An Empirically Derived Framework for Classifying Parallel Program Performance Tuning Problems

Anna Hondroudakis and Rob Procter
Department of Computer Science
Edinburgh University
Mayfield Road
Edinburgh EH9 3JZ
Scotland
Tel: +44 131 650 5177
E-mail: ah rnp@dcs.ed.ac.uk

ABSTRACT

The results of research into the use of parallel program performance tuning tools demonstrate that, despite the large number of tools described in the literature, very few of them are actually used on any scale within the parallel programming community.

Effective and efficient tuning is of great importance to parallel program development. In this paper we present the results of a study of the problems of tuning as identified by people for whom this task is a routine part of their work. A variety of techniques were employed in the investigation, including questionnaires, analysis of e-mail query logs and in-depth interviews.

We analyse the results to establish a framework for classifying tuning issues in terms of their characteristics and importance, and discuss its implications. Not least, a better understanding of what it is that makes tuning such a difficult task might lead to the improved design of tuning support tools.

Keywords
Performance problems, empirical studies, frameworks, tool design

1. INTRODUCTION

The quest for functionally correct application programs that take advantage of the power of the underlying hardware requires usable and efficient software tools and environments to assist with parallel program development and performance tuning. To this end, considerable effort and expense have been devoted to tuning tool design [4]. A plethora of tools are now available which provide support for inter alia performance data collection, analysis and filtering, visualisation and even automatic performance diagnosis. However, the results of surveys of tool use remain disappointing [5, 14]. Users complain that tuning tools are hard to learn and use, and do not provide them with the information they need.

Effective and efficient tuning is of great importance to the parallel programming community and understanding the problems which contribute to its difficulties would undoubtedly help to improve tuning tool design. Unfortunately, progress towards this goal has been hindered by the apparent lack of input from an important source: tuners themselves. By way of addressing this state of affairs, we have been conducting studies of tuning issues as perceived by tuners, using interviews, questionnaires and e-mail query logs.

Despite the evident relevance of understanding tuners' work to the design of effective tuning tools, very few studies (with notable exceptions) have addressed this issue beyond the context of the laboratory. Curtis's observations about the limitations of laboratory-based investigations addressed studies of sequential programming [2], but they are equally pertinent to the study of parallel programming problems.

In one early study of parallel programmers, Pancake found that almost 90% relied exclusively on PRINT statements instead of using a debugging tool [13]. In a larger, followup study, Pancake et al. [14] asked 448 participants of Supercomputing '93 whether they had used any computer-based tools during the stages of parallel program development, namely: fixing a serial program; writing a parallel program; improving the basic model; debugging; tuning performance and setting up program runs. Amongst other results, it was reported that 29% of informants had used no tool at all.

The study we present in this paper draws upon the experiences of a rather smaller number of parallel program developers. However, it goes beyond earlier studies in both the depth and breadth of its exploration of issues pertaining to performance tuning. Whereas Pancake's survey targeted parallel programming tool use, our study seeks to analyse the practice of tuning as it is shaped by the context of current tool provision and the wider concerns of parallel program development.

Our study was conducted in two parts. We begin by presenting and analysing the results of the first study, and propose a framework for classifying the issues raised. The framework is then used to analyse the results of a second study which was based upon queries received by a parallel programming user support service. Finally, we discuss how the framework
can be used to inform the design of tuning tools.

2. PHENOMENOLOGICAL STUDIES OF PRO-GRAMMING

The starting point for our work has been Eisenstadt’s study of the phenomenology of sequential code debugging, i.e., how debugging is performed by real programmers [3]. Eisenstadt elicited debugging anecdotes, descriptions of bugs and of bug-fixing strategies from developers of academic and commercial software through e-mail and news bulletin boards. His analysis focused on the reasons bugs are difficult to find, how they were found and their causes. We summarise below those aspects with greatest bearing on our own study.

Eisenstadt’s analysis of responses on why bugs are hard to find revealed four distinct categories. In order of decreasing frequency of citation, these were as follows.

Cause/Effect Chasm Often the manifestation of a program bug is far removed in space and/or time from its cause. It may not be easy to find the root of the problem when the evidence of the bug cannot be related to the program source code. Specific instances in Eisenstadt’s study included bugs that are intermittent, inconsistent, or infrequent, and those with “...too many degrees of freedom” i.e., bugs with so many possible contributory factors that systematic testing is problematic. About one third of Eisenstadt’s informants cited the Cause/Effect Chasm as the reason why a bug was hard to find.

Inadequate Tools Eisenstadt’s informants reported that bugs often disappeared when debugging tools were switched on, a phenomenon known as the probe effect. Other instances of this category included where debugging tools could not be used for various reasons, e.g., because of memory constraints. Inadequate Tools was a significant category in Eisenstadt’s study. Together with the Cause/Effect Chasm, it accounted for 53% of all responses.

Faulty Assumption/Model Being able to call upon a conceptual model that explains the operation of a system software or hardware component reduces significantly the search space for the cause of a bug; equally it can exacerbate the difficulties if it is faulty. The Faulty Assumption/Model category accounted for about 12% of Eisenstadt’s informants’ responses.

