Adventures with portability

E. Post
Centre for Advanced Computational Solutions, Lincoln University, New Zealand

Abstract

In this paper we describe the issues we are facing, and some of the solutions we have developed, in porting a complex multi-component simulation model of pastoral dairy-farming scenarios for execution on a high-performance Linux cluster. This exercise is part of a wider goal to develop and implement global optimization procedures for this model which necessarily require very large numbers of simulations to be executed.

The structure of the model poses some particularly interesting challenges as it consists of a strongly object-oriented VisualWorks Smalltalk framework which instantiates third-party "component sub-models", written in a variety of standard (e.g. Fortran, C) and proprietary (e.g. ACSL, ObjectPascal) languages, to represent the functionality of the real-world farm sub-systems. Furthermore, it was unfortunately originally constrained to being implemented in a Microsoft Windows environment and made heavy use of the Microsoft COM/DCOM protocol for communication between objects.

Although still in the early stages of on-going work, the techniques developed and lessons learned should be relevant to aspects of other situations where there is a requirement to run "legacy applications" on high-performance clusters.

1 Introduction

Scientists at a dairy research institute in New Zealand, Dexcel Ltd [1], have developed a simulation model to study the management of pastoral dairy farms and identify sustainable strategies which give the best production while maintaining the cows and pastures in good condition. A computer model is considerably cheaper for evaluating many scenarios than the traditional method of running farmlet field trials for several years. Also, such real-life trials are subject to uncontrolled variables such as climate, which make it difficult to obtain good and repeatable results. Thus a reasonable simulation model of this complex system has a very important role to play in reducing experimental costs and identifying promising scenarios which can then be verified by more focussed field-work. Similar arguments also apply to complex systems in other disciplines of course, so the requirement for good techniques to implement large multi-component models is quite general.

Representation of a pastoral dairy farm system needs models of cows (metabolism), pastures (or other crop species) and soil of the paddocks (fields) at the very least, along with input of climate data (the overall system driver). The climate data can be historical or a model itself. The overall model must also allow various management scenarios to be simulated. Dexcel recognised that for such a system it would be more appropriate to develop a simulation framework explicitly designed to facilitate incorporation of existing and future sub-model components and to readily allow contributions from third parties [2][3]. This approach made it possible to take advantage of ongoing research and development by third-parties and easier to keep up-to-date as component models could be replaced as the current state of research in each area advanced. The framework concept also allows different sub-system models at different levels of complexity to be used depending on the requirements of the specific modelling exercise. This whole system of framework and components is referred to as the Whole Farm Model (WFM).

1.1 Initial development of the Whole Farm Model

The WFM is based on the object-oriented (OO) paradigm, as objects in software map well onto objects in the real world, such as cows, pasture species etc. Legacy software components also tend to encapsulate object-like functionality. VisualWorks Smalltalk [4] was chosen to develop the framework as it has definite advantages in being mature and stable, purely OO, and portable across platforms including Windows, Linux and several other varieties of Unix. Also, its advanced IDE, incremental compilation and automatic garbage collection make development much easier and quicker than in C++.

The initial legacy components that were incorporated into the WFM were the MOLLY cow metabolism model [5] written in ACSL (advanced continuous simulation language) and PasGrow, a pasture model written in Fortran [6]. Both these models, especially MOLLY, were far too complex and extensive to be quickly or easily rewritten. Also, both enjoy reasonably wide use and so may be developed further by others. Thus if later Dexcel wanted to use enhanced versions of these or other models developed by third-parties it would be relatively easy to “plug in” a new external component but much more difficult to update model code that was built-in to a system and couldn’t easily be replaced. Therefore it was important to use each model in its entirety and avoid recoding any part of it. This was also made more difficult because MOLLY has to be run in the proprietary ACSL environment [7].

