Reusing sequential software in a distributed environment

To cite this article: A Bartoli et al 1995 Distrib. Syst. Engng. 2 2

View the article online for updates and enhancements.

Related content
- Understanding the behaviour of distributed applications through reverse engineering
  T H Kunz and J P Black
- Configuring scientific applications in a heterogeneous distributed system
  Patrick T Homer and Richard D Schlichting
- An overview of real-time ANSAware 1.0
  Guangxing Li
Reusing sequential software in a distributed environment

A Bartoli, G Dini, M Luise, G Pazzaglia, C A Prete and N A D'Andrea

Dipartimento di Ingegneria dell'Informazione, Università di Pisa, Via Diotisalvi 2, 56126, Pisa, Italy

Abstract. In this paper we present and discuss a real experience of reusing sequential software in a parallel and physically distributed computing environment. Specifically, we have combined the functionalities of two existing systems previously developed at our Department. One, Tracs, is a programming environment designed to facilitate the design of distributed applications. Applications are constructed out of basic design components that have a well defined interface and are context-independent. This not only enhances structuring and flexibility, but also embeds a great potential for component reuse. Tracs also has the ability to generate automatically implementation details of certain design components. That is, the user may implement only part of certain components and delegate the duty of completing them to the environment itself. As a special case of this feature, for certain styles of process interactions users may provide code that is fully sequential and the environment automatically generates the code for inter-task communication and synchronization. This may in itself facilitate the development of distributed applications, relieving the programmer of the task of dealing with many low-level programming details and allowing him to focus only on the application relevant problems. However, in this paper we are especially interested in discussing another aspect of this framework, namely its potential for reusing existing sequential applications.

We will present a real application of the Tracs environment to the reuse of an existing body of sequential software for simulating telecommunication systems. This software was generated by an existing tool, called SPACE, developed by the Telecommunication Research Group at our Department and used in day-to-day research. SPACE allows the user to define digital transmission systems graphically through an interactive block-diagram editor and automatically translates the graphical description into a Fortran program actually implementing the different signal processing functions referred to by each block. The typical

1. Introduction

Recent technological developments have made widely available networked groups of workstations whose aggregate computing power make them both a practical and a cost-effective approach for many computer-intensive applications that do not require supercomputing resources [1]. Practical experience in several academic and industrial environments, moreover, has shown that these groups are often used to run pre-existing sequential software properly modified, rather than as a platform to run parallel applications built from scratch [2]. The issues related to methods and techniques for facilitating the reuse of sequential software in parallel and physically distributed computing environments are thus gaining more and more importance.

In this paper we present and discuss our experience in this area and, in particular, we give special emphasis to the issues related to the integration of reuse-oriented capabilities in general-purpose programming environments.

Our experience is based on Tracs, a graphical programming environment designed to facilitate the development of distributed applications involving groups of networked, heterogeneous machines [3]. Tracs has been developed by some of the authors at the Dipartimento di Ingegneria dell'Informazione of the Università di Pisa as part of a project funded by the Esprit III research programme. Tracs promotes a component-based approach to the design of distributed applications. Applications are

† Tracs—Flexible real-time environment for traffic control systems—project No 6373, sector II.3.15.
SPACE-generated Fortran program consists of a sequence of independent simulations, each related to a different set of user parameter values that identify a particular configuration of the transmission system (filter bandwidth, signal-to-noise ratio, etc.). This kind of computation can be easily modeled according to the so-called process farm paradigm [4].

Tracs comes into play since tasks that interact according to the process farm programming paradigm belong to the set of components that are partially automatically built by Tracs. The process farm is also a relevant example of when a user is required to supply only application dependent sequential code, whereas the Tracs process farm generator will take on producing the code for inter-task process farm interaction style.

We have built a filtering tool that restructures SPACE-generated sequential programs to let them match the input format required by the Tracs process farm generator. The result is that users of SPACE have their programs transparently parallelized and can take advantage of networked and heterogeneous workstations to run their simulations.

