# What Do Programmers of Parallel Machines Need? A Survey

Susan Squires
Sun Microsystems Inc.
Santa Clara, CA
susan.squires@sun.com

Walter F. Tichy
University of Karlsruhe
Karlsruhe, Germany
tichy@ipd.uka.de

Lawrence Votta
Sun Microsystems Inc.
Menlo Park, CA
lawrence.votta@sun.com

**Abstract:** We performed semi-structured, open-ended interviews with 11 professional developers of parallel, scientific applications to determine how their programming time is spent and where tools could improve productivity. The subjects were selected from a variety of research laboratories, both industrial and governmental. The major findings were that programmers would prefer a global over a per-processor view of data structures, struggle with load balancing and optimizations, and need interactive tools for observing the behavior of parallel programs. Furthermore, handling and processing massive amounts of data in parallel is emerging as a new challenge.

## 1 Introduction

Programmer productivity has been improving, but at a much lower rate than processor speeds [4]. This disparity has led to a shift of bottlenecks in the supercomputer community consisting of the computer and human.

In many cases, the bottleneck for achieving results is no longer compute time but programmer time. To explore the phenomena, questions in our survey were therefore aimed at obtaining a time and task breakdown for development work and eliciting suggestions of where tool support might improve programmer productivity.

## 2 Method

### 2.1 Survey Sampling

The sample size of participants in this study is in keeping with the techniques for purposive sampling. Purposive sampling is used for exploration or pilot studies when the demographics of the participants are clear but the unit of analysis or important questions are unknown [1]. In qualitative studies of nonrandom cases typical of a class of participants, such as programmers of parallel machines, purposive sampling can be “sufficient to display something of substantive importance” [2]. For such surveys experienced researchers recommend sampling at least 6 to 7 participants to document core experiences that can generate questions for follow-up quantitative studies (see for example [5], [6], [8]).

### 2.2 Interview Construction

Semi-structured, open-ended interviews follow a set of well-understood rules. They build rapport in the first segment and then look for deeper information. The interviewers summarize information in order to confirm the data with the respondent.

While open-ended conversations are

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

allowed, interviewers make sure all questions are eventually addressed. They listen for native language: words, terms and descriptions to fully understand the world of the respondent. Techniques such as reflecting back help the interviewer achieve a deeper emphatic understanding [6], [8].

An interview guide was initially designed with the aid of a social scientist and tested with two subjects. The feedback was used to clarify the statement of purpose and modify the questions. Since part of the interview was to elicit time and task breakdown, and these estimates are most reliable when done within the context of a step-by-step description of an actual project, the interviewer asked the subject to identify a parallel programming project that he/she remembered well. The project might involve writing new code or adapting existing code. The respondent was asked to describe the project in detail and to identify phases that the project went through and how much time the phases took. The phases suggested were:

1. understanding the problem and/or existing software;
2. studying documentation;
3. designing a solution;
4. writing new code or adapting existing code;
5. testing, debugging and checking;
6. correctness of the results;
7. optimizing the code; and
8. scheduling production runs.

Respondents were, of course, free to specify different phases, for instance those of a prototype workflow. Respondents were asked to identify the most difficult and time-consuming phase. Furthermore, we included a specific question to estimate the amount of communication and synchronization code in the application.

The next block of questions centered on tools. First, respondents were asked what programming languages, communication libraries, debuggers, version control, or other tools they used. Second, the interviewer probed explicitly for tools that might speed up the development if the programmers had to do the same project again. In this part of the interview, summarization of the ideas expressed was particularly important.

In closing, the interviewer asked whether the respondent would like to add anything or had any questions. The interviewer finally expressed thanks and left the conversation open for quick follow-up calls or emails.

### 2.3 Respondents

Candidates were initially approached to participate in the study based on their senior status and experience. In all, 16 candidates were identified and 11 of them participated in the survey. The remaining five did not reply to e-mail. All respondents were professionals with a PhD in a scientific discipline, and their main work was developing parallel applications or benchmarks. The 11 respondents came from both industrial and governmental research labs in the US and Germany. No claims are made that the sample is representative, but the areas in which they have experience is found widely scattered, with 4 of the ll were in the area of computational physics called Lattice Quantum Chromodynamics (QCD). Six of the respondents worked in teams while the other five worked alone when developing the application discussed. All candidates were male.

