3. Preliminary Analysis

Experiments 1 and 2 established confidence that the SUMS methodology effectively gathers objective, fine-grained information describing the software development process. Subjects progressed at usual rates, unencumbered by the data acquisition tools that quietly logged their actions. Voluntary participation rates were high: 83% in Experiment 1, and 100% in Experiment 2. The extremely preliminary and limited nature of our data precludes actual analysis at this stage. Instead, we focus on the nature of the data collected for experiment 1, which will be the critical foundation for later work.

19,717 actions were logged by acquisition components during experiment 1. Figures 2 and 3 summarize logged actions by subject and by component. The distribution by subject showed a wide range of activity, ranging from 4.4 to 676 actions per hour. Data acquisition rates for individual components vary widely, with source code changes and compilations being among the less frequent. The components' patterns show indications of dependence on a variety of as-yet-undetermined factors.

Shifting to the history view, Figure 4 provides a view of subject 18's activity. The plot is a timeline, where the vertical axis jitter serves only to separate the 505 actions represented. We see a distinct period of development from approximately 10-40 minutes, most simply indicated by frequent source code changes and recompilations. From the next 35 minutes, intense activity from the *sums_window* component reveal that the subject was debugging with TotalView, frequently re-running the exercise binary, visualizing its content, and working through the source. This is evident through application name and class data logged by *sums_window*, with values in this case of app_name=XWinClass and app_class={Etnus, Program,Process,visualize,"prun<laplace_mpi_c>.0"}. Subject 18 then recompiled laplace_mpi.c, requiring 5

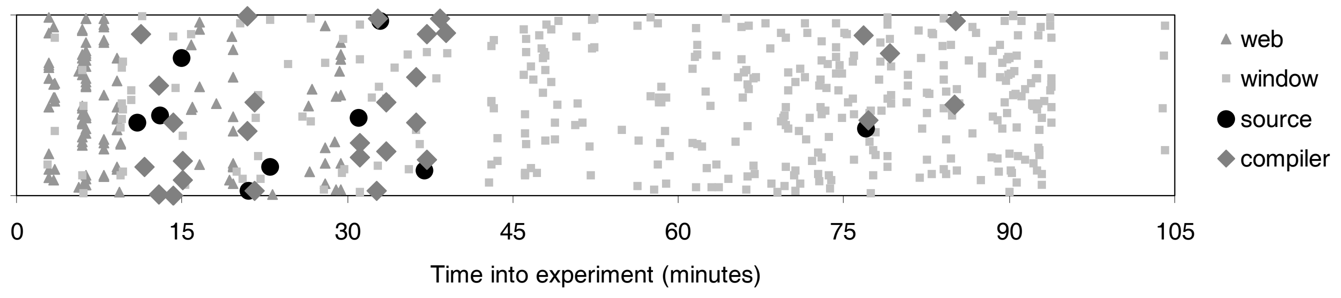


Figure 4. History for exp_id=1, user_id=18, showing only source code updates, compilations, web accesses, and window focus actions. Development and testing phases are clearly visible. The latter is identifiable with a high degree of confidence by the single source change, consisting of reducing the number of iterations from 1000 to 200, followed by 5 compilations with debugging enabled and successively closer approximations to the correct combination of libraries. The final link line was correct. For each data point, additional data on URL (web), compiler and command line (compiler), file changes (source), and window name and class (window) are available, from which detailed analyses of effort and relationships can be performed.

attempts to get the link line right, executed the result, and returned to TotalView. Incorporation of additional features into the analysis, such as the nature of individual source code changes and summarizing time spent in each development phase for all subjects, is underway.

4. Conclusion and Future Work

We implemented the SUMS Data Acquisition Components, a lightweight, componentized system of interacting but independent tools for non-intrusively observing software development processes. Two initial productivity experiments provided seed data, which we are now using to establish effective data mining techniques. Additional experiments will increase SUMS data to statistical significance. We are actively seeking new participants to improve sampling. Experimental systems, programming models and languages, and problem domains will be expanded as new architectures, compilers, and development environments become available.

Applications of both unsupervised and supervised learning algorithms will be applied to the existing data, and as data from additional productivity experiments arrives, the techniques developed will be reapplied for validation and to attain statistical relevance. This modeling will incorporate HPCS productivity measures and workflows, providing a valuable common ground for ongoing dialog with other productivity efforts. As the volume of empirical data expands, mining of SUMS data will allow detailed assessment of factors underlying productivity. Using neural nets and other machine learning techniques trained with results of sufficient productivity studies, we will also explore the feasibility of generating predictions regarding the effects that perturbations to existing architectures and systems would have on their productivity.

The relatively sparse opportunities during which data are collected, together with the complexity of the relationships that must be understood, warrant investigating fully the issues faced by individual software developers prior to introducing additional complexity. For example, lessons learned from the case of individuals may inform extension of SUMS to more general cases of teams of programmers and distributed, long-term development.

5. Acknowledgments

SUMS is supported by IBM through PERCS [Grant number W01229580], in collaboration with Rami Melhem and Raymond Hoare (University of Pittsburgh). We would like to thank Ram Rajamony and Mootaz Elnozahy (IBM Research) for insightful discussions during the conception of SUMS. Nicolas Baldi, Kevin Kibler, James Larkby-Lahet, and Mark Smorul implemented the initial data acquisition components used in Experiment 1.

6. References

[1] Federal Plan for High-End Computing: Report of the High-End Computing Revitalization Task Force (HECRTF), May 10, 2004. http://www.itrd.gov/hecrtf-outreach/.

[2] E. P. DeBenedictis. Will Moore’s Law Be Sufficient. In Proceedings of the 2004 ACM/IEEE conference on Supercomputing, Pittsburgh, Pennsylvania, 1-12.

[3] DARPA High Productivity Computing Systems program. http://www.darpa.mil/ipto/programs/hpcs/

[4] J. Kepner, D. Koester, and B. Lucas. HPCS Productivity Team Phase 2 Kickoff. Available at www.highproductivity.org.

[5] S. Faulk, P. M. Johnson, J. Gustafson, A. A. Porter, W. Tichy, and L. Votta, Measuring HPC Productivity, International Journal of High Performance Computing Applications, December, 2004.

[6] See, for example, T. Hastie, R. Tibsshirani, J. Friedman. The Elements of Statistical Learning: Data Mining, Inference, and Prediction. Springer-Verlag, New York, 2001 and M. Kantardzic. Data Mining. John Wiley & Sons, Inc., Hoboken, New Jersey, 2003.

[7] L.V. Kale and S. Krishnan, CHARM++ : A Portable Concurrent Object Oriented System Based On C++. In Proceedings of the Conference on Object Oriented Programming Systems, Languages and Applications, Sept-Oct 1993. ACM Sigplan Notes, Vol. 28, No. 10, pp. 91-108.

[8] W. W. Carlson, J. M. Draper, D. E. Culler, K. Yelick, E. Brooks, and K. Warren, Introduction to UPC and language specification, Technical Report CCS-TR-99-157, Center for Computing Sciences, Bowie, MD, May 1999.

[9] R. W. Numrich and J. K. Reid, Co-Array Fortran for parallel programming, Technical Report RAL-TR-1998-060, Rutherford Appleton Laboratory, Oxon, UK, August 1998.