Incorporation of these external component models into the WFM was managed as follows. The Smalltalk framework contains “proxy” Smalltalk objects for each external component. These hide the details of particular component implementations, making it easier to substitute alternatives when needed [8]. The proxy objects translate messages between the framework and the components by using some inter-process communication (IPC) mechanism, and also manage the component's life-cycle concomitantly with its own. When the source code for the external components is available, and if it is in languages such as C, C++, Fortran or Pascal, it is relatively easy to compile the source code into shared libraries, or executables, with little or no change. IPC between the Smalltalk framework and these components can be managed in various ways. Various IPC options such as sockets, pipes, COM, CORBA, DDE (Dynamic Data Exchange) etc. were considered, with a final decision to use COM influenced by the institutional Windows environment [8]. Since Fortran compilers do not directly support COM a C++ "wrapper" was written to handle communication between the framework and the Fortran pasture model. Incorporating MOLLY was more difficult because of its ACSL environment. However, ACSL provides for the use of DDE and initially this was used. Later an intermediate COM application was written to communicate between the framework and MOLLY [8]. As these components, especially MOLLY, need large amounts of computing resources, both in power and memory, the model was extended to run on a distributed network of PCs using DCOM for IPC [9].

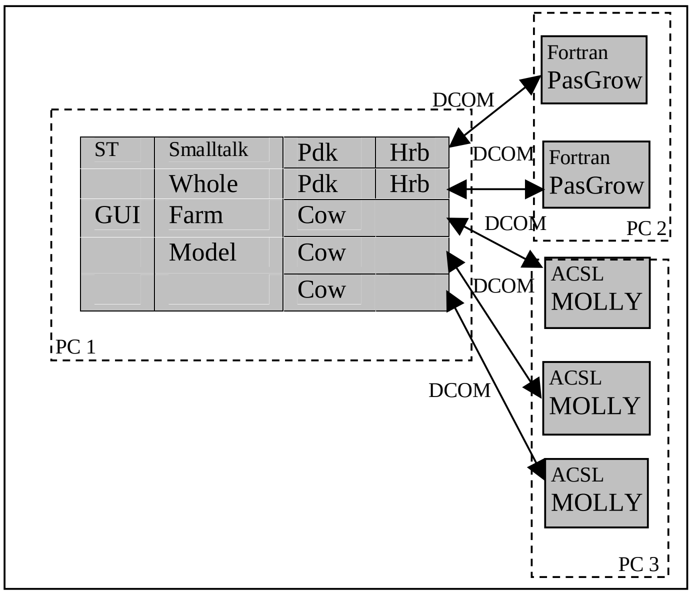

Figure 1: Original system under Windows

Figure 1 illustrates the WFM at the stage when it had been extended to run on a distributed system of several PCs connected by a network. This version of the WFM includes both the Smalltalk framework to manage the simulation and also the graphical user interface, also in Smalltalk. The user selects which components to use and this information is used to run a single simulation, and the user may view various outputs as the simulation runs. Each `Paddock` object (represented by `Pdk`) has several components, such as area, soil type, pasture type. This diagram show how the pasture type is implemented as a proxy `Herbage` object (`Hrb`) which manages and interacts with the external Fortran `PasGrow` component. Also each `Cow` object is a proxy object interacting with an external

MOLLY cow component in ACSL. Communication between the proxy objects in the framework and the external components is managed using DCOM. This means that the framework and external components may run either on the same processor or on different processors. Since the ACSL environment is proprietary all instances of the MOLLY cow component have to run on the machine with the ACSL environment. However, the framework and external pasture components can run on the same machine as the ACSL environment or on different machines. This diagram shows the framework running on one PC, all the pasture components running on a second and all the cow components on a third, but various other combinations are possible.

Since this model has been developed Dexcel scientists have been validating it by comparing its results with actual result obtained from field farmlet trials and have found good correlation, showing that their system can be used to predict production for various scenarios reasonably realistically [10].

The next stage in their research is to use this system in an optimization study which requires running very large numbers of farm simulations with different scenarios so as to identify best, and perhaps new, management strategies. Such work requires far more computing resources than are typically available on one or several PCs on a network so the decision was made to investigate porting the system to a Linux cluster and take advantage of the power of parallel processing. The rest of this paper describes our experiences and lessons learned in this exercise so far.

2 Porting and parallelizing the WFM

Our immediate aim in porting the farm system from a Windows uni-processor or network environment to a Linux parallel environment was to get the system up and running and parallelized as quickly as possible so it could be used in an optimization study. We did not have the time or resources for extensively rewriting either the framework or any of its components so it was important to be able to port as much as possible with minimal changes. It was also desirable that as far as possible the same software could be used either in a Windows uni-processor environment or in the Linux parallel environment to minimise maintenance. Overall, we adopted a pragmatic policy of first making it work, and then, as time and resources permitted, improving the way we did things so as to achieve better performance and efficiency.