It is worthwhile to point out, however, that the techniques we have used are not peculiar to SPACE-generated programs only. Conversely, as it will be clearer from the rest of the paper, we may reasonably claim that they can be easily applied to most programs that may be structured according to the process farm paradigm.

Special emphasis has been placed also on the fact that the target architecture was physically distributed. In particular, we work in a setting in which the various machines involved in the computation may fall under different administrative domains, so that they can be unexpectedly turned off, rebooted and disconnected from the network at will by the respective owners. Accordingly, a fundamental part of the automatic parallelization process results from the distribution of the processing system. Our approach hides these aspects from the programmer and carefully exploits the semantics of the application to achieve a solution that is simpler than in the more general case of distributed applications.

The paper is structured as follows. In section 2 we will discuss how process farms are supported by Tracs and give a simple example. In section 3 we will outline the tool SPACE and discuss in detail the problems we have encountered in parallelizing automatically the programs it generates. Then, in section 4, we will discuss the problems arising from the fact that the computing system is physically distributed and outline our solutions. Finally, in section 5 we will describe the status of the work and enter into some discussion.

1.1. Related works

The study of techniques and tools able to facilitate the programming of physically distributed computing platforms has become an extremely important research and engineering area. Industrial products that harness collections of workstations on a network for providing functionalities of batch processing are already commonplace [5]. Batch processing, however, is just an extreme of the possible applications in this area, the other extreme being represented by those applications that are intrinsically parallel, such as the one discussed in this paper.

Within this scenario, a significant role is played by those programming environments for distributed systems that aim at facilitating both the design of distributed applications and the migration of programmers expert in sequential programming. There are several notable examples of this kind, in which the programmer is required to provide only sequential pieces of code: Enterprise [6], an environment in which the programmer graphically specifies the parallel programming paradigms to be associated with the various modules; HenCE [7], that follows a similar philosophy: Paralex [8], where computations are expressed according to the so-called dataflow paradigm; CODE [9], where dependencies between sequential modules are expressed graphically and where each module is activated by its own firing rule which is a function of the module's input dependencies.

Equally important, however, is the reuse of existing sequential programs, an issue in which there is not much real experience reported in the literature in the context of distributed system programming. In this paper we discuss extensively the problems we have encountered in a real experience of this kind. Furthermore, major emphasis is often placed on hiding from the programmer the issues related to exploitation of parallelism. Unequal emphasis is placed in relieving the programmer of the issues related to the physical distribution of the computing platforms: potential for independent failures and for possibly unreliable communication. Major notable exceptions include [10,11], as well as a few other approaches—such as in [6,8]—layered on top of toolkits that address distribution in a systematic way [10]. However, this layering approach may greatly increase both size and management cost of the overall system. On the other hand, by properly exploiting the semantics of the application, efficient solutions can be devised that are much simpler than in the general case of distributed applications. The approach we have followed is an attempt in this direction.

2. Automatic generation of process farms in Tracs

Various methods to classify parallel programs, according to the kind of parallelism they exploit, have been proposed in the literature [4, 12, 13, 14]. The term process farm is often used to indicate parallel programs in which there are many copies of the same task—called worker—which perform independent executions working on different data (job). The whole computation takes place under the control of a task called master. Communications among tasks regard only jobs and results: each worker contacts the master to get a job, performs its part of elaboration and sends results back to the master. While carrying out a job, a worker need not perform any communication with other tasks. Many variations around this scheme exist, including slightly different terminology, still maintaining the basic features outlined here [15].
A Baltoli et al

Figure 1. Overall structure of Tracs process farms. The Tracs-generated code wraps around the user-provided code and takes care of all the details concerning synchronization and inter-task communication. It in turn relies on the Tracs run-time support for message-passing and for dealing with possible heterogeneity in data format. The Tracs-generated code invokes user-provided functions whenever the elaboration depends on the specific application. The application also determines the actual structure of both jobs and results, which are provided by the user through proper message models.