Spaghetti Code Eisenstadt found a 100% correlation between complaints that a code was too messy to debug and it being written by another person. The Spaghetti (unstructured) Code category accounted for less than 3% of responses, however.

Eisenstadt also extracted four distinct techniques for finding bugs from his data. In order of decreasing frequency of citation, these were as follows.

Gather data This category encompasses experimentation techniques such as use of printifs and inserting breakpoints in the program execution. These techniques were nominated by over 50% of Eisenstadt’s informants.

Inspection This is Eisenstadt’s hybrid term, an abstraction for a variety of techniques, including “inspection”, “simulation” and “speculation”. One or more of such techniques were cited by approximately 25% of informants.

Expert help About 10% of Eisenstadt’s informants resorted to consulting with a recognised expert.

Controlled experiments Some informants reported the use of controlled experiments in an attempt to investigate what the cause of a bug may be. In principle, these would begin with a hypothesis based on a conceptual model of a system component (software or hardware). If this model is accurate, then the programmer should quickly find the cause of the bug and eliminate it. If the model is wrong then more experimentation is needed to find the bug, and the model gets updated or extended. This technique accounted for less than 10% of informants’ responses.

3. THE FIRST STUDY

Fifty two tuners participated in the first part of our investigation. Eighteen informants worked for university establishments and thirty four for commercial companies. Of the latter group, less than one third reported that they had had
any training. A profile of informants, including years of experience, is shown in Figure 1.

Twenty six tuners responded to questionnaires made available through relevant news bulletin boards and the World Wide Web. In addition, and to enable issues raised by questionnaire respondents to be explored in depth, twenty six tuners working for a number of UK and European organisations and institutions were interviewed.

Participants were asked to identify the most difficult aspects of tuning, describe their own particular practices, and for information about their background and experience. They were also asked to provide illustrations of their work via “war stories” — i.e., difficult or notable problems they had encountered. A complete record of the responses can be found in [5].

4. A FRAMEWORK FOR THE STUDY OF TUNING PROBLEMS

As can be seen from Table 1, our tuning problem framework is an expanded version of Eisenstadt’s debugging framework. To explain the results, we turn first (in order of decreasing frequency of citation) to those findings which can be categorised within Eisenstadt’s framework. Finally, we describe findings which did not fit within the original framework.

4.1 Cause/Effect Chasm

In tuning, the Cause/Effect Chasm is a measure of the gap between knowing that performance is sub-optimal and knowing what part of the code is responsible — the problem of source code reference. Unlike sequential code debugging, performance problems are seldom intermittent or infrequent, and so are relatively easy to observe. Our informants emphasised that although it is relatively easy to spot the symptoms of poor performance, it is often very difficult to find the location of its cause. A complicating factor here may be the nature of code optimisations performed by the compiler:

“Another hard part is figuring out exactly what gives the performance improvement when some optimisations are performed. For example, when you rearrange basic blocks at compile time based on heuristics, performance improvement comes from better cache locality and from better branch prediction. The hard part is figuring out exactly what percentage of improvement came from which change. Knowing where to fix and knowing the solution for best performance are two separate issues. Knowing the solution comes from experience and it can take one to many tunings.”

Informant 69

Bridging the Cause/Effect Chasm begins with the collection of program trace data, the volume of which can be very large, particularly in the case of massively parallel systems. Relating the low-level account of program behaviour trace data provides is difficult:

“...So, my tool gave hundreds of different performance metrics about the processors but we would not see where these metrics applied.”

Informant 59

Some responses indicated that part of the difficulty can be attributed to misleading information given by tuning tools (see Inadequate Tools). The simplest examples were cases where informants had to deduce the value of a particular performance metric from other ones that were provided by the tuning tool. The following quote exemplifies problems encountered when the abstractions provided by tools do not match their users’ requirements:

“We have got a tool like paragraph and I think ...it gives you too much information, it is showing what is being going, but in terms of where the time is spent it is not clear where the time is spent. Cray have a tool that actually shows you where the time is spent, but it is cumulated over all processors, so in some sense it is too much condensed information.”

Informant 47

Under the Cause/Effect Chasm category, Eisenstadt included problems associated with too many degrees of freedom. As an example, of this, he cited the uncertainty introduced by changes in program environments such as compilers. Our informants also provided examples of this type, but we argue that the qualitative differences between the circumstances of serial and parallel program development are such that these factors merit a category of their own, which we have called Change. We will return to this category later.

4.2 Faulty Assumption/Model

Understanding the detailed behaviour of parallel architectures and system software is not something that the average tuner finds easy to do:

“Getting to optimise the serial program can be hard enough because you have to know the things about the RISC architecture that you don’t want to know. I know them by reading the BYTE and listening to colleagues.”