[10] K. Yelick, L. Semenzato, G. Pike, C. Miyamoto, B. Liblit, A. Krishnamurthy, P. Hilfinger, S. Graham, D. Gay, P. Colella, and A. Aiken, Titanium: A high-performance Java dialect, in ACM 1998 Workshop on Java for High-Performance Network Computing, 1998.

[11] G. Baumgartner, D. E. Bernholdt, D. Cociorva, R. Harrison, S. Hirata, C.-C. Lam, M. Nooijen, R. Pitzer, J. Ramanujam, and P. Sadayappan. A High-Level Approach to Synthesis of High-Performance Codes for Quantum Chemistry. In Proceedings of the 2002 ACM/IEEE conference on Supercomputing, Baltimore, Maryland, 1-10.

[12] D. Koester. HPCS Applications Analysis and Assessment: Phase 1 Summary. In 2003 HPC User Forum Presentations, Sundance, Utah, 2003. http://www.idc.com/hpc.

[13] H. Nordstrom, Squid Web Proxy Cache, www.squid-cache.org; O. Pearson, Squid: A User's Guide, http://squid-docs.sourceforge.net/latest/book-full.html.

[14] P. DuBois, MySQL, Second Edition, Sams Publishing, Indianapolis, Indiana, 2003.

[15] D. F. Swayne, D. Cook, A. Buja, H. Hofmann, and D. T. Lang, Interactive and Dynamic Graphics for Data Analysis: With Examples Using R and GGobi, http://www.public.iastate.edu/~dicook/ggobi-book/ggobi.html.

[16] I. Witten and E. Frank. Data Mining: Practical Machine Learning Tools and Techniques with Java Implementations. Morgan Kaufmann Publishers, San Francisco, California, 2000.

Understanding HPC Development through Automated Process and Product Measurement with Hackystat

Philip M. Johnson
Michael G. Paulding
Collaborative Software Development Laboratory
Department of Information and Computer Sciences
University of Hawai'i
Honolulu, HI 96822
johnson@hawaii.edu
mpauldin@hawaii.edu

Abstract

The high performance computing (HPC) community is increasingly aware that traditional low-level, execution-time measures for assessing high-end computers, such as flops/second, are not adequate for understanding the actual productivity of such systems. In response, researchers and practitioners are exploring new measures and assessment procedures that take a more wholistic approach to high performance productivity. In this paper, we present an approach to understanding and assessing development-time aspects of HPC productivity. It involves the use of Hackystat for automatic, non-intrusive collection and analysis of six measures: Active Time, Most Active File, Command Line Invocations, Parallel and Serial Lines of Code, Milestone Test Success, and Performance. We illustrate the use and interpretation of these measures through a case study of small-scale HPC software development. Our results show that these measures provide useful insight into development-time productivity issues, and suggest promising additions to and enhancements of the existing measures.

1. Introduction

High performance computing systems are becoming mainstream due to decreasing costs and increasing numbers of application areas with computation and/or data intensive processing. With this interest, however, comes new challenges. For example, recent initiatives in the HPC community [8, 1] have concluded that low-level HPC benchmarks of processor speed and memory access times no longer necessarily translate into high-level increases in actual development productivity. Put another way, the bottleneck in high performance computing systems is increasingly due to software engineering, not hardware engineering.

To make matters even more interesting, high performance computing application development often differs in significant ways from the systems and development processes traditionally addressed by the software engineering community:

- The requirements often include conformance to sophisticated mathematical models. Indeed, requirements may often take the form of an executable model in a system such as Mathematica, and the implementation involves porting to the HPC system.
- The software development process, or "workflow" for HPC application development may differ profoundly from traditional software engineering processes. For example, one scientific computing workflow, dubbed the "lone researcher", involves a single scientist developing a system to test a hypothesis. Once the system runs correctly once and returns its results, the scientist has no further need of the system. This contrasts with standard software engineering lifecycle models, in which the useful life of the software is expected to begin, not end, after the first correct execution.
- "Usability" in the context of HPC application development may revolve around optimization to the machine architecture so that computations complete in a reasonable amount of time. The effort and resources involved in such optimization may exceed initial development of the algorithm.

Fortunately, there is an emerging interdisciplinary community involving both HPC and software engineering researchers and practitioners who are attempting to define new ways of measuring high performance computing systems, ways which take into account not only the low-level hardware components, but also the higher-level productivity costs associated with producing usable HPC applications.

This paper presents an approach to investigating the software engineering problems associated with high performance computing system application development. It involves the introduction of technology into the HPC development environment which unobtrusively gathers process and product data. This process and product data can be used for two purposes. First, it can be used to provide a more wholistic perspective on productivity, one that includes measures of performance, functionality, and development. Second, it can be used to provide new insight into the process of high performance system application development, which can be used to identify bottlenecks in the development process and assess the consequences of process or product changes on these bottlenecks. We have been applying this approach to an ongoing case study of high performance computing system application development in our laboratory, and this paper reports on our initial results.

The remainder of the paper is organized as follows. Section 2 introduces the technology we have developed, called Hackystat, which supports unobtrusive collection and analysis of product and process measures. Section 3 introduces “Software Project Telemetry“, which is the principal approach to measurement collection and interpretation we have adopted for this research. Section 4 introduces a case study adapted from the Truss Purpose-based Benchmark (PBB) [3], which uses the problem specification but collects and analyzes an alternative set of metrics. Section 5 presents our initial conclusions from the use of these metrics and our future directions.

2. Automated process and product measurement with Hackystat

An important characteristic of our approach to understanding HPC software development and productivity is that measures of product and process must be automatically collected. This requirement limits the kinds of data we can collect, but dramatically lowers the cost of collecting these measures and provides a level of scalability for measurement not possible with expensive, manual data collection.

For the past several years, we have been developing a framework for automated software development process and product metric collection and analysis called Hackystat. This framework differs from other approaches to automated support for product and process measurement in one or more of the following ways:

- Hackystat uses sensors to unobtrusively collect data from development environment tools; there is no chronic overhead on developers to collect product and process data.

- Hackystat is tool, environment, process, and application agnostic. The architecture does not suppose a specific operating system platform, a specific integrated development environment, a specific software process, or specific application area. A Hackystat system is configured from a set of modules that determine what tools are supported, what data is collected, and what analyses are run on this data.

- Hackystat is intended to provide in-process project management support. Many traditional software metrics approaches are based upon the “project repository” method, in which data from prior completed projects are used to make predictions about or support control of a current project. In contrast, Hackystat is designed to collect data from a current, ongoing project, and use that data as feedback into the current project.

- Hackystat provides infrastructure for empirical experimentation. For those wishing to compare alternative approaches to development, or for those wishing to do longitudinal studies over time, Hackystat can provide a low-cost approach to gathering certain forms of project data.

- Hackystat is open source and is available to the academic and commercial software development community for no charge.

The design of Hackystat [6] has resulted from of prior research in our lab on software measurement, beginning with research into data quality problems with the PSP [5] and which continued with the LEAP system for lightweight, empirical, anti-measurement dysfunction, and portable software measurement [7].