Figure 2. Implementation of process farms in Tracs. The user provides source files containing sequential code (figure 1) and a description of jobs and results. The latter is translated by a tool (specialized for the master and for the worker) into an additional source file containing the code to be wrapped around the user-provided code. The result is two Tracs task models.

Before presenting our implementation of process farms, it is useful to give a brief outline of Tracs. Space precludes a full discussion, that can be found in [3]. The programming model supported is based on message-passing and is a rather traditional one [16]. The primary way in which tasks interact is via a remote procedure call mechanism that we call service [17]. Tracs places special emphasis on promoting a modular approach to the application design. Programmers are encouraged to structure their applications in term of basic design components with well-defined interfaces: structures of messages (message models); descriptions of tasks in terms of code and communication interface (task models); logical structures of groups of tasks, such as farms, pipelines, grids and so on (architecture models). Design components are context-independent, thus greatly facilitating their reuse.
Figure 3. Summary of the automatic parallelization process. The boxes delimited by dotted lines emphasize that the enclosed parts were developed independently of each other. Starting from a graphical description of a digital transmission system, SPACE generates a sequential Fortran program that simulates it. This program is then restructured by a text filter to let it match the format required by the TRACS process farm generator. Finally, the files output by the text filter are input to the generator, that completes the implementation of the TRACS design components necessary for building the corresponding process farm. The various steps are hidden by the SPACE graphical interface. The filter will be discussed in more detail in section 3.2 and in figure 7. Originally, the TRACS process farm generator was not meant to be used in conjunction with SPACE, but it was simply a means to make it easier for users to implement distributed process farms.

The crucial feature of TRACS in the context being discussed is its ability to generate automatically some implementation details of certain design components. In particular, this is possible for task models participating in a process farm. In this case, the user writes only part of the task models code, the one that actually splits the problem up into separate jobs and that carries out these jobs. The environment will generate automatically the remaining part of the code, that will make the task model behave as desired (figure 1). In practice, the user first describes jobs and results by means of TRACS message models. Then, he writes the code for building and processing jobs and results according to a certain required interface [18]. Finally, he invokes the process farm generator that generates the remaining code and combines it with the user-provided code to build two complete TRACS task models—one for the master, the other for the worker (figure 2).

The difference with an ordinary TRACS application is that the user need not build task models in their entirety: he provides only part of them, their completion being performed automatically. Furthermore, the user-provided code need not invoke message-passing constructs, but it can even be completely sequential. In particular, the integration between SPACE and TRACS has been performed by restructuring the SPACE-generated sequential programs to let them match the interface required by the TRACS process farm generator. The overall structuring is outlined in figure 3.

As a simple example, consider the problem of computing the values of \( \sin(x) \) across a specified interval. In this case, a job is basically a value for \( x \), whereas the result of a job is a pair \( x, \sin(x) \). The corresponding message models can be defined graphically, as the other design components supported by TRACS [3]. The functions that must compose the code for the master and the worker are outlined in figure 1 and detailed specifications for them are given in [18]. Such a code can be written in any of the languages supported by TRACS, currently C, C++, Fortran. It is possible, for instance, to write the master in a language and the workers in a different one, or even to use different languages for workers running on different machines. Fortran code for the example being discussed is given in figure 4 (right) and figure 5.