Informant 44

<table>
<thead>
<tr>
<th>Category</th>
<th>Occurrences</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cause/Effect Chasm</td>
<td>25</td>
</tr>
<tr>
<td>Faulty Assumption/Model</td>
<td>17</td>
</tr>
<tr>
<td>Inadequate Tools</td>
<td>11</td>
</tr>
<tr>
<td>Other People’s Code</td>
<td>5</td>
</tr>
<tr>
<td>Repetitive Nature</td>
<td>24</td>
</tr>
<tr>
<td>Change</td>
<td>4</td>
</tr>
</tbody>
</table>

Table 1: Informants’ responses classified with respect to the tuning problem framework.
Compounding the problem is the fact that there are a large variety of high performance parallel computer architectures. The architecture of the target machine, together with available compilers, determines the options the programmer has for writing the parallel program:

“The data consisted of three materials whose computational requirements on machine A were rated 1:2:128 respectively. The rate of processing required for each type of material in the data set was not the same on machine B. The reason was that the processing of the most expensive material required a lot of floating point operations which were cheap on machine B whereas the processing of [other] materials involved a lot of memory copying which is relatively expensive on machine B. When the 1:2:128 rate was used for the decomposition strategy on machine B, the performance gain decreased.” Informant 51

Once the programming model is selected, the programmer then has to consider how data and control can be best distributed in the program. The programmer has to be aware of low level details which affect the performance of the program:

“…and the latest architectures are RISC which I am not familiar with, for example, with vector architectures you know what to do, you know with the inner loops and things. In terms of cache use, I am little bit lost.” Informant 47

Not surprisingly, porting a program to another machine with a different architecture is regarded as extremely difficult. One reason is that it increases the likelihood that details of models get confused or forgotten. Sometimes, the way to overcome performance problems on a new architecture is to redesign the application:

“It is difficult to try to keep changes clean and portable. I assumed that shared variables were implemented by replication [and that] read operations were fast and write operations involved one or more messages. As an aside, since the network was an ethernet, I also wanted to have different messages sent at different times to avoid congestion. So, I tried to keep the number of write operations as small as possible. On a point to point network …this should be implemented differently.” Informant 6

Our informants seemed more comfortable when reasoning about program performance at the more concrete levels afforded by the notion of parameters. These may be thought of as dimensioning the performance space in terms of the practical measures that the tuner might take to improve performance. These might range from decomposition strategy on the one hand, through to compiler optimisation flags on the other. Often the effects of these parameters are inter-related and only partially understood. Our informants stated that their problems typically involved between less than five to more than twenty parameters. However, ninety percent of responses were in the range 1–10, and 50% in the range 5–10.

Tuners may try to predict what combinations of parameter values will lead to improved performance, but this is not easy with the numbers of parameters involved:

“The domino effect of changes of parameters with respect to performance is quite challenging.” Informant 5

Our informants indicated that they often found it necessary to fall back on the use of trial and error in order to arrive at a satisfactory model of the interactions between the parameters. Tuning then becomes a heuristic search necessitating the evaluation and comparison of the results of a succession of “experiments” [6]. We will address this aspect of tuning later.

4.3 Inadequate Tools

Earlier studies leave little doubt that the impact of inadequate tools have on the parallel software development process is significant, especially when factors such as the immaturity of much parallel system software are taken into account (see Change). Our informants were asked whether they had ever used any performance analysis tool with their parallel code. The responses confirmed earlier studies: despite the large number of tuning tools that now exist, approximately 50% reported that there was no tool available to them. Approximately 10% reported that they had built their own tools, and only 40% of informants had used an externally developed performance tool.

Our informants provided numerous examples of how tools may mislead the way in which performance information is presented:

“The problem with apprentice is that if you have a particular routine it will tell you how much time you waste in memory access… if the routine is very inefficient in its numerical operations, memory access time loss gets hidden by the inefficiency of the routine. If you rewrite the routine to optimise it …the bottleneck shifts to memory access, which will result in apprentice telling you “your routine is wasting 80% of its time” …So, apprentice can be misleading, if you use it for one routine, you might get the wrong idea.” Informant 44

4.4 Other People’s Code

There are a number of implications for the way tuning is performed when the code is written by someone else, or is written in the context of multi-people, multi-organisation development projects. Interviews with informants revealed a number of distinct patterns in large scale project organisation and structure. In some cases, tuning experts formed a separate group within the project team with responsibility solely for the application’s efficient parallelisation. Alternatively, optimisation work might be contracted out to experts working for consultancy companies. Indeed, it was not uncommon for the project as a whole to be distributed teams in a number of different organisations.

Tuning may involve many changes to the program, necessitating frequent checks for correctness. Communication overheads can grow rapidly in any kind of large scale project, but parallel program development seems to impose particularly demanding requirements on team members with respect to documenting, conveying and justifying decisions which separately — and together — affect program performance. Often, achieving a clean division of responsibilities seems difficult:
"I go mad. I tried to keep up with the updates of everyone. I got three updates every day. I tried to keep as many common routines — the matrix construction, or the time integration, or the boundary value determination routines — as possible but daily changes in the serial program structure and parameter lists made this impossible. Everything each of them are doing affects me."

Informant 44

Finding tools that everyone can (or wants to) use may also be a problem:

"The way we are working here today is from this point of view very haphazard. We have a lot of versions, and this is a significant problem. Although we have a version control system, the guys in the parallel world don't use it because it is installed on a different system from the one they work on... so we are at the situation where there are lots of tar files lying there and it is slightly difficult to follow what you have."