To use Hackystat, the project development environment is instrumented by installing Hackystat sensors, which developers attach to the various tools such as their editor, build system, configuration management system, and so forth. Once installed, the Hackystat sensors unobtrusively monitor development activities and send process and product data to a centralized web service. If a user is working offline, sensor data is written to a local log file to be sent when connectivity can be established with the centralized web service. Project members can then log in to the web server to see the collected raw data and run analyses that integrate and abstract the raw sensor data streams into telemetry. Hackystat also allows project members to configure “alerts“ that watch for specific conditions in the sensor data stream and send email when these conditions occur.

Hackystat is an open source project with sources, binaries, and documentation available at http://www.hackystat.org. There is also a public server available at http://hackystat.ics.hawaii.edu. Hackystat has been under development for approximately three years, and currently consists of around 900 classes and 60,000 lines of code. Sensors are available for a variety of tools including Eclipse, Emacs, JBuilder, Jupiter, Jira, Visual Studio, Ant, JUnit, JBlanket, CCCC, DependencyFinder, Harvest, LOCC, Office, and CVS.

3. Software Project Telemetry

A major application of Hackystat has been the development of a new approach to software measurement analysis called "Software Project Telemetry". We define Software Project Telemetry as a style of software engineering process and product collection and analysis which satisfies the following properties:

Software project telemetry data is collected automatically by tools that unobtrusively monitor some form of state in the project development environment. In other words, the software developers are working in a "remote or inaccessable location" from the perspective of metrics collection activities. This contrasts with software metrics data that requires human intervention or developer effort to collect, such as PSP/TSP metrics [4].

Software project telemetry data consists of a stream of time-stamped events, where the time-stamp is significant for analysis. Software project telemetry data is thus focused on evolutionary processes in development. This contrasts, for example, with Cocomo [2], where the time at which the calibration data was collected about the project is not significant.

Software project telemetry data is continuously and immediately available to both developers and managers. Telemetry data is not hidden away in some obscure database guarded by the software quality improvement group. It is easily visible to all members of the project for interpretation.

Software project telemetry exhibits graceful degradation. While complete telemetry data provides the best support for project management, the analyses should not be brittle: they should still provide value even if sensor data occasionally "drops out" during the project. Telemetry collection and analysis should provide decision-making value even if these activities start midway through a project.

Software project telemetry is used for in-process monitoring, control, and short-term prediction. Telemetry analyses provide representations of current project state and how it is changing at the time scales of days, weeks, or months. The simultaneous display of multiple project state values and how they change over the same time periods allow opportunistic analyses—the emergent knowledge that one state variable appears to co-vary with another in the context of the current project.

Software Project Telemetry enables a more incremental, distributed, visible, and experiential approach to project decision-making. It also creates perspectives on system development that can provide new insight into HPC development processes, as we illustrate in the case study below.

4. Process and Product measures for HPC utilizing Hackystat

The development of an HPC system from a software engineering perspective raises many interesting questions. How long does such a system take to develop? Do some components take longer to develop than others? How much of the system is devoted to the sequential code, and how much is devoted to the parallelization of this code? How did the developer allocate their time during development to these activities? Do different choices of HPC tools and technologies lead to different answers to these questions? Would a different application area lead to similar or different results?

We believe that automated infrastructure for the collection and analysis of product and process data is an important first step toward enabling the HPC community to generate answers to these questions, and then use these answers to improve the tools and techniques for HPC development. The question is, what process and product measures can be both automatically collected and used to provide interesting insight into the questions raise above? This case study investigates the use of the following measures: Active Time, Most Active File, Command Line Invocations, Parallel and Serial Lines of Code, Milestone Test Success, and Performance.

The following sections describe each of these measures and illustrate them with sample data from a one week "snapshot" of development of the Optimal Truss Design problem in our case study.

4.1. The Optimal Truss Design problem

Our case study focuses on the development of a system for optimal truss design. Specifically, the system finds a pin-connected steel truss structure that uses as little mass as possible to support a load connected from three attachment points on a wall to the load-bearing point away from the wall. This problem was originally developed for use in research on Purpose-Based Benchmarks (PBBs) [3]. PBBs gather a different and complementary set of metrics in order to assess productivity in terms of acceptability to the customer.

The system is being implemented by one of the authors, Michael Paulding, and thus generally conforms to the "lone researcher" workflow for HPC development. The system development process and associated case study started in the Spring of 2004 and is still ongoing. To date, the implementation of the Optimal Truss Design problem consists of approximately 1,200 source lines of code.

The solution to the Optimal Truss Design problem developed in this case study involves several components. The first component is termed the "sequential workhorse", which includes the task of solving a truss once all of its elements are defined. Solving a truss includes the calculation of its mass, which is determined by summing the mass of each of its components (e.g. all steel joints and members). In addition to mass calculation, solving a truss also includes verifying equilibrium and deformational constraints. Equilibrium constraints require that all forces and moments within a truss net zero magnitude, thus ensuring that the truss is not accelerating. Deformational constraints require that the length of members (strut or cable) used in the truss do not exceed construction safety limits. These limits are defined and known prior to runtime.

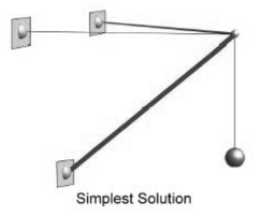


Figure 1. An unoptimized solution to the Truss problem

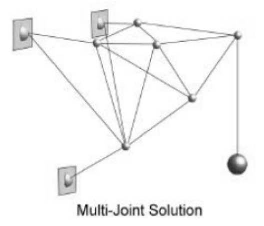


Figure 2. An optimized solution to the Truss problem

The second component of the Optimal Truss system generates the permutation of all possible truss topologies within the domain space, ensuring that the configuration with minimal mass is a global minimum. The domain space for the initial implementation is a 2-dimensional mesh of points, defining the rectangle formed between the attachment points and the load bearing point. Exploring all possible configurations results in a combinatorial explosion as the mesh size is increased and this served as the first point of parallelism in the implementation. The task of parallelizing topology generation can be equally divided among the available nodes. This can be accomplished in an "embarrassingly parallel" manner, where each row of the mesh is assigned to a different processor to permute.

The third component of the system performs geometry assignment for all trusses. After generation, a topology defines the path of each truss from the attachment points to the load bearing point, but it does not specify what type of member connects each joint. In this stage, either a strut or a cable is substituted for each member, flushing out all permutations. Once the geometry has been assigned, it can be given to a processor to compute the mass of the truss and to determine whether the topology is valid under the equilibrium and deformational constraints.

Now that the description of the Optimal Truss Design problem, used in our case study, has been explained, it is prudent to illustrate and investigate the measures applied to the problem.