2.1 Porting the WFM to Linux

We were fortunate in that VisualWorks Smalltalk [4] has virtual machines (byte-code interpreters) for both Windows and Linux environments so it was straightforward to port the Smalltalk framework from Windows to Linux. Only minimal changes (such as path names) to the Smalltalk code were needed and these were implemented in a more general way so as to make the modified framework platform-independent. The WFM GUI is not needed in the parallelised optimisation work and fortunately the strong model-view-controller [20] separation inherent in VisualWorks Smalltalk made it relatively easy to run the WFM either with or without a GUI, again leaving the framework platform-independent.

2.2 Parallelizing the application for the optimisation study

Since porting the Smalltalk was straightforward, and since simple native Smalltalk cows and pasture growth components had been written we could use these in our first parallel implementation and initially avoid the complication of the external components described above. We also opted to simplify the initial parallelization by having each complete farm simulation, including the framework and all components, running on just one processor, and distributing the many separate entire farm simulations over the available processors. However, we recognise that distributing the component models of a single farm over several processors may well be needed later for better load balancing.

2.2.1 Creating the first parallel application

The first step was to create a parallel application to execute many farm simulations on many processors. Since our Linux cluster is a distributed system an obvious solution for achieving parallelization is to use the Message Passing Interface (MPI) [11]. However, MPI is implemented for languages such as C, C++ and Fortran, and we have not yet found an implementation for Smalltalk. This is complicated somewhat because most, if not all, Smalltalks use a virtual machine which interprets intermediate code rather than creating an executable. Although it is possible that ultimately we may be able to link the MPI library to the Smalltalk virtual machine and enable Smalltalk to communicate directly with MPI initial investigation did not show us how to do this easily and time constraints forced us to take an alternative approach to get things working quickly. Thus we wrote a small program in C using the MPICH version of MPI from Argonne National Laboratories [12] to manage the parallelization. This program initialised MPI on each processor in the cluster and then started the Smalltalk virtual machine running the WFM framework as a separate process on each processor by using the `system()` function in C.

2.2.2 Initial communication with files

With a Smalltalk WFM running on all processors we still needed to be able to communicate between the separate parallel MPI/C processes and their corresponding WFM Smalltalk processes. Again we initially took a pragmatic approach to get it working quickly and implemented IPC via files. Each MPI/C program running on each processor would write a file containing the parameters to start each farm simulation. The Smalltalk application on the same processor then read this file, performed the simulation according to these parameters and wrote to a result file. This file was then read by the MPI/C program on the same processor and the result sent back for collation by the master processor. This approach was extremely inefficient as processes spent a lot of time waiting for files to be written, and reading and writing files. However, it served well as a first step in getting the parallel application running.

2.2.3 Improving inter-process communication by using sockets

The next step was to replace the communication via files with something more efficient. Ideally we would prefer to use MPI eventually, but as we have discussed above this has not immediately been possible. Even though the Windows version of the WFM uses COM/DCOM we decided not to use this as COM is essentially proprietary software for Windows, although COM equivalents do exist for Linux. CORBA [13] was a serious candidate, offering not only robust inter-object communication but integral management of the whole component-object lifecycle, but it has a significant learning curve and we decided to use sockets, initially at least, for a number of reasons. e.g. it is easy to implement sockets in both C and VisualWorks Smalltalk, and sockets are a standard communication method on both Linux and Windows. Sockets also allow communication between processes on either the same processor or on different processors connected by a network

Another of our reasons for choosing sockets was that this is one of the underlying communication mechanisms in implementations of MPI and we consider that even if we can't eventually persuade Smalltalk to use MPI directly perhaps there is a possibility that we can persuade Smalltalk to communicate via a socket which may be managed on the other side by MPI. However, we have not explored this yet and may not follow this route if others prove better. We found the discussion of sockets in [14] and [15] very useful, and their example programs were reasonably easy to adapt to our purposes.

2.2.3.1 Introduction to sockets

Sockets are a communication mechanism that allows client/server systems to be developed either on a single machine or across networks [16]. Many Unix functions and network utilities, such as `rlogin` and `ftp`, use sockets for communication.