Informant 52

4.5 Repetitive Nature

Without reliable predictive performance models, tuners must consider many reasons for performance loss, the many locations in the program where bottlenecks may occur, and investigate the effects of different combinations of parameter values:

"Tuning itself is kind of game, finding strategy, slowly progressing inside a foreign wild programming world... good tools are the weapons you need to survive."

Informant 7

In these circumstances, tuning is largely experimental, heuristic and often opportunistic. It may take a long time and may be suspended and resumed numerous times. Typically, major bottlenecks are addressed first, but tuners may come back to a specific problem if in the meantime some more knowledge is acquired or if there is time. Performance can be improved to different degrees. Tuning can be repeated until specific performance goals are met, or may be constrained by time pressure due to deadlines, or the finite time available on the target machine. Time constraints may have a major impact on the strategy adopted:

"We had to make the thing running for the demo. It changed the way we worked because when you are working with deadlines, the first thing you want to do is get the thing to run in a way you are able to live with. And then, when you do tuning you do it in a very careful way so as not to blow something really important. Maybe, if there were no deadlines, you would use more time to tune during development."

Informant 42

Our informants' responses to the question of tuning duration ranged from "less than a day" to "months". However, 94% of these replies were "several days" or longer, and 50% were "several weeks" or longer.

From our informants' accounts, it is possible to distinguish two different types of experimentation:

- Initial runs can help tuners form a model of the interaction of the various performance parameters. A special case in this class are the runs which verify whether a particular problem e.g., load balance, exists.
- Runs of small pieces of code which test the effectiveness of alternative solutions. Before one of the solutions is adopted, it is compared with the others in a small piece of code in order to allow easy and cheap (in terms of tuner's and machine's time) experimentation.

Once progress has been made in determining the cause of the performance problem, tuners will carry out further program runs to validate hypotheses and assess the degree of improvement achieved. Three classes of runs could be distinguished from informants' accounts:

- Runs which check program correctness and the performance gain after a change in the code. When the scalability of a change is examined, runs are repeated on as many available sets of processors as possible. As experience in tuning increases, these runs test progressively more changes at any one time.
- After a version "freezes", the program is run on many processor sets in order to show the behaviour of the program as the input data changes, or as the problem size changes.
- Runs where two versions of the code are compared under the same hardware or system software.

This process of tuning experimentation, validation and assessment is highly repetitive and (as we have seen above) may be prolonged. In such circumstances, keeping an accurate record is vital for the effective management of the work. Tuners may utilize records of tuning experiments in a number of ways. For example, they may wish to keep for later reference all or part of the records of several experiments in order to document [7]:

1. the problem under investigation and its manifestation in the trace data,
2. the parameter values, and the path through the performance space,
3. performance metrics associated with each set of parameter values, and
4. the state of the program when tuning was concluded.

At any time during the course of tuning, tuners may need to consult experiment records in order to find out:

1. the parameter values that produced the best overall performance so far,
2. the parameter values that produced the best value for one specific performance metric, and
3. whether the current performance is better than that achieved with a specific set of parameter values.

A number of informants made comments which pointed to the dangers of skimping on task documentation:

"On pieces of paper. They were not organised. I often had to repeat the experiments to carry on from where I stopped."

Informant 15
We asked our informants to nominate the reasons why they documented their work. The results are listed in Table 2. The two most cited reasons reflect the repetitive nature of the task. Some informants also stressed the value of accumulating tuning case histories across projects “to use as a learning aid and for future performance sessions”:

“I can see what changes made the biggest differences hopefully to help me the next time I tune a parallel program.”  

Informant 21

Other comments reflected issues arising from the often collaborative nature of parallel software development, and the importance of sharing knowledge amongst project members (see Other People’s Code). The value of accumulating and sharing the kinds of task-relevant information not found in manuals was stressed. Such informal and tacit knowledge — ‘tricks of the trade’ — which is rooted in practice and experience, is always difficult to capture [19], and tuning is no exception:

“But other problems we find are the continuous documentation of problems, hints, tricks, things that people have found in use and have it up here [points to his head] and they never wrote them down so that everyone else can use. Yes, the biggest problem is with the little tricks and hints, ways around the problems.”  Informant 48

4.6 Change

Our informants reported that it is common to have to tune programs without being able to rely upon advanced compiler and run-time support. They also have to attempt to do their work when hardware and system software are still unstable and immature. Inconsistencies may arise when different software components get improved at different points in time, as in the following case:

“The application program ARPS had been performing a nearest neighbour computation in one part of the code which CMAX [a parallelising pre-processor] translated into an expression containing a number of EOSHIFTS. Unfortunately, the latest version of CSHIFT in the CM Fortran run time library is currently more optimised than EOSHIFT.”  

Informant 17

4.7 Discussion

The majority of the categories in our tuning problem framework match those of Eisenstadt. Of this original group, however, we find that the category Faulty Assumption/Model is now the second most commonly occurring. The quick and efficient solution of performance problems requires a lot of knowledge about the specifics of the underlying machine. Informants’ responses confirmed that this presented a major problem for many.