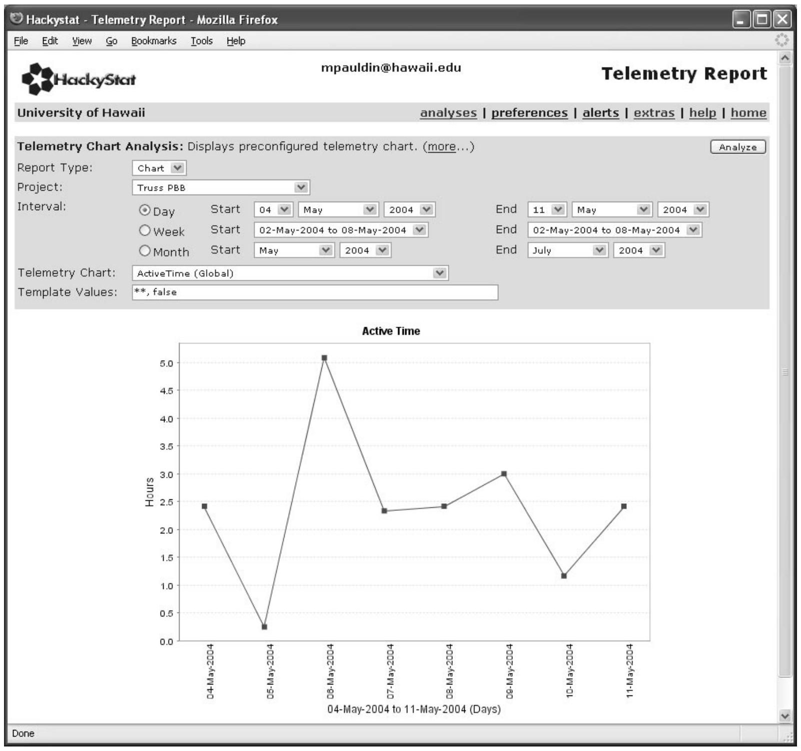
4.2. Active Time

Active Time is a measure of the time spent by developers editing source code (or other files) related to the system. Active Time can be collected automatically through the use of sensors attached to the editor used by developers. The sensors collect active time via a timer-based process inside the editor that wakes up every 30 seconds and checks to see if the active buffer has changed in identity or size since the last 30 seconds. If so, a timestamped "statechange" event is sent to the Hackystat server. Active Time does not reflect effort spent by developers on the project that does not involve editing files, including time spent viewing a file without performing editing actions. Support for non-editing activities such as "reading" is a subject of future research, but even the restricted definition of Active Time appears useful in the HPC context as a proxy for overall effort. For example, it helps a development team answer questions such as: *"How much of the overall development effort was spent editing files?"* or *"Did all team members devote equal time to writing code?"* or *"When was team effort focussed on code development during the project?"*

Figure 3 shows the Active Time associated with development of the Optimal Truss Design application for a sample period in May, 2004.

4.3. Most Active File

A measure related to Active Time is the "Most Active File". One way to abstract the raw event stream sent from an


Hackystat - Telemetry Report - Mozilla Firefox

mpauldin@hawaii.edu

Telemetry Report

University of Hawaii

analyses | preferences | alerts | extras | help | home

Telemetry Chart Analysis: Displays preconfigured telemetry chart. (more...)

Report Type: Chart

Project: Truss PBB

Interval: [✓ Day] Start 04 May 2004 End 11 May 2004

Week Start 02-May-2004 to 08-May-2004 End 02-May-2004 to 08-May-2004

Month Start May 2004 End July 2004

Telemetry Chart: ActiveTime (Global)

Template Values: **, false

Figure 3. Active time

editor-based Hackystat server begins by representing each day as a sequence of 288 five minute intervals. If a developer actively edits one or more files within a five minute period, then determine which file was edited most during that five minutes, and assign the "credit" for that five minute interval to that file and that file alone, which we call the "Most Active File". (We performed a calibration study which found this to be a reasonable abstraction.) The Most Active File abstraction may be useful in the HPC context as a way of determining what specific files were the focus of developer attention, and how that focus of attention changed over the course of development.

For example, Figure 4 shows the Most Active Files associated with Optimal Truss Design during the first few days of this time interval.

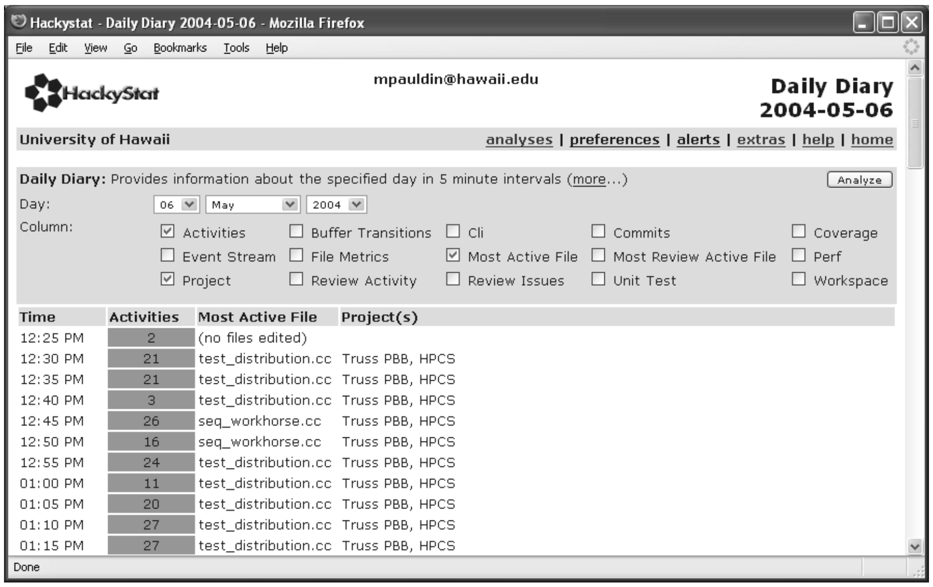
4.4. Command Line Invocations

In addition to time spent editing files in an editor, HPC development frequently involves extensive use of shell processes to invoke programs such as make, gcc, etc. We have implemented a sensor for the Unix command shell (based upon the 'history' shell mechanism) to record these command line invocations. Command Line Invocation data can be useful in the HPC context as a way of providing further insight into the types of activities performed by developers during the development of the HPC code. For example, if the HPC developer spends significant time working at the command line without concurrent editing of code, then it might be useful to develop an enhanced representation of Active Time that accounts for this type of effort as well. While the current sensor only captures command invocations and not their results, it might be useful to extend the sensor to capture the results of command line invocations in certain circumstances. For example, recording whether or not a compilation succeeded or failed as well as what types of run-time errors occur could help identify potential development bottlenecks.

Figure 5 illustrates Command Line Invocation data for a portion of one day during the development of the Optimal Truss Design system.

4.5. Parallel and Serial Size

To understand HPC software development, it helps to be able to represent both "serial" and "parallel" code. We have enhanced our size measurement tool, LOCC, with a token-based counter for C++ that allows us to count non-comment source lines of code, and determine for each line of code whether or not an MPI directive occurs on it. Thus, for HPC programs built using C++ and MPI, we can determine (a) the total number of files in the system, (b) the total non-


Hackystat - Daily Diary 2004-05-06 - Mozilla Firefox
File Edit View Go Bookmarks Tools Help
HackyStat
Analyze
Done

mpauldin@hawaii.edu

Daily Diary
2004-05-06

University of Hawaii analyses | preferences | alerts | extras | help | home

Daily Diary: Provides information about the specified day in 5 minute intervals (more...)