A server application will create a socket as a system resource, and give it a name. This socket is then bound to a particular port. The client application will then use the name of an existing socket to connect to this socket, via the port. The two applications can then read data from and write data to the socket in the same way as to any other I/O device. Applications can use sockets to communicate with other processes on the same processor or with processes on other processors connected by a network or the internet. More than one application can use the same socket to communicate with the same server application.

Sockets can communicate either with streams or with datagrams. Streams provide a connection that is a sequenced and reliable two-way byte stream, so that data is guaranteed not to be lost, duplicated or re-ordered without an error being reported. Large messages are broken into smaller sections, transmitted and then recombined in order. A datagram socket does not establish and maintain a connection, only small amounts of data can be transmitted and there is also no guarantee that the data has been received correctly or in sequence.

Sockets can read and write typed data so as long as both applications are using the same types they can read or write the data directly. If there are implementation differences between the processors on which the communicating processes are running, such as with multi-byte integers there are functions defined in `netinet/in.h` that can handle some of these conversions.

2.2.3.2 Performance improvement after implementing socket communication

Using sockets rather than files improved parallel performance significantly. When running our application using sockets on our Linux cluster consisting of nine 800 MHz processors, each with 128Mb memory, connected by private, switched 100 Mb Ethernet, it took only 6.3 hours to run 112,500 1-year simulations of (very small) dairy farms as compared to 33 hours using files, and CPU utilization improved to 99% where with files it was very low.

2.2.4 Porting and parallelizing an external component

Having achieved a reasonably efficient parallel implementation of a “pure Smalltalk” WFM, the next task was to incorporate the external component models. So far we have ported only the Fortran PasGrow model. It was relatively easy to get this code compiled on Linux using the GNU f77 compiler. The main problems were the various Microsoft extensions (mainly related to the now unwanted COM extensions) which were removed or replaced with standard Fortran 77. Since we had already successfully implemented socket communication between C and Smalltalk we used the same approach to write a C wrapper incorporating all necessary socket communication and calls to the pasture model and link this with the compiled Fortran code. Socket methods in the Smalltalk proxy objects allowed them to communicate directly with instances of the socket-enabled `PasGrow` executable, created by the Smalltalk `UnixProcess>>forkJob:withArguments:` method.

As we have mentioned the framework uses “proxy” objects as an interface between the framework and the external component models. Thus, if a farm has say 20 paddocks, each growing pasture, then each paddock will be associated via a proxy object with a separate instance of the external pasture growth model running as a separate process. Similarly, if a farm has 50 cows and cows are being modelled by an external component then there will be 50 external cow processes running. Obviously having such a large number of processes running for each farm simulation is inefficient as it will involve a large amount of context switching and large amounts of memory and perhaps resultant paging. However, although the WFM was originally implemented to allow such detailed representation of actual farms, smaller examples are quite adequate for many studies, and owing to limitation of time and resources we have taken the pragmatic approach of doing only what was necessary to get the system up and running. Now that we have achieved a reasonably efficient parallel implementation for optimization we can explore ways of improving it. Some possibilities are discussed in the following sections of this paper.

2.3 The parallelized optimization application

Figure 2 illustrates the dairy farm system as it has been parallelized for the optimization application, which typically needs to run many thousands of simulations in total. This diagram shows just three processors, the master and two slaves, although the application will run on any number of processors.

The MPI/C program running on the master manages the optimization application, and sends simulation tasks to the slaves, and the slaves receive these