**Table 1: Areas and Number of Respondents**

| ***Area*** | ***# of Respondents*** |
|---|---|
| Lattice QCD | 4 |
| Particle Physics Data Analysis | 1 |
| Gravitational Wave Analysis | 1 |
| Quantum Chemistry | 1 |
| Interval Arithmetic | 2 |
| Reengineering | 1 |
| Illumination modelling | 1 |
| Total | 11 |

## 3 Findings

### 3.1 Application Classes

We found three different application classes: computation-bound with regular structure, computation-bound with irregular structure, and I/O-bound. Importantly, we noted that developers approach these classes in markedly different ways and face different challenges.

**Computation-bound with Regular Structure (4 Respondents)**

This class is characterized by large, multi-dimensional arrays that are distributed over multiple compute nodes and processed in parallel. Developers for this class struggle with the per-processor view imposed by MPI and similar communication packages on distributed memory machines. A per-processor view of an array means that programmers divide the array in chunks that are stored on individual computer nodes and write their programs for these chunks rather than for the array as a whole. The undesirable side effect of this approach is that whenever data needs to be fetched from, or stored on, another chunk, special communication code, typically in MPI, must be inserted. The extra MPI code was typically 50-60% of the entire code.

In one reengineering project, where the respondent described how he took existing benchmarks written with MPI and rewrote them with a global or shared-memory view, the code became dramatically shorter: it shrank to one fifth to one tenth of the original. This respondent spent extra effort writing code in as clear a way as possible and eschewed all optimizations that might obfuscate the intent of the algorithm. The differences in code bloat observed by this individual may be due to the per-processor view requiring code in addition to the MPI code. Optimizations and sloppy coding may also account for some extra code.

MPI code is not easy to write; some respondents called it the most difficult aspect of a parallel application. A global view of arrays, as in HPF or ZPL [3], was deemed far superior. One respondent said: *"If we had this, a lot of programmers could do something more meaningful than writing MPI."*

One-sided communication was seen as mitigating the problems that MPI causes, without going all the way to a global view. With two-sided communication as in MPI, the programmer must write two statements for a single communication: one to send the data, and one to receive and store it into the target variable. In addition, sender and receiver must be synchronized in such a manner that the sent message is received by the correct receiving statement, not an earlier or later one. In contrast, one-sided communication allows the sender to deposit a datum into a target variable at the receiver without the cooperation of the receiver. Thus, the communication code is roughly half as long and synchronization is needed less often.

In large applications, the problem often is to redistribute data from one step to the next, in order to be able to process it in parallel. Redistribution is costly on current machines.

**Computation-bound with Irregular Structure (5 Respondents)**

This class of applications is characterized by many sub-problems to be solved, whose number and distribution over the computation nodes is unpredictable. One typically finds irregular tree structures or parameter space searches in this class of problems. Each processor maintains a list of sub-problems that need to be processed. During processing, new sub-problems may get added to the list. When the list's length falls below a certain threshold, the processor obtains additional work from neighboring nodes (work stealing). Alternatively, an overloaded node can spread its work to less loaded nodes. Other problems are irregular meshes that are subdivided dynamically. N-body problems also have the characteristic of unbalanced load, but in order to redistribute the load, the data structure (a space-dividing oct-tree) needs to be redistributed.

One respondent reported that parallelizing a serial version of an irregular problem took as long as writing the sequential version, even though the computational kernels remained unchanged. Another respondent reported that the statistical distribution of sub-computations in his application was reasonable so no load balancing was necessary.

It is not clear whether load-balancing in irregular applications can be done by libraries (providing, for instance, distributed work queues) or by language constructs, or is highly application dependent. It appears to be a neglected area. Three respondents worked on this type of problem.

**I/O-bound (2 Respondents)**

This class of applications is characterized by massive data files that need to be processed in parallel. Examples are the interpretation of detector data in high-energy physics or the analysis of gravitational wave data. Here, the problem is not parallelism or synchronization. These applications are "embarrassingly parallel" in that they are easily split in many sub-computations that do not communicate. The problem, however, is to stream the data through the machine in parallel and collect and merge the results at the end. Truly parallel file systems and sufficient I/O bandwidth are essential. Since scalable parallel file systems, in which a single file can be accessed concurrently are rare, developers end up splitting the files manually and storing the pieces on different file servers. An additional problem is bookkeeping: knowing which data sets have been processed. This task can become very important because detectors may deliver Terabytes of data per day and do this for several years. System administrators are quickly overwhelmed with the task of keeping track what data was and what was not analyzed.