Our study confirms that tuning tools are often inadequate or poorly matched to users’ needs, though the category Inadequate Tools only ranks fourth in our revised framework (or third using Eisenstadt’s original framework). In fact, the ranking of this category may be misleading. One reason is that about 50% of our informants reported that there were no suitable tools available to them, thereby effectively reducing the size of the reporting group by half. Another reason is to do with the problems of classifying informants’ responses. Inevitably, there were cases which could be classified under more than one heading. It was evident, for example, that some instances of the Cause/Effect Chasm might be attributable — at least in part — to tool deficiencies. Where multiple factors might be implicated in problems, we have counted only the one which we judged to be the most significant.

When asked to specify tool requirements, approximately half of informants’ replies were suggestions for general features and 30% for abstraction mechanisms to control the visualisation of performance data and to provide appropriate bridging abstractions — i.e. ways in which tuners might close the Cause/Effect Chasm between performance data and program behaviour (see Figure 3). This provides corroboration of the importance of this particular issue.

Almost all of our informants emphasised that parallel computing is a particularly dynamic sector which changes constantly. One of the implications of Change is that documentation may quickly get out of date. Another is that performance problem identification and solution inevitably becomes very context specific. Local expertise is therefore at a premium.

The categories of problems described so far are pertinent to understanding the cognitive issues associated with tuning in the small [6]. The remaining reflect issues related to tuning in the large [7].

Parallel program development projects are often large in scale and involve collaboration with many others [5]. In our framework, the category Other People’s Code substitutes for Eisenstadt’s Spaghetti Code. We did this in order to emphasise that the origins of Other People’s Code lie in the large scale nature of projects, rather than in the idiosyncrasies of individual’s programming styles.
There is a superficial resemblance between the *gather data* technique employed in serial program debugging and tuners' reliance upon experimentation. However, the differences are such that for the latter, experimentation becomes part of the problem, rather than a trusted problem solving technique. It is for this reason that we have added the category *Repetitive Nature* to the tuning problem framework. It emphasises the overheads of conducting trial and error experimentation with program performance when the work is repetitious and extends over long periods of time. An even more compelling reason for introducing a new category, however, is the frequency with which this problem occurs in tuners' accounts: it is second only to the *Cause/Effect Chasm*. It seems clear that documenting the tuning process and its outcomes is itself a major task requiring a significant amount of effort.

5. THE SECOND STUDY

The Edinburgh Parallel Computing Centre (EPCC) has a long established on-line, e-mail support service for local and remote users of its high performance computing facilities. Users' queries are usually answered within 24 hours. Queries and answers are kept in an on-line archive. All the performance related queries submitted October 1993 and March 1996 were selected from the archive with EPCC's permission and studied. The total number of such queries was eighty eight and they had been submitted by seventy five users. The identity of the senders was coded to preserve confidentiality.

The queries and experts' responses were initially analysed and classified in four general headings which were chosen to reflect their basic form. Table 3 lists the classes and their frequency of occurrence. Selected queries are summarised in Figure 2 to illustrate the kinds of issues for which users sought expert help.

<table>
<thead>
<tr>
<th>Specific Class</th>
<th>Occurrences</th>
</tr>
</thead>
<tbody>
<tr>
<td>Specific Advice</td>
<td>44</td>
</tr>
<tr>
<td>Why Is That Happening?</td>
<td>21</td>
</tr>
<tr>
<td>General Advice</td>
<td>20</td>
</tr>
<tr>
<td>Inadequate Tools</td>
<td>3</td>
</tr>
</tbody>
</table>

Table 3: Initial classification of on-line support service queries.

5.1 Specific Advice

In this class we included queries where users had identified fairly precisely what it was they needed to know. The issues raised ranged from high to low level. For example, some queries sought reassurance that the technique being used could bring about a performance gain. Often, high level queries revealed faulty assumptions about factors that affected performance on a particular machine or programming model. Instances of low level queries included requests for typical values of various metrics which could have an impact to their code, such as bandwidth and latency rates of various communication calls.

A significant number of low level queries concerned compiler issues. For example, some questioned the relevance of old performance improving techniques in the light of compiler changes, others asked what performance optimising flag options could be used with the current compiler, or for explanations of how particular compiler directives worked. Such queries reflect the impact of compiler changes on performance. Code optimisation techniques have to be reviewed in the light of new compiler versions. Even when documentation is readily available and up to date, it is difficult to know to which extent old techniques are still relevant (Q19, Figure 2). The situation is further complicated by inconsistencies in the run-time environment: in one case part of the system software was not optimised for the changes introduced by a new version of the compiler (Q1, Figure 2).

Some of the reported performance problems manifested themselves as run-time problems or bugs. Support service experts answered these questions with advice on how a performance improving technique would remove the problem. In one case, a user asked how to do check-pointing because the execution time of his program was longer than the maximum allowable run time. The expert made a number of performance improving suggestions which alleviated the need for check-pointing.

Overall, queries in this class typified problems associated with the *Faulty Assumption/Model and Change* categories.