It is important to point out that TRACS process farms have been implemented as a sort of add-on to an existing prototype: the master task is simply a task in which one source file is generated by a tool, therefore it looks to the rest of the environment like any other task. The same is valid for the worker. In fact, the internals of the rest of the environment do not even know of process farms, which thus inherit all the features of TRACS without any special treatment—management of data heterogeneity, generation of a proper Makefile, and so on. As will be discussed in section 5, we suggest that this modular structuring...
character*56    jobhdl
real            jobx
common/job/jobhdl, jobx

character*56    resulthdl
real            resultx
real            resultsin
common/result/resulthdl,
* resultx, resultx.sin

Figure 4. On the left there is the Fortran description of jobs and results. These are common blocks generated automatically by the environment starting from a graphical description. The first member of each—jobhdl and resulthdl—is needed by the Tracs run-time support and its content is transparent to the user. Notice the correspondence between field names in message models and in common blocks: the latter is obtained by prepending the name of the common block and an underscore to the former. On the right is shown the user-provided code for the worker task model of the example, which is composed of just one single function meant to carry out the actual computational job. Its structure is fairly simple: it fills in the common block named result and returns the predefined value FARM_TRUE.

is a sound engineering basis for supporting the reuse of sequential software.

3. Automatic parallelization of sequential software

3.1. Overview of the Software Tool SPACE

SPACE (Software PAckage for Communications Engineering) is a specialized software tool running on UNIX graphic workstations equipped with the X11 library xlib, developed by the Telecommunication Systems Research group at our Department to aid the design of advanced digital transmission systems.

Specifically, SPACE belongs to the last generation of interactive user-friendly time-domain simulators of transmission systems, and is basically composed of three main modules integrated in a user-friendly operating environment:

- **Interactive Graphic Interface** with a simple editor of block diagrams and the standard pop-up menu operating environment
- **Automatic Program Generator** that builds-up the simulation program relevant to the system described by the block diagram
- **Core Library** containing the code of the signal processing modules corresponding to the different blocks in the diagram

The graphic interface allows a quick definition of the overall architecture of the transmission system to be analysed (with no need of a special skill dedicated simulation languages), and produces a list of modules and connections that is input to the automatic program generator. The latter outputs the source of the Fortran program that implements the time-domain simulation of the particular system. This main program is linked to the core simulation library that contains the object of the various Fortran subroutines to carry out the signal processing tasks inherent to a transmission system structure (signal generation and filtering, spectral analysis, bit-error rate evaluation and so on). All simulation results can be managed through a further integrated custom graphical tool available within the start-up operating environment to visualize signals and/or output data.

A didactic application example of SPACE is shown in figure 6, that depicts a trivial system composed of a linear BPSK modulator followed by a Nyquist root-raised-cosine filter [19], a spectrum analyser, a signal logger and an eye pattern scope. As is shown, the pop-up menu Supervisor in the title bar contains a few items related to simple job-control functions, including automatic program generation, simulation build-up and execution, and the Parallel Execution function that is the subject of this paper. Through the functions in the further item Variables in the title menu (not shown), the user can define some system attributes (typically, a signal-to-noise ratio, a filter bandwidth etc) to be regarded as variables assuming values in a finite enumerated or range & step set.

3.2. Parallelizing SPACE-generated programs

As addressed above, the typical use of SPACE involves defining some attributes of the telecommunication system as *parameters*. These may assume a finite set of values within user-defined ranges, and the behaviour of the system is simulated for each possible combination of these values. Hence, a SPACE-generated program is basically structured as a sequence of nested loops, one for each parameter, where the innermost one executes the same simulation procedure with different input data.

The key point to note is that the various executions of the simulation procedure are independent of each other, therefore SPACE-generated programs can, in principle, be easily parallelized according to the process farm paradigm. It is straightforward to realize that, roughly speaking, a worker would execute the simulation procedure on a job constituted by a given combination of parameters, whereas the master would execute all the code dealing with initialization and loop control.