A second problem with massive data handling has to do with how to skip errors in the data. Erroneous records can cause the analysis program to fail. A smart analysis tool should be able to identify erroneous records and skip them on the next run.

Though physics has the most data to process, other scientific areas may soon have similar requirements. Biologists already handle large amounts of data.

## 3.2 Work and Time Breakdown

Respondents were asked to identify phases in their development work and estimate the time they spent for each. Most of them thought that a raster of problem understanding, designing algorithms, followed by implementation and debugging and optimization reflected their work adequately. One respondent followed a prototype model, characterized by an initial prototype followed by a complete rewrite with successive releases.

Table 2 provides an overview of the time breakdown. Problem understanding varied – around 20% of total effort, with the exception that the reengineering effort consumed 50%. The bulk of the time went into implementation and debugging. Only Lattice QCD researchers spent significant time on optimizing the computational kernels (writing assembly code for them). Other respondents said their programming was permanent optimization (parallelization and communication). Nobody mentioned that reading documentation was an issue.

MPI use varied widely. For instance, the data analysis problems needed little MPI, since only data scattering and gathering needed to be implemented.

**Table 2: Percent of Time in Task for Each Respondent**

| **Area of Re-spon-dent** | **% of Time in Task** | | | | |
|---|---|---|---|---|---|
| | **Under-stand-ing** | **Design** | **Imple-men-tation** | **Opti-miza-tion** | **Runs** |
| **QCD1** | 22 | 10 | 56 | 12 | 0 |
| **QCD2** | 0 | 30 | 60 | 10 | 0 |
| **QCD3** | 0 | 0 | 0 | 100 | 0 |
| **QCD4** | 0 | 20 | 30 | 50 | 0 |
| **Data Anal1** | 0 | 20 | 80 | 0 | 0 |
| **Data Anal2** | *Prototyping process used is not consistent with a linear model of time usage.* | | | | |
| **Chem** | 20 | 20 | 40 | 0 | 20 |
| **Inter-val Arith1** | 20 | 50 | 25 | 5 | 0 |
| **Inter-val Arith2** | 40 | 0 | 60 | 0 | 0 |
| **Reeng-ineer-ing** | 50 | 25 | 25 | 0 | 0 |
| **Illumi-nation** | 25 | 0 | 50 | 0 | 25 |

## 3.3 Tools

Respondents used C/C++/Fortran compilers, editors, MPI libraries, and problem specific libraries. A few also used version control tools (SCCS, RCS, CVS).

GDB or DBX was used by a few to debug serial programs. Surprisingly, only one respondent was using a parallel debugger (TotalView). The rest used print statements that produced trace files for later analysis.

The reasons given for using print statements with trace files varied. Some stated they were unwilling to learn a new debugger that might disappear again in a short time. They felt that tools should be in the public domain to improve the chances that the tools survived the ups and downs of vendors.

For other respondents, there was simply no interactive debugger available, because they were working in a batch environment. Several also said that debuggers were not helpful because they tended to produce too much trace data. It was simply easier to implant output statements or output macros that could be turned on and off than "taming" a debugger.

Finally, those who spent a lot of time on optimization thought that they were more interested in execution profile data, because logic errors weren't that frequent or difficult to detect. However, debuggers are meant for detecting logic bugs and are not good profiling. In any case, debuggers and profilers should introduce very little overhead.

One team, the Graviational Analysis team, switched from C/MPI to Matlab. In the first iteration, the application took ten months to build using C/MPI and the resulting program, running on a PC cluster, was disappointingly slow – three weeks to process one week of data.

Further, the programming environment for the cluster was poor and awkward, since interactive debugging was impossible.

In a second attempt, the Gravitational Analysis Team rebuilt the entire system using Matlab only. Even though the team was learning Matlab, the entire project was completed in 3 months. The respondent liked Matlab, because it was possible to run scaled down versions on a workstation and the interpreted mode of Matlab made it possible to watch the computation in detail.

Using Matlab, the team was able to reduce the time to program the computational kernel: from 2-3 weeks to 3 days. In the process, however, a number of additional changes were made. First, rather than feeding the data from a single file server, filtering it and passing suitable subsets to the worker nodes, the Matlab version simply passed all the data to all the worker nodes. The filtering was then done by the worker nodes themselves. Second, the data was split up and distributed over nine file servers. This change both eliminated the I/O bottleneck and also MPI. The worker nodes simply opened files to get their data. In essence, the team simulated manually what a scalable parallel file system should do automatically. A special program later collected the files produced by the worker nodes and merged the data. The result was a dramatic speed-up. Instead of three weeks it took less than a day to process the data for a week.