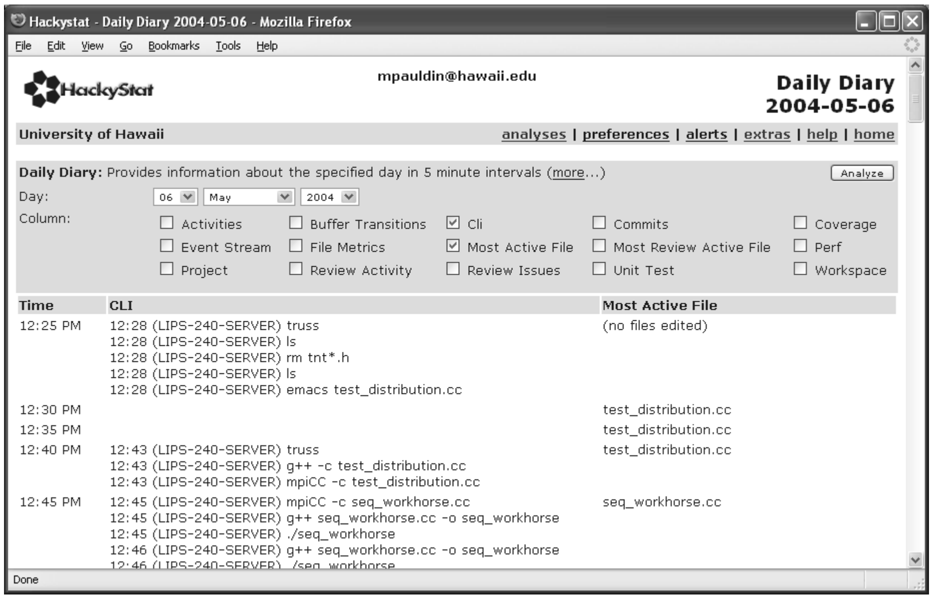
Day: 06 May 2004

Column:

☑ Activities	☐ Buffer Transitions	☐ Cli	☐ Commits	☐ Coverage
☐ Event Stream	☐ File Metrics	☑ Most Active File	☐ Most Review Active File	☐ Perf
☑ Project	☐ Review Activity	☐ Review Issues	☐ Unit Test	☐ Workspace

Time	Activities	Most Active File	Project(s)
12:25 PM	2	(no files edited)	
12:30 PM	21	test_distribution.cc	Truss PBB, HPCS
12:35 PM	21	test_distribution.cc	Truss PBB, HPCS
12:40 PM	3	test_distribution.cc	Truss PBB, HPCS
12:45 PM	26	seq_workhorse.cc	Truss PBB, HPCS
12:50 PM	16	seq_workhorse.cc	Truss PBB, HPCS
12:55 PM	24	test_distribution.cc	Truss PBB, HPCS
01:00 PM	11	test_distribution.cc	Truss PBB, HPCS
01:05 PM	20	test_distribution.cc	Truss PBB, HPCS
01:10 PM	27	test_distribution.cc	Truss PBB, HPCS
01:15 PM	27	test_distribution.cc	Truss PBB, HPCS

Figure 4. Most Active File


Hackystat - Daily Diary 2004-05-06 - Mozilla Firefox
File Edit View Go Bookmarks Tools Help
HackyStat
Analyze
Done

mpauldin@hawaii.edu

Daily Diary
2004-05-06

University of Hawaii analyses | preferences | alerts | extras | help | home

Daily Diary: Provides information about the specified day in 5 minute intervals (more...)

Day: 06 May 2004

Column:

☐ Activities	☐ Buffer Transitions	☑ Cli	☐ Commits	☐ Coverage
☐ Event Stream	☐ File Metrics	☑ Most Active File	☐ Most Review Active File	☐ Perf
☐ Project	☐ Review Activity	☐ Review Issues	☐ Unit Test	☐ Workspace

Time	CLI	Most Active File
12:25 PM	12:28 (LIPS-240-SERVER) truss 12:28 (LIPS-240-SERVER) ls 12:28 (LIPS-240-SERVER) rm tnt*.h 12:28 (LIPS-240-SERVER) ls 12:28 (LIPS-240-SERVER) emacs test_distribution.cc	(no files edited)
12:30 PM		test_distribution.cc
12:35 PM		test_distribution.cc
12:40 PM	12:43 (LIPS-240-SERVER) truss 12:43 (LIPS-240-SERVER) g++ -c test_distribution.cc 12:43 (LIPS-240-SERVER) mpiCC -c test_distribution.cc	test_distribution.cc
12:45 PM	12:45 (LIPS-240-SERVER) mpiCC -c seq_workhorse.cc 12:45 (LIPS-240-SERVER) g++ seq_workhorse.cc -o seq_workhorse 12:45 (LIPS-240-SERVER) ./seq_workhorse 12:46 (LIPS-240-SERVER) g++ seq_workhorse.cc -o seq_workhorse 12:46 (LIPS-240-SERVER) ./seq_workhorse	seq_workhorse.cc

Figure 5. Command Line Invocations

commented size of each file in the system; (c) whether or not a file consists purely of serial (non-MPI) code or not; (d) for files containing MPI directives, the frequency of occurrence of each MPI directive; and (e) for files containing MPI code, what percentage of the non-comment source lines of code contained an MPI directive.

Figures 6, 7, and 8 provide perspectives on size data for the Optimal Truss Design system.

4.6. Functionality

We have defined a process for measuring the functionality of an HPC application and tracking its development progress through the use of unit testing. We have termed this approach "Progress Assessment through Milestone Tests" (PAMT).

Essentially, PAMT is a process in which HPC application designers draft the specification for the system as a set of unit tests prior to development. Each unit test is defined such that it represents a milestone, or significant aspect of application functionality. Quantitative interpretation of "significant" is determined by the application designer or program manager and is expected to vary between HPC projects. Defining the set of milestone tests prior to development provides a specification for the system and also serves as a mechanism to promote test driven design.

Once the milestone tests have been defined, the development team has a concrete set of tests to implement that, together, represent the functionality of the entire system. The development team can then implement the milestone tests in any order and their progress through the application can be monitored. System progress and functionality is measured by investigating the number of milestone tests passing in ratio to the total number of milestone tests representing the system. In most cases, a development team will begin implementation with zero milestone tests passing and finish development when all milestone tests pass.

For the Optimal Truss Design problem, a set of 10 milestone tests were defined prior to implementation. Individually, each test represents a significant functionality of the application and together they provide a specification for the entire system. For the Optimal Truss problem, the milestone tests were written in CppUnit, a unit test framework for the C++ programming language. Below is an example of a single milestone test for the Optimal Truss problem.

> **Milestone Test 4:** This test verifies that the application is capable of representing a 2-dimensional topology. In the Optimal Truss specification, a topology is defined as a set of 2 trusses that individually connect the 2 attachment points to the load bearing point. Interconnections (members) between the trusses are allowed. Therefore, for this milestone test, given 2 attachment points, a load bearing point and the number of joints, the application must be able to query:
>
> 1. Each of the trusses connecting the 2 attachment points to the load bearing point
> 2. The set of members composing one of the trusses in the topology
> 3. Given a truss, whether it is part of the topology

From this chart is is evident that during the development period from 04-May-2004 through 11-May-2004 that the Optimal Truss application progressed from 1 milestone test passing at the beginning of the interval to 5 milestone tests passing at the end. It is important to note that this interval represents a sample of the development period and does not capture start to finish. In addition, this trend indicates a consistent increase in passing milestone tests. However, it is quite possible for development to lower the number of successful milestone tests, indicated by a negative slope in the trend.