5.2 Why Is That Happening?

Queries in this class typically sought an explanation for mystifying program behaviour (Q20, Figure 2). In one case, a user was concerned about a parallel code whose timings were drastically different from one day to another. The user had identified at least two possible causes, but was unable to determine from the evidence which one was relevant:

"I have faced recently a quite weird problem on the T3D and wonder whether somebody else ever noticed the same... the time per iteration has CHANGED for the SAME job, namely, now it is getting slower by a factor between 2 and 4! Similar behaviour I have observed with another job which used 64 PEs. In this case, it was slowed down by a factor of 4-6 for all runs performed after 2nd of November. So, I cannot say whether there is any correlation with the number of PEs that would mean that this is a communication problem... In all these cases, the same source code has been used. But I have RECOMPILLED the code after 1st of November with the same level of optimisation, so that this might be some compilation problem."

Q19

Other instances in this class confirmed that programmers may be unaware of the way instrumentation may affect performance (Q10a, Figure 2) as in the case where a user complained of different results when the code was compiled with
the apprentice*flag enabled [18].

Overall, the queries in this class were typical of those associated with Cause/Effect Chasm problems.

5.3 General Advice

The queries in this class sought general advice on efficient parallel programming (Q2, Q12, Figure 2). Some users simply wanted to know where to look for information on the most relevant optimisation techniques (Q3, Figure 2). Others asked for information about efficient data and work distribution techniques:

"We would appreciate any information on efficient data and work distribution techniques for large arrays."  

Q7

In some cases, a piece of parallel code would be submitted along with the query:

"This is my code. It only achieves 6.6 Mflops. Could you suggest ways in which it might be improved?"  

Q5

Some queries revealed that performance expectations may simply be too high:

"Could you give us some example codes that managed to perform up to 130 Mflops per second on the T3D?"  

Q32

Queries in this class suggested a general lack of understanding of performance issues and hence of strategic knowledge for tackling performance problems. They may be considered to be extreme cases of Faulty Assumption/Model problems.

5.4 Inadequate Tools

There were a very few specific queries which might be directly attributable to Inadequate Tools. They included complaints about not being able to interpret the performance information presented by tools and comments about the incorrectness of on-line documentation. The former tended to be closely associated with a lack of detailed understanding of code execution:

"I have a question concerned with the apprentice tool. In the COSTS window, the bar chart always shows that Integer Adds is the dominant operation in the code. However, I expect that the code should perform floating point operations most of the time. Is there a conversion from floating point operation to integer operation inside the T3D? This will affect my judgement about how fast the code runs."  

Q51

In other cases, the choice of terminology in the presentation of performance information was inappropriate for inexperienced users:

"I've used apprentice to see the performance of my code. The observations for my code are:

Combined losses due to single instruction issue, instruction cache and data cache activity are estimated to be 8.33% of the measured time.

Greatest aggregate improvement may be gained by improving children of EVOLUTION STUDY.

Synchronisation or work construct which offers the greatest potential improvement is STMTS@26.

With these comments, I don't understand if my program is good or not. I suppose it means that 8.33% of the time only one process works. But what I'd like to know is how to improve my code. What is "children of a subroutine"? And what is the work construct?"  

Q45

Although it is impossible to verify from the query/answer log alone, we suspect that tool deficiencies would be implicated in a significant number of the other queries.

5.5 Discussion

Approximately 46% 2 of queries could be attributed to a basic lack of basic knowledge of tuning and of the factors that affect performance. Half of the queries (50%) sought advice about the use of specific performance determining parameters, or about how to improve a particular part of the code. This suggests that these programmers know what would improve the performance of their program, but they do not have enough information to implement it successfully (Q3, Figure 2). In many of these cases, people were caught out by changes in the programming environment of which they were unaware.

After the initial analysis, the queries were re-classified from the perspective of the tuning problem framework. The results are shown in Table 4. Some differences are evident when these are compared with the results of the first study. We should note, however, that methodological differences between the two studies make any simple comparison difficult. Most importantly, we may reasonably assume that queries submitted to the support service are necessarily representative of more difficult problems than those reported by participants of our first study: they concern problems that users were not able to solve for themselves. This may

<table>
<thead>
<tr>
<th>Query Class</th>
<th>Occurrences</th>
</tr>
</thead>
<tbody>
<tr>
<td>Faulty Assumption/Model</td>
<td>47</td>
</tr>
<tr>
<td>Change</td>
<td>21</td>
</tr>
<tr>
<td>Cause/Effect Chasm</td>
<td>17</td>
</tr>
<tr>
<td>Inadequate Tools</td>
<td>3</td>
</tr>
</tbody>
</table>

Table 4: Support service queries re-classified within the tuning problem framework.

2 The occurrences of Why Is That Happening? and General Advice were added together.
be the explanation for the absence of problems in the Repetitive Nature category in the second study.

Both Faulty Assumption/Model and Change problems rank higher in the order than before. As the queries were anonymised, we have no means of confirming whether the increased ranking of the former could be attributed to users’ inexperience. The increased frequency of problems attributable to Change may be an inevitable overhead of being a remote user of high performance computing facilities. Remote users may simply find it more difficult to keep up with changes in the system environment.