To parallelize SPACE-generated programs automatically, thus, all that is needed is to adapt their structure to the format required by the Tracs process farm generator (figure 3). This is done through a simple filter written in lex and yacc that generates several output files containing
For this reason, the master will terminate immediately: it will terminate only when all the results of the jobs previously dispatched have been received. When the master receives a result, this is passed to farm_result() in the result common block, that in this case simply prints the value on the screen. Before terminating the master, the function farm_exit() is invoked, which in this case does nothing. A more realistic example could involve opening a file for saving the results within farm_init(), saving the results on that file within farm_result() and closing the file within farm_exit(). Note that the variables keeping track of the interval to be scanned are stored in a common block (interval), since they must be accessed by two functions (farm_init(), farm_job()) and since their values must survive across multiple invocations. Actually, these constraints might have been satisfied by passing the needed variables as parameters from the (hidden) main routine. Since different process farms need different parameters, however, the automatic generation of main routine would have become more complicated, therefore we rejected this approach for merely pragmatic reasons.

Reusing sequential software in a distributed environment

Concerning the code for the master and for the worker, it must comply with the interface required by the Tracs process farm generator (section 2 and [18]). The code for the worker contains the only function farm_work() which is obtained basically by copying the assign, definition, execution and exit sections (figure 7, left). The code for the master contains instead four functions, that are basically related to the reading and do sections. These contain global variables initializations—e.g. parameters, intervals and steps of variation—and loop control, respectively.

It is worthwhile discussing in detail the problems related to loop control, however, since this is where the changes operated by the filter are more substantial. The point is that there is a mismatch between what farm_job() is required to do and the structure of the do section, its counterpart in the SPACE-generated program: whereas the former is required to prepare a job and return immediately, the latter is instead a sequence of nested do statements. Note that hand-written process farms are most likely structured as the do section, therefore this problem is of general interest.

To discuss how we have solved this problem, let us consider the sample do section shown in figure 9, left. Such a code must be the starting point for generating a farm_job() function satisfying the following requirements:

(i) It does not contain any loop.

(ii) When invoked repeatedly, it executes the various statements stat.i, stat.j, stat.k, in the same order, the same number of times, and with the same sequence of values for the variables i, j, k, as the do section.

The corresponding solution, which is automatically generated by the filter being discussed, is shown in figure 9,
Figure 6. An example of a block diagram generated within the SPACE environment. The system is constituted by a linear BPSK modulator followed by a Nyquist root-raised-cosine filter and by three output blocks: a spectrum analyser, a data logger and an eye diagram viewer. The simulation of this system is performed for each combination of the values of certain parameters describing the internal behaviour of both the modulator and the filter (not shown here).

Figure 7. Summary of the actions performed by the filter. Starting from a SPACE-generated Fortran program (box on the left) the filter generates the sequential simulation code for both the master and the worker, as well as a description of what jobs and results are. The meaning of the various sections forming a SPACE-generated program is summarized in figure 8.
Reusing sequential software in a distributed environment

<table>
<thead>
<tr>
<th>Section name</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>include</td>
<td>Inclusion of user-defined modules.</td>
</tr>
<tr>
<td>declaration</td>
<td>Declaration of all the variables used in the program.</td>
</tr>
<tr>
<td>reading</td>
<td>Reading of initial values for parameters from input file.</td>
</tr>
<tr>
<td>do</td>
<td>Beginning of loops on parameters.</td>
</tr>
<tr>
<td>assign</td>
<td>Calculation of parameters given as expressions.</td>
</tr>
<tr>
<td>definition</td>
<td>Initialization of data structures describing system blocks.</td>
</tr>
<tr>
<td>execution</td>
<td>Main simulation cycle.</td>
</tr>
<tr>
<td>exit</td>
<td>Writing of results to output files.</td>
</tr>
<tr>
<td>enddo</td>
<td>Closing of loops on various parameters.</td>
</tr>
</tbody>
</table>

Figure 8. Meaning of the sections composing SPACE-generated Fortran programs (figure 7).

```fortran
do i=IMIN, IMAX, STEPI
  do j=MIN, MAX, STEFJ
    do k=KMIN, KMAX, STEPK
      if(gate.i.eq.1) then
        gate.i = -1
      else
        j = JMIN
      enddo
    enddo
  enddo
enddo
```