4.7. Performance

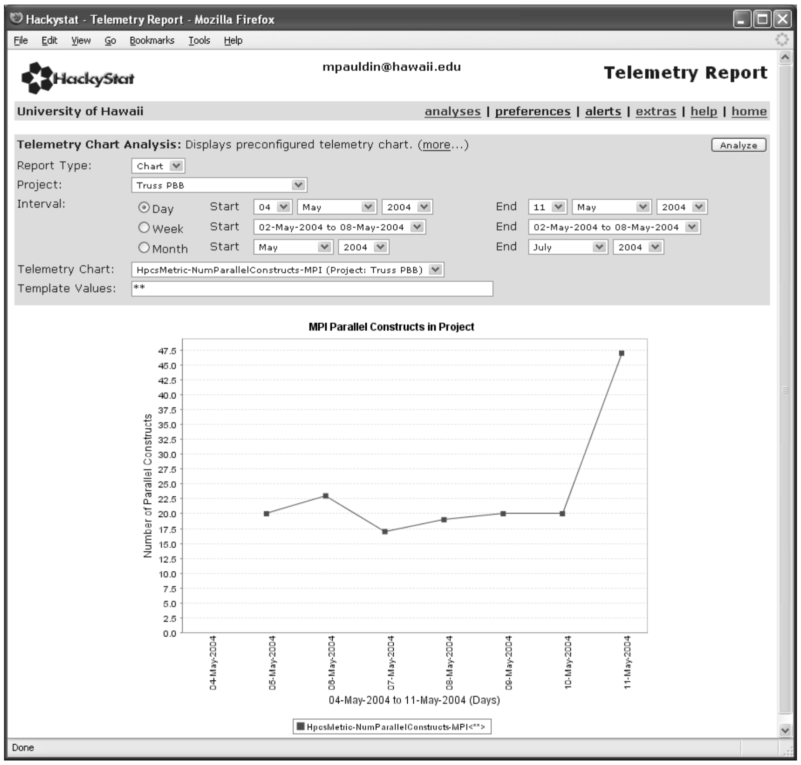
The high performance computing community has developed a broad range of standard measures to characterize parallel performance, including degree of parallelism, average parallelism, speedup, redundancy, and utilization. In this research, we are not attempting to specify the "right" performance measure for any particular application area. Instead, we advocate that performance be measured regularly throughout development using as many metrics as necessary to best characterize the application.

Performance measures are not generally interesting as absolute numbers, since the absolute values are obviously dependent upon current hardware and other physical resources. Performance measures are interesting as relative numbers, in the sense that the way they change over time tells us whether or not and to what extent developers could tune an initial implementation to improve its performance, how the code evolved to obtain this performance increase, and whether or not functionality was sacrificed in order to do so.

Figure 10 shows the execution (wall time) performance of the Optimal Truss Design system developed in the case study for a sample time interval.

5. Conclusions

After accumulating the data trends provided by the Hackystat system, we are able to gain insights that assist in understanding the development of HPC applications. From


Hackystat - Telemetry Report - Mozilla Firefox
File Edit View Go Bookmarks Tools Help
HackyStat
mpauldin@hawaii.edu
Telemetry Report
University of Hawaii
analyses | preferences | alerts | extras | help | home
Analyze
MPI Parallel Constructs in Project
Number of Parallel Constructs
0.0 2.5 5.0 7.5 10.0 12.5 15.0 17.5 20.0 22.5 25.0 27.5 30.0 32.5 35.0 37.5 40.0 42.5 45.0 47.5
04-May-2004 05-May-2004 06-May-2004 07-May-2004 08-May-2004 09-May-2004 10-May-2004 11-May-2004
04-May-2004 to 11-May-2004 (Days)
■ HpcsMetric-NumParallelConstructs-MPI<**>
Done

Telemetry Chart Analysis: Displays preconfigured telemetry chart. (more...)

Report Type: Chart

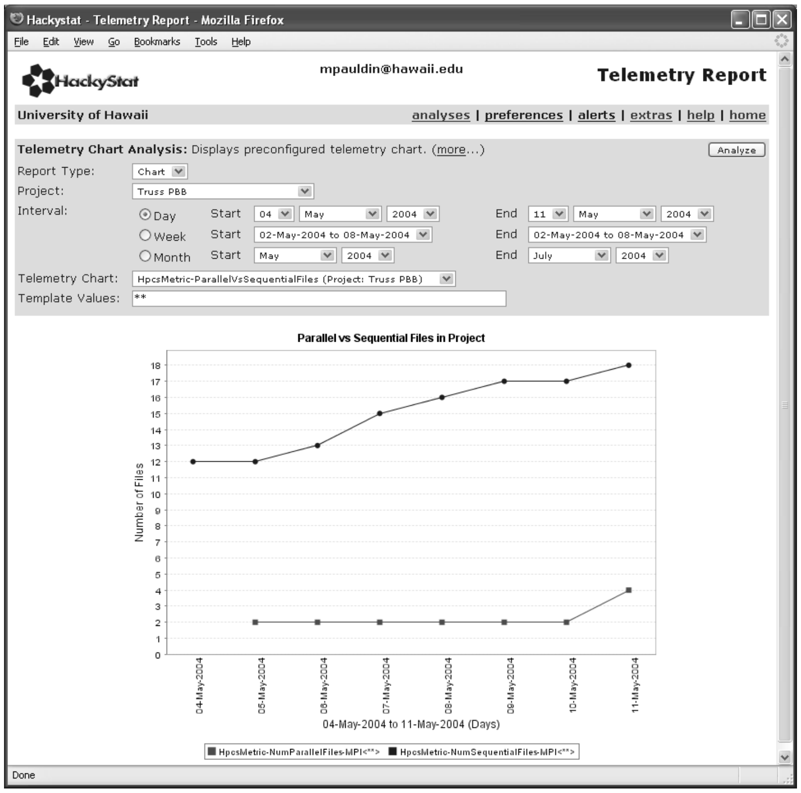
Project: Truss PBB

Interval: [✓ Day] Start 04 May 2004 End 11 May 2004; Week Start 02-May-2004 to 08-May-2004 End 02-May-2004 to 08-May-2004; Month Start May 2004 End July 2004

Telemetry Chart: HpcsMetric-NumParallelConstructs-MPI (Project: Truss PBB)

Template Values: **

Figure 6. Parallel vs. sequential constructs


Hackystat - Telemetry Report - Mozilla Firefox
File Edit View Go Bookmarks Tools Help
HackyStat
mpauldin@hawaii.edu
Telemetry Report
University of Hawaii
analyses | preferences | alerts | extras | help | home
Analyze
Parallel vs Sequential Files in Project
Number of Files
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18
04-May-2004 05-May-2004 06-May-2004 07-May-2004 08-May-2004 09-May-2004 10-May-2004 11-May-2004
04-May-2004 to 11-May-2004 (Days)
■ HpcsMetric-NumParallelFiles-MPI<**> ■ HpcsMetric-NumSequentialFiles-MPI<**>
Done

Telemetry Chart Analysis: Displays preconfigured telemetry chart. (more...)