Respondents suggested the following list of tools. Almost half of them (5) asserted that a global view language would improve productivity greatly. Parallel I/O, load balancing libraries, debuggers, and profilers were mentioned twice each, the rest once.

1. Programming languages with global arrays, such as Fortran 90, HPF, ZPL, or a global array toolkit (5 citations).
2. Scalable, parallel I/O (2 citations).
3. Libraries for load balancing (2 citations).
4. High-level parallel debuggers and profilers (CM-5's parallel debugger was mentioned as a good one.) (2 citations).
5. Low-level profilers or simulators to see what goes on in the processor for optimization purposes (cache misses, TLB misses, pipeline stalls) (2 citations).
6. Communication profilers. Presently, communication optimization is done by trial and error (1 citation).
7. Prettyprinters for the various Fortran versions (1).
8. Translator from Fortran 77 to Fortran 90 (1).
9. Smart source browsers that have static analysis capability, for instance for tracing variable use, checking initialization, indexing and pointers; highlighting dead variables and dead code, etc. (1).
10. Tools to simplify/rewrite/clean up formulas (1).
11. Automated regression testing (a la Junit) (1).
12. Comparative debuggers: suppose there are successive versions of a program. At a certain point, a bug is noticed. In which of the versions was the bug introduced? This could be solved with a technique called delta debugging (run the revealing test case on all versions automatically) [9] (1).
13. Distributed versions of basic data structures, such as lists, queues, priority queues, etc. (1).
14. A tool that visualizes the status of all processors in a time line (idle, running, I/O) (1).
15. For SMP-clusters: compilers should make sure shared memory communication is used where possible; message exchange only when going outside of the SMP node (1).
16. One-sided communication primitives (1).
17. Aids for deciding whether 32-bit accuracy is enough. Without it, everybody uses 64-Bit accuracy everywhere, doubling the cost in memory bandwidth and cache usage (and energy) (1).
18. Aids for handling erroneous input data. The massive data sets to be processed are never perfect. A faulty data record can lead to a program crash. Need a way to identify the faulty data record and skip it on the next run automatically (1)

## 4. Conclusions

In conclusion the most pressing needs we observed seem to be:

1. A program model based on a global view or virtual shared address space, combined with

compilers that produce efficient communication code. Although early experience with HPF compilers was disappointing, more recent work in HPF [7] and ZPL compilers [3] showed that compilers can produce communication code competitive with hand-written code.
2. Scalable, truly parallel I/O for handling massive data sets.
3. Libraries for load balancing and distributed versions of basic data structures.
4. Fancy parallel debuggers and profilers (at the levels of algorithm, communication operations, and processor internals).

## References

[1] H. Russel Bernard, Research Methods in Anthropology: Qualitative and Quantitative Approaches. Alta Mira Press, 1995.

[2] H. Russell Bernard (ed), Text Analysis, in Handbook of Methods in Cultural Anthropology, pp 613, Walnut Creek: Altamara Press, 1988.

[3] Bradford Chamberlain, Sung-Eun Choi, Steven Deitz, and Lawrence Snyder. The high-level language ZPL improves productivity and performance, Proc. IEEE International Workshop on Productivity and Performance in High-End Computing, 2004.

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

[5] W. Penn Handwerker and Danielle Wozniak. Sampling Strategies for the Collection of Anthropological Data: An Extension of Boaz's Answer to Galton's Problem. *Current Anthropology*, 38(5): 869-875, 1997.

[6] J. M. Morse. Designing Funded Qualitative Research. In Handbook of Qualitative Research. Norman K. Denzin and Y. S. Lincoln, eds., pp. 220-235. Thousand Oaks, CA: Sage Publications, 1994.

[7] Matthias M. Müller, Compiler-generated vector-based prefetching on architectures with distributed memory, High Performance Computing in Sciences and Engineering '01, pp 527-539, Springer Verlag, 2001.

[8] Susan Squires and B. Byrne, (eds), Creating Breakthrough Ideas, Westport, Bergin & Garvey, 2002.

[9] Andreas Zeller. Yesterday, my program worked. Today, it doesn't. Why? Proc. ESEC/FSE 99, Vol. 1687 of LNCS, Springer Verlag, pp 253-267, 1999.