Figure 9. Example of do section (left). There are three loops controlled by the parameters i, j, k, respectively. On the right is shown the basic structure of the corresponding `farm_job()` function.

right. The variables controlling the loops (parameters i, j, k) are placed in common blocks, so that their values are maintained across multiple invocations. The code makes use of an additional set of `gate` variables, one for each loop but the innermost one (`gate.i`, `gate.j`). `Gate` variables are obviously placed in common blocks and they determine whether a loop iteration has been completed. There is an if branch for each loop except the innermost one, controlled by the corresponding `gate` variable (lines 1–9, 10–19). The innermost loop has no `gate` variable, hence its statements are always executed (line 20). When a branch is entered, the `gate` is closed so that it will not be entered again at the next invocations (lines 3, 11). Then, the statements associated with that loop are executed (lines 4, 13) and the parameter associated with the next loop is set to its minimum value (lines 5, 14). Finally, before leaving a branch, it is checked whether that loop has been completed—e.g., the gate of the previous branch must be re-opened (lines 16–18, 22–24). When the outermost loop has been completed, no further statements must be executed (lines 7–8). To make the solution work, the initialization performed in `farm_init()` must be augmented by statements that set to 1 all `gate` variables, and that set to its minimum value the parameter controlling the outermost loop—e.g., i = IMIN.

Two issues, concerning variable declarations, are to be taken into account. On the one hand, SPACE-generated programs contain only a main program whereas the code for the master must be structured in four functions, thus one has to figure out which functions use which variables. On the other hand, one has to figure out how to separate (and possibly duplicate) variables across the master and the worker code. Accordingly, the filter classifies variable
declarations in the following non-overlapping groups: variables that describe jobs and results; variables that must be used by more than one master function (thus they must be contained within common blocks, see also the caption of figure 5); variables that will be used locally by the worker or by a master function. Based on this analysis various include files are generated, that are included by functions, source files.

The remaining problem is associating the variable identifiers used throughout the Fortran code with the common blocks job and result, which are the areas where the code generated by the Tracs process farm generator will buffer incoming/outcoming messages. Differently stated, the code generated by the filter will be able to associate the variables that constitute a job and a result with the corresponding common blocks. To show our solution, let the message model in figure 10 (left) be the XDR description of a result, that the filter obtained by analysing the variables of a certain SPACE-generated program. The fields of the XDR description have the same names as the corresponding variable identifiers. For the Tracs process farm generator to work properly, it suffices that the filter uses these names also for the fields of the common block result (figure 10, right). This simple solution allows leaving both the SPACE-generated code and the Tracs machinery for dealing with process farms unaltered.

4. Taking into account physical distribution

4.1. Overview

An important implication of the fact that we are dealing with distributed process farms, is that the various participating machines may be independently administered, for instance user-owned workstations that lend CPU cycles at low priority, or clusters belonging to different groups in either the same or different organizations. In these cases, any machine may unexpectedly leave the farm without any prior negotiation—e.g. shutdown, crash, kill off the processes just because the owner of the machine wants to withdraw it. Unless the application has been structured to cope with this kind of event, it may thus easily hang up or crash. Attacking these issues becomes of fundamental importance, especially for long simulations, in order to exploit efficiently the available computing resources. Unfortunately, constructing an application that takes into account the potential for independent failure of the various parts composing a distributed computing system is definitely not a straightforward job [20, 21, 22]: It may involve know-how that is orthogonal to the object of the application itself and, in any case, it may require a substantial investment. In practical settings, it is not uncommon to proceed by just launching the application and hoping that it will complete without any failure occurring meanwhile.

Indeed, efficient solutions can be developed by properly exploiting the semantics of process farms. They can also be made transparent to the user-provided code, by properly structuring the process farm generator. That is, we suggest that the code automatically generated to be wrapped around the user-provided code can, and indeed should, attack both parallelism and physical distribution at once, starting from a description of the desired process interaction style. In a sense, this may be viewed as a generalization of RPC stub compilers [23, 24]. In the remaining part of this section we will discuss the assumptions underlying our solutions and give some intuitions about them. More details can be found in [18].