Report Type: Chart

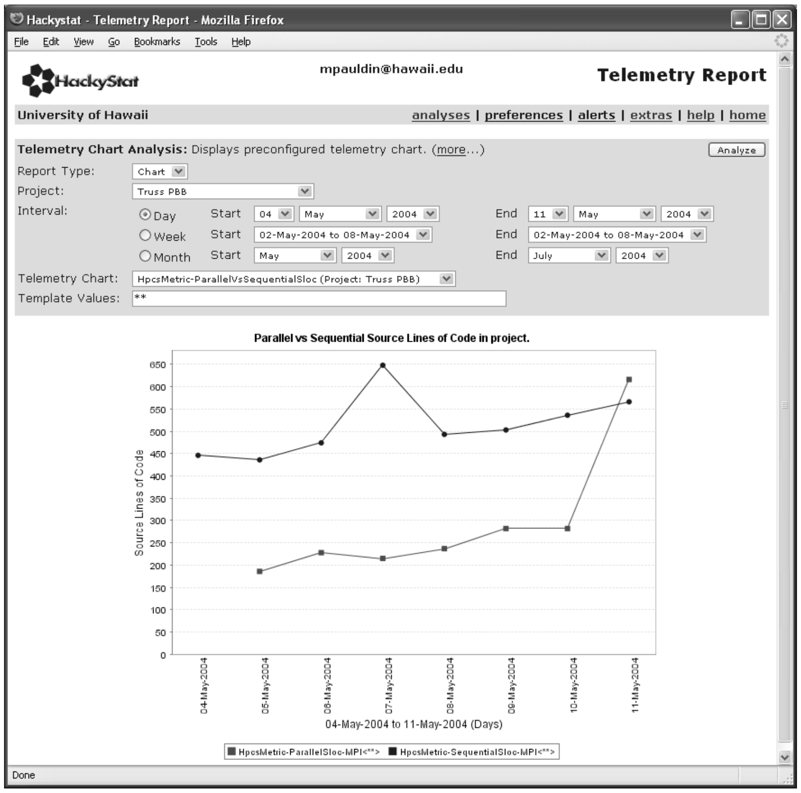
Project: Truss PBB

Interval: [✓ Day] Start 04 May 2004 End 11 May 2004; Week Start 02-May-2004 to 08-May-2004 End 02-May-2004 to 08-May-2004; Month Start May 2004 End July 2004

Telemetry Chart: HpcsMetric-ParallelVsSequentialFiles (Project: Truss PBB)

Template Values: **

Figure 7. Parallel vs. Sequential Files


Hackystat - Telemetry Report - Mozilla Firefox

mpauldin@hawaii.edu

Telemetry Report

University of Hawaii

Telemetry Chart Analysis: Displays preconfigured telemetry chart. (more...)

Report Type: Chart

Project: Truss PBB

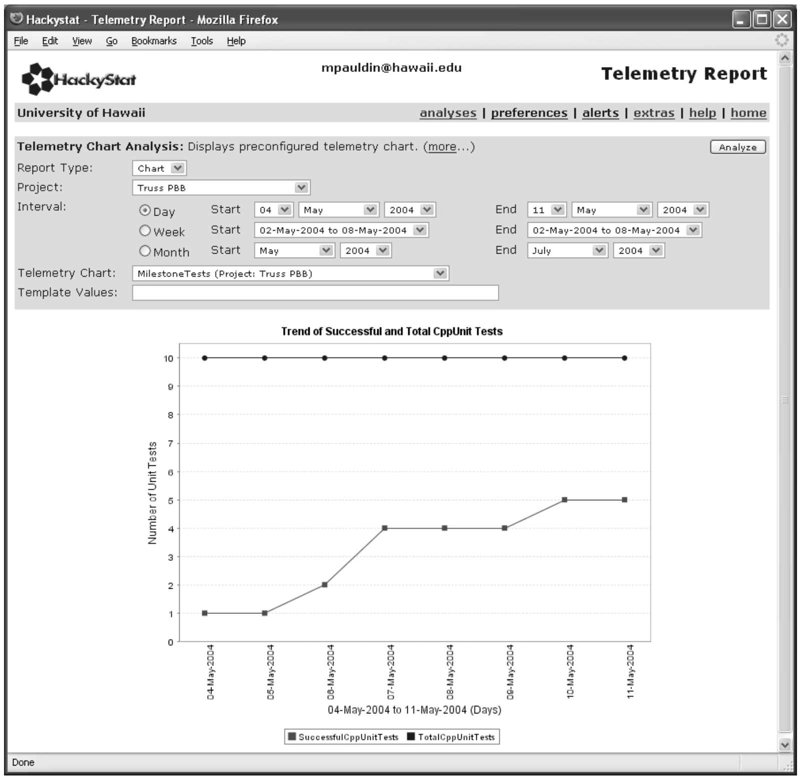
Interval: [✓ Day] Start 04 May 2004 End 11 May 2004
Week Start 02-May-2004 to 08-May-2004 End 02-May-2004 to 08-May-2004
Month Start May 2004 End July 2004

Telemetry Chart: HpcsMetric-ParallelVsSequentialSloc (Project: Truss PBB)

Template Values: **

File
Edit
View
Go
Bookmarks
Tools
Help
analyses | preferences | alerts | extras | help | home
Analyze
Parallel vs Sequential Source Lines of Code in project.
Source Lines of Code
650
600
550
500
450
400
350
300
250
200
150
100
50
0
04-May-2004
05-May-2004
06-May-2004
07-May-2004
08-May-2004
09-May-2004
10-May-2004
11-May-2004
04-May-2004 to 11-May-2004 (Days)
▨ HpcsMetric-ParallelSloc-MPI<**>
■ HpcsMetric-SequentialSloc-MPI<**>
Done

Figure 8. Parallel vs. Sequential SLOC


Hackystat - Telemetry Report - Mozilla Firefox

mpauldin@hawaii.edu

Telemetry Report

University of Hawaii

Telemetry Chart Analysis: Displays preconfigured telemetry chart. (more...)