It will come as no great surprise to learn that much of the information sought in the queries we analysed could be found in on-line documentation. There is no reason to suppose that high performance computer users are any different from other computer users in their reluctance to consult manuals except as a last resort.

6. IMPLICATIONS FOR TOOL DESIGN

By identifying and characterising tuning problems as experienced by tuners, the framework we have presented provides a way of determining the focus for the future development of performance tuning tools. Since our concern throughout has been to understand problems from tuners’ perspectives, however, we also conducted a survey of requirements among the participants of our first study. The results are summarised in Figure 3.

We will first consider the implications of our studies from the point of view of tuning in the small. There are already many tools which provide what are often sophisticated techniques for the visualisation of performance data such as execution traces (e.g., [15, 16]). The persistence of the Cause/Effect Chasm in our studies suggests, however, that such techniques often fail to provide meaningful information about the nature of the problem. To fix performance problems, tuners must relate the low-level account of program behaviour provided by the trace data to the high-level representation of program behaviour contained within the source code. Among the requirements put forward by tuners (see Figure 3) were support for:

- user definable events that are aggregates of other lower level ones, and
- better techniques for isolating events of interest.

More detailed research is needed to understand the nature of Cause/Effect Chasm problems. We do not imagine, however, that there is a single ideal visualisation technique, nor do we believe that visualisation of trace data to be the only issue: solving Cause/Effect Chasm problems involves establishing relationships between multiple sources of data, yet most tools fail to provide adequate data integration facilities [8]. Another point emphasised by our informants is that visualisation needs change from problem to problem, and from user to user: visualisation techniques need to be flexible and easy adapted to match.

No visualisation technique, however sophisticated, is going to eliminate the need for the exercise of interpretive skills by the user. Where these are lacking, as our studies suggest they often are, better support is required to help less experienced users learn these skills more quickly. Better — indeed any — training would help, but the nature of these skills is probably such that “learning by doing” would be more effective. To this end, the availability of a repository of tuning case histories and exemplar problems drawn from the local development context could be very useful resource. For example, users could search it for matches with their own problems. It would also provide a suitable context for making other forms of documentation more problem-oriented, an approach which is more likely to meet users’ needs [1].

One approach to tackling Faulty Assumption/Model problems is to employ decision support tools to provide background knowledge and hints for less experienced tuners. Decision support is now available in a number of tools (e.g., [12]). Our studies indicate that more attention needs to be paid to dialogue design for decision support, especially to take into account the fact that many potential users may be inexperienced.

We would also argue that the effects of Change are such that any decision support must be adapted to the needs of the local development environment. As we have seen, one effect is that every development context is different, and so performance problems and solutions are usually quite specific to local hardware and software configurations. Adapting decision support to local circumstances could be achieved through integration with local tuning case histories and exemplars.

Whether with or without decision support, tuning case histories could be used to contrast performance before and after optimisation. Exemplars could be used both to explore the relationships between performance determining parameters, and as sources of suggestions for problem solving tactics. In these ways, the expertise of experienced tuners could be made available for re-use by others [9].

We now turn to the implications of our studies for tuning in the large. Tuning documentation has many similarities with a technique for software documentation known as design rationale [11]. Traditionally, project documentation captures design decisions, but doesn’t explain why they were taken. In contrast, the design rationale approach encourages clear documentation of the reasoning underlying design decisions. This is an important aid to comprehension by other members of the software team. In the same way, the documentation of tuning should provide a rationale of why a particular program configuration produces the best performance. In this way, some of the problems stemming from the Repetitive Nature of tuning could be tackled. However, documenting practices of this kind clearly demand good support tools.

Any of the many computer-based tools now available to support software documentation (e.g., [10, 17]) might find application within parallel software development. We believe, however, that there are significant qualitative differences in the documentation needs of parallel software development, which reflect both the experimental, iterative nature of tuning and the large volumes of information (e.g., traces) it generates. To meet the requirements raised by the problems of Repetitive Nature and Other People’s Code, we have proposed the concept of the Tuner’s Workbench [7]. The use of such a tool would also address many of the overheads implied by the creation and management of a tuning repository.
Finally, a major problem with current tuning tools is that they are often remain local to the site of their development. More than half of our informants answered that they have not used any externally developed tuning tool. Better communication is needed between tool developers, tool users and vendors of parallel systems so that the tools of tool development efforts can be shared among the wider parallel software development community.

7. ACKNOWLEDGEMENTS

We would like to thank all those who volunteered to take part in the studies, and EPCC for their permission to examine their query archives. Finally, we would like to thank the reviewers for their helpful comments.