4.2. Outline of the proposed solutions

Roughly speaking, the potential for independent failures of the various parts composing a physically distributed computing system may affect a process farm in three ways. In the case of a worker crashing, the master would never receive the results for the job dispatched to that worker, so, unless the master is prepared to cope with this kind of event, the whole computation would be blocked ‘forever’. In the case of the master crashing, it must be possible to restart the computation so that the jobs already carried out before crashing are not dispatched again, otherwise the computing resources would not be used efficiently. For the same efficiency reasons, it must be guaranteed that in the case of communication failures, for instance transitory network partitions that isolate some workers from the rest of the system, results obtained by those workers are not lost.

We make three basic assumptions. First, we assume that workers’ execution may be thought of as idempotent, i.e. repeating the same job more than once is harmless from the point of view of the correctness of the computation. This is a simplifying assumption but it is also a realistic one for the vast majority of applications of process farms. Second, we model our computing system as an asynchronous one, that is, there is no upper bound on the relative speed of processes nor on the time necessary to deliver a message [25]. This is a realistic way of taking into account the varying load and scheduling strategies of the available computing resources. Unlike the previous assumption, this is not a simplifying one, because a fundamental property of asynchronous systems where processes may fail is that it is not possible to decide in finite time whether a process that does not respond is crashed or is simply very ‘slow’ [26]. Accordingly, when implementing a real system, a process that does not respond for a ‘while’ is often declared crashed [27, 28, 29]. How to deal with processes that were assumed crashed but that in fact were not, is an application-dependent issue. Waiting for an unlimited amount of time would be safe—one would be able to decide whether the process is really crashed or not—but it would not be live—the computation would not make any progress. Besides, choosing a proper trade-off between safety and liveness is one of the fundamental problems of distributed computing. Finally, we assume that any failure occurring during a simulation eventually recovers.

The Tracs-generated code for the master maintains internally the following data structures: a completed list, containing all dispatched jobs whose results have already
been received; and a pending list, containing all dispatched jobs whose results have not been received yet. Roughly speaking, the idea for coping with worker failures is that when the user-provided code has no more jobs to dispatch, the master dispatches again those jobs that have been pending for a 'long' time. Although one might end up with more workers working on the same job, note that this is harmless both from the point of view of the correctness—jobs are idempotent—and of the efficiency—when the computation is about to finish, a worker would have nothing else to do.

However, processing the same result several times might not be correct, for instance because it might involve a non-idempotent operation such as writing on a file. To avoid this, the Tracs-generated code makes use of job identifiers, i.e. pieces of information that uniquely identify each job and are managed transparently with respect to the user-provided code: a job description sent by the master carries a job identifier, that will be attached by the worker to the corresponding result; furthermore, job requests sent by a worker carry the identifier of the last job carried out by that worker. By making proper use of job identifiers and of completed and pending lists, the master is able to discard multiple results for the same job.

Note that multiple results might also be caused by retransmissions operated by a single worker, that are necessary to cope with transitory network partitions and/or master unavailability. Retransmissions might also cause the delivery to a worker of multiple descriptions of the same job, which are discarded by means of mechanisms similar to those just outlined.

Finally, the master state is checkpointed on permanent storage whenever \( K \) jobs have been completed, so that it may be restored in case of crashes. Although this avoids restarting the computation from scratch, a crash might cause a job to be multiply scheduled or a result multiply processed, because the state since the last checkpoint is volatile. However, as the master is usually idle for a large percentage of time, setting \( K \approx 1 \) is not unreasonable, unless a worker is running on the same machine as the master. By offering to the application the ability to specify \( K \), we make it able to select the desired tradeoff.