Report Type: Chart

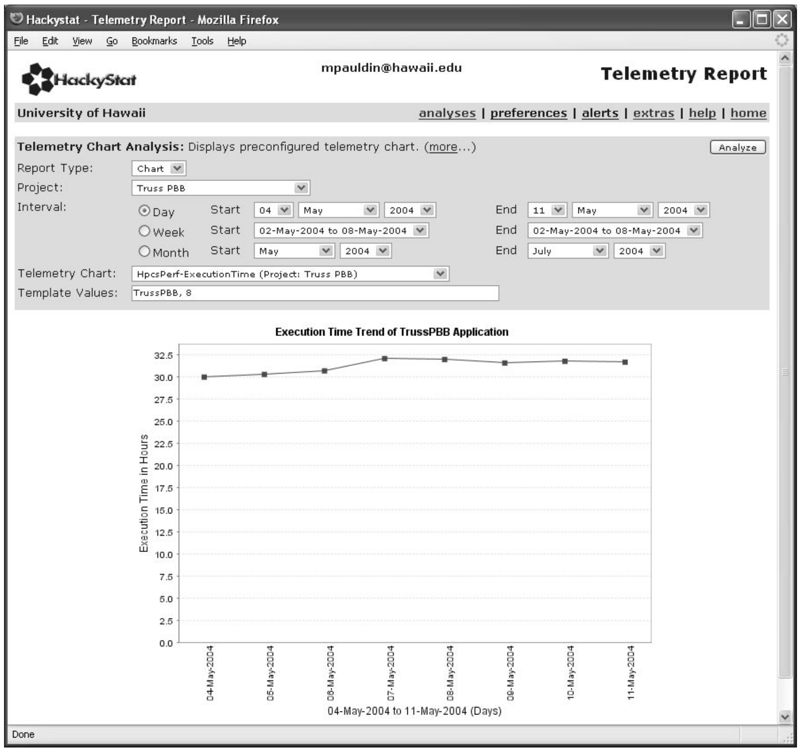
Project: Truss PBB

Interval: [✓ Day] Start 04 May 2004 End 11 May 2004
Week Start 02-May-2004 to 08-May-2004 End 02-May-2004 to 08-May-2004
Month Start May 2004 End July 2004

Telemetry Chart: MilestoneTests (Project: Truss PBB)

Template Values:

Figure 9. Progress Assessment through Milestone Tests


Hackystat - Telemetry Report - Mozilla Firefox

mpauldin@hawaii.edu

Telemetry Report

University of Hawaii

Telemetry Chart Analysis: Displays preconfigured telemetry chart. (more...)

Report Type: Chart

Project: Truss PBB

Interval: [✓ Day] Start 04 May 2004 End 11 May 2004

Week Start 02-May-2004 to 08-May-2004 End 02-May-2004 to 08-May-2004

Month Start May 2004 End July 2004

Telemetry Chart: HpcsPerf-ExecutionTime (Project: Truss PBB)

Template Values: TrussPBB, 8

Figure 10. Truss Execution Time Performance

the graphs presented earlier, we are able to make interpretations about development activities, development progress and application performance and functionality tradeoffs.

5.1. Development Activities

From the data presented in Figure 5 we are able to interpret the developer activities of that particular day. The daily dairy for 06-May-2004 lists all the commands issued to the console on that day. Figure 5 is a sample of all the command line invocations issued, but it provides insight to the developer activities on that day.

For example, from the command line invocations and most active file data for 06-May-2004, we can observe that the developer devoted the most time on the *test_distribution.cc* file. It so happens that this file implements the distribution of 2-dimensional mesh from which the truss topologies are constructed. Furthermore, the distribution of topologies is a significant function of the Optimal Truss problem and has been designated as a milestone test of the system. Therefore, from this data, an observer can conclude that the developer was investing his efforts on implementing functionality on this day, rather than on increasing performance by optimizing code.

In addition, an observer, such as a project manager or the developer himself, can observe the active time trend in Figure 3 to understand the time invested to implement a particular milestone. For example, in Figure 3 on 06-May-2004, it is evident that the developer spent over 5 hours editing code to implement the topology distribution milestone.

5.2. Development Progress

The data presented in Progress Assessment through Milestone Tests (PAMT) chart, as illustrated in Figure 9, provides a clear illustration of the real-time progress being made on the Optimal Truss problem.

There are two trends presented in this figure, one representing the total number of milestone tests defined for the Optimal Truss problem and the other representing the number of milestone tests passing at the conclusion of each day.

In the Optimal Truss problem, the total milestone tests are represented by the horizontal line fixed at 10 unit tests. This indicates that there are 10 milestone tests encompassing the Optimal Truss problem and that the project manager has not added or removed any of these milestones during this time interval. It is quite possible that a project manager may have to alter milestones in order to meet deadlines and this analysis provides a trend for this purpose. For example, if the total milestone tests is altered, the total tests trend on the chart will move up or down accordingly.

The PAMT chart also allows an observer to track the progress made through the application. For example, in this figure, the lower trend represents the number of milestone tests passing on each day. Every time a new milestone test passes, it indicates that another unit of functionality has been added to the system provided that all previously passing tests still pass after the change.

Coupling the PAMT data with the active time data in Figure 3, an observer is also able to interpret a quantitative measurement of how much development time was devoted to a particular unit of functionality. For example, on 06-May-2004, approximately 5 hours of editing were invested to add one unit of functionality to the application. This is indicated by the number of passing milestone tests increasing from one to two on this day. In addition, an observer can quickly understand the percentage complete of the system. On 06-May-2004, the Optimal Truss problem has 2 out of 10 milestone tests passing and is therefore 20% complete.

5.3. Application Performance and Functionality Tradeoffs

When one combines the data presented in the Performance chart in Figure 10 with the PAMT Functionality chart in Figure 9, it reveals an example of the interactions between performance and functionality in HPC development.

One of the primary objectives of HPC development is to obtain the fastest possible execution time of the system. This goal influences developers to frequently (if not constantly) think about or perform optimization on their code.

However, as functionality is added to the application, it is common for the performance of the system to decrease, indicated by an increase in execution time.

The data presented in figures 10 and 9 reveal this performance and functionality tradeoff. For example, execution time between 04-May-2004 and 07-May-2004 increaseses roughly from 30.0 hours to 32.5 hours. On the other hand, 3 additional milestone tests, representing system functionality, are implemented successfully during this interval. This indicates that three units of functionality have been added at the cost of an addition of approximately 10% in execution time.

Data presented in these figures allow project managers to understand how functionality increases affect system performance. It also gives them a starting point to determine which functionality should be optimized in the case where the performance degradation is not acceptable. Trends such as these enable project managers and developers to understand the development process and make in-process decisions to affect the development outcome.

In conclusion, we have found that Active Time, Most Active File, Command Line Invocations, Parallel and Serial Lines of Code, Milestone Test Success, and Performance constitute an interesting set of process and product measures that can be automatically collected during HPC development. As our case study continues, we will look for other opportunities to use this measures to gain insight into opportunities to improve high performance computing.

References

[1] The DARPA high productivity computing systems program. http://www.highproductivity.org/.

[2] B. Boehm. *Software Engineering Economics*. Prentice Hall, 1981.

[3] J. Gustafson. Purpose Based Benchmarks. *International Journal of High Performance Computing Applications*, 12(1):14, 2004.

[4] W. S. Humphrey. *A Discipline for Software Engineering*. Addison-Wesley, New York, 1995.

[5] P. M. Johnson and A. M. Disney. The personal software process: A cautionary case study. *IEEE Software*, 15(6), November 1998.

[6] P. M. Johnson, H. Kou, J. M. Agustin, C. Chan, C. A. Moore, J. Miglani, S. Zhen, and W. E. Doane. Beyond the personal software process: Metrics collection and analysis for the differently disciplined. In *Proceedings of the 2003 International Conference on Software Engineering*, Portland, Oregon, May 2003.

[7] P. M. Johnson, C. A. Moore, J. A. Dane, and R. S. Brewer. Empirically guided software effort guesstimation. *IEEE Software*, 17(6), December 2000.

[8] D. A. Reed, editor. *The Roadmap for the Revitalization of High-End Computing*. Computing Research Association, 2003.