8. REFERENCES


<table>
<thead>
<tr>
<th>ID</th>
<th>Context</th>
<th>Knows</th>
<th>Does not know</th>
</tr>
</thead>
<tbody>
<tr>
<td>Q1</td>
<td>Information sought on how compiler works after compiler change</td>
<td>Array sizes affect performance</td>
<td>In the light of the new compiler how much and in what way array sizes affect performance Some of the system libraries are optimised for the old compiler</td>
</tr>
<tr>
<td>Q2</td>
<td>Advice on general optimisation techniques for improving IO</td>
<td>Which part of the code needs improvement</td>
<td>Anything about how IO is performed and what the factors are that can affect performance</td>
</tr>
<tr>
<td>Q3</td>
<td>Documentation on alternative communication library</td>
<td>Existence of libraries with optimised maths functions</td>
<td>How to access information on the optimised library</td>
</tr>
<tr>
<td>Q5</td>
<td>Code is sent to be optimised</td>
<td></td>
<td>Anything about options within the performance model that have a different impact on performance how the compiler distributes arrays and executes directives. Techniques that can be used to optimise performance</td>
</tr>
<tr>
<td>Q8</td>
<td>Problem with opening too many files</td>
<td>Knows the problem has to do with the large number of files</td>
<td>The reason why the files cannot be opened simultaneously, how IO is done, technique to sequentialise the opening of the files</td>
</tr>
<tr>
<td>Q10a</td>
<td>Apprentice distorts timings</td>
<td></td>
<td>Reason apprentice distorts timings, how instrumentation is done. What the impact on performance is.</td>
</tr>
<tr>
<td>Q12</td>
<td>Tool showed where time was spent</td>
<td>Where time is spent</td>
<td>Basic issues such as what affects performance and what techniques could be applied in that particular case</td>
</tr>
<tr>
<td>Q14</td>
<td>Picking wrong documentation on the compiler</td>
<td>Optimisation flags exist which can be used to improve performance</td>
<td>That two compilers exist one for the front end and another for the mpp system. How to invoke information on the one for the mpp system</td>
</tr>
<tr>
<td>Q19</td>
<td>Timings of the same code were longer after a specific date</td>
<td>Suspects a hardware or compiler problem</td>
<td>When the compiler changes, how much and why a compiler change can affect the code, how much a hardware upgrade or a hardware error can affect the program</td>
</tr>
<tr>
<td>Q20</td>
<td>Mathematical functions perform poorly compared to their performance on other machines</td>
<td>Knows that they perform badly</td>
<td>The design of the processor chip is the reason for the poor performance of the maths functions</td>
</tr>
<tr>
<td>Q23</td>
<td>How much code performance depends on the speed of the front end</td>
<td>Speed of the front-end machine affects performance</td>
<td>Relation between the front end and the back-end what affects the speed of the code and how the front end interacts with the back-end (CM)</td>
</tr>
<tr>
<td>Q36</td>
<td>Poor performing code compared to the same code running on other machines</td>
<td>Knows where the problem is</td>
<td>Too high expectations from machine based on peak performance advertised by vendors. What aspects of the machine design affect performance</td>
</tr>
</tbody>
</table>

Figure 2: Summary of selected queries from the second study.
<table>
<thead>
<tr>
<th><strong>Tool features</strong></th>
<th><strong>User interface</strong></th>
<th><strong>Reliable tools</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>profiling output for each processor</td>
<td>graphical interface to the profiler</td>
<td>minimum possible perturbation</td>
</tr>
<tr>
<td>show idle and busy times on processors</td>
<td>better graphics to show overall performance</td>
<td>link with compiler to predict performance</td>
</tr>
<tr>
<td>and show the reasons for them</td>
<td>easy to use tools</td>
<td>handle big trace files</td>
</tr>
<tr>
<td>see what is going on in message passing library</td>
<td>graphs for statistics</td>
<td>reliable profiling info</td>
</tr>
<tr>
<td>have profiling for an incomplete code</td>
<td>better help in apprentice</td>
<td>reliable better quality tools</td>
</tr>
<tr>
<td>play the run back and forth</td>
<td>easy to understand displays</td>
<td></td>
</tr>
<tr>
<td>measure what program is doing on each processor</td>
<td>better suggestions</td>
<td></td>
</tr>
<tr>
<td>statistics eg. minimums and maximums</td>
<td></td>
<td></td>
</tr>
<tr>
<td>communication graphs in apprentice</td>
<td></td>
<td></td>
</tr>
<tr>
<td>gantt chart with arrows to show communications</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ratio of computation/communication</td>
<td></td>
<td></td>
</tr>
<tr>
<td>frequency of function calls</td>
<td></td>
<td></td>
</tr>
<tr>
<td>communication statistics</td>
<td></td>
<td></td>
</tr>
<tr>
<td>call graph</td>
<td></td>
<td></td>
</tr>
<tr>
<td>a 2-d map of what is processor is doing</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><strong>Abstractions</strong></th>
<th><strong>Tuning management</strong></th>
<th><strong>Abstractions</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>allow analyst to relate different views of the data</td>
<td>managing versions in multi-person project development</td>
<td>visualise user-defined and higher level events</td>
</tr>
<tr>
<td>source code reference</td>
<td>scalability analysis</td>
<td>print only the relevant profiling info (not getting thousands of lines)</td>
</tr>
<tr>
<td>look at summaries and then look at things in more detail</td>
<td>compare different runs</td>
<td>info about outliers and representative processors</td>
</tr>
<tr>
<td>click on a communication and see the line of code that invokes the communication</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>