5. Status and discussion

At the time of writing, a prototype that automatically parallelizes SPACE-generated programs and run them on a cluster of HP 9000/735 and 732 workstations is already operational. Since Tracs provides support for Aix 3.1, Ultrix-RISC 4.1, SunOS 4.1.3, FreeBSD 1.0.0, including these other platforms should be a smooth process.

A simple example of the performance improvement that can be obtained is reported in figure 11. In the project being considered, however, performance is not a real issue: typical SPACE workloads involve, in practice, jobs whose computing times far outweigh the communication times. Accordingly, certain optimizations that had been originally planned—for instance, decreasing the job scheduling time by means of a sort of chunk scheduling that dispatch several jobs to a worker in a single operation [31]—have simply been put aside. The users of the system, that is the Telecommunication Systems Research group at our Department, have been satisfied by the experience gained and a further release of SPACE integrating the new features is being developed.

From the software engineering point of view, it should be adequately emphasized that reuse of sequential code in a distributed environment can hardly come for free. Our experience is a strong confirmation of this: in spite of the fact that we had to deal with programs generated automatically, hence having a 'regular' structure; that their intrinsic parallelism was probably the easiest to manage, because of the extremely simple communication pattern; that our typical workloads exhibit a large computation-to-communication ratio thus relieving us of the need of implementing elaborate and/or tricky optimizations; nevertheless, we had to face many low level distributed programming details. Although they have been solved, a fairly skilled programmer was necessary.

As a consequence, the overall Tracs framework has turned out to be an extremely valuable feature in practice: a process farm is simply a set of design components that happen to be generated automatically, and therefore it inherits all the facilities of Tracs—message-passing, data heterogeneity, mixed-language programming, remote compilations and loading, etc.—without modifying Tracs in any way. The key point is that these additional

\[ \text{struct MYFARM_RESULT_IN} \{ \]
\[ \text{SRV_HANDLE } bdl; \]
\[ \text{float } snr; \]
\[ \text{int } ndiff; \]
\[ \text{int } count; \]
\[ \text{\}; } \]

\[ \text{character56 bdl} \]
\[ \text{real snr} \]
\[ \text{integer ndiff} \]
\[ \text{integer count} \]
\[ \text{common/result/bdl, snr, ndiff, count} \]

Figure 10. Example of result of a job: XDR description of a Tracs message model (left), and Fortran common block matching that model (right).
functionalities have been available since the very beginning of the development of the parallelization tools, which has greatly contributed to making their debugging less painful. Perhaps more importantly, it is worth mentioning that during the initial debate about the opportunity of investing human resources in the experience being discussed, a factor that played a crucial role was precisely the consideration that the tools would have immediately inherited all the features of Tracs, and this made us confident that our effort would have been focused only on the essential parts of the problems related to parallelism and distribution.

To sum up, we believe that building the environment around a notion of design component in which some implementation details of certain components can be generated automatically has proven to be an appropriate framework. The internals of the environment do not even need to know about which entity generated which part of which components, therefore automatic parallelization can be implemented transparently to the core of the environment, by simply generating automatically the communication and synchronization code of certain tasks. Indeed, the same scenario discussed here can easily form the basis for implementing additional parallelization tools dedicated to other important parallel programming paradigms—grid-based, dataflow-like, client-server.

The previous considerations thus encourage us to believe that, concerning the reuse of sequential software in distributed environments, the kind of modular structuring proposed in Tracs has a great potential for simplifying development, maintenance and enhancement of both the environment and the tools for automatic parallelization. And, that the Tracs framework may constitute a sound basis for further engineering efforts in this area.

References

Reusing sequential software in a distributed environment


[23] Sun Microsystems *rpgen (Man pages)*


[27] Birman K P 1993 The process group approach to reliable distributed computing *Commun. ACM* 36 36–53


[31] Lilja D J 1994 Exploiting the parallelism available in loops *IEEE Comput.* 27 13–26