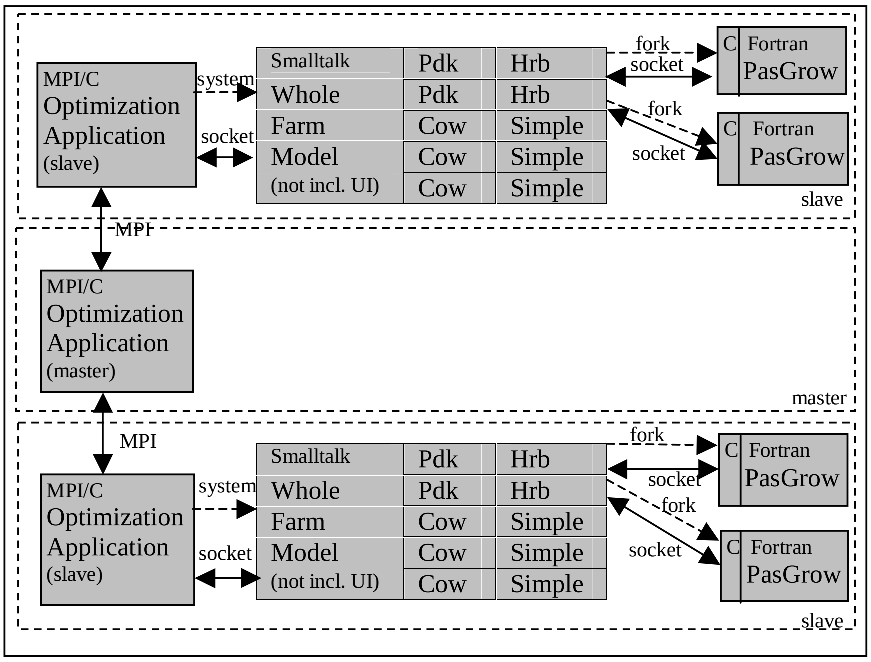

Figure 2: New system parallelized for Linux, showing a master and two slaves, although any number of slaves is possible. A realistic farm specification would require at least 15 Paddocks and 10 Cows

tasks and perform all the simulations, sending the simulation results back to the master, with all master/slave communication by MPI. On each slave the MPI/C program starts (using `system()`) the WFM framework (without GUI) running on the Smalltalk virtual machine which establishes a socket connection between the MPI/C process and Smalltalk WFM process on the same processor and reads an input file which specifies which model components to use.

Each Smalltalk framework then starts up (using the Smalltalk `forkJob` method) and manages through proxy objects the external components modelling pasture growth for each paddock. Each `Paddock` object (represented by `Pdk`) has a `Herbage` proxy object (`Hrb`) interacting with the external Fortran `PasGrow` component with sockets. This system is still using a `Simple` cow model written in Smalltalk as the more complex external cow model has not yet been ported.

When all simulations are completed the master sends by MPI a termination message to each slave which then, via sockets, sends a termination message to the Smalltalk WFM framework which then terminates itself, closing down the Smalltalk virtual machine.

3 Discussion and future work

The work described in this paper is still in its early stages. Thus many of the issues we are dealing with are closely linked to the direction of our future work.

For this reason we combine discussion of these issues with our plans for future work.

As our primary purpose is to be able to use the parallel model for optimization studies our first priority is to increase the usability of this model, mainly by porting more components. However, as time permits we will also attend to other issues such as increasing the overall flexibility of the model and improving the parallel performance, e.g. By improved inter-component communication and parallelizing at a finer level than the whole farm.

3.1 Porting of further components

Our work so far has shown that is possible to port, parallelize and incorporate legacy components written in various languages into a complex system. We hope to continue this work by porting and parallelizing further components, such as a more complex model of a cow. We are also likely to port additional models for modelling pasture and crop growth and cows (and other animals) which are written at different levels of complexity and which can then be selected as alternatives when running the parallel WFM.

The highest priority is the powerful MOLLY cow model, and although it should be possible to port this model more or less as it stands, since ACSL will run on Unix as well as on Windows, it is likely the cost of licence fees to run ACSL on several processors in parallel will be prohibitive. Also any other users of the WFM will need to have ACSL licences, decreasing the opportunities for other people to use our work. Communication with ACSL may also be a problem that is not solved as easily as we have with sockets elsewhere in our application. Thus this is one case when we may actually rewrite the model in a more portable language and discontinue use of ACSL, because the cost of rewriting may be cheaper than the cost of licences and possible restrictions on the number of processors. However MOLLY is a component originally developed by third-parties and we will need to negotiate permission to do this.

In considering whether to re-write an existing model or to use it as legacy code it is important to consider the relative costs of porting as compared to re-writing and such issues as licences. Obviously the quality of the model and its degree of validation (often the most expensive part of model development) are important factors. Each case must be considered individually.

3.2 Replacing socket communication by MPI

Our initial intention was to use MPI for all IPC. As we have already explained this has not immediately been possible and we have therefore used a simple socket scheme as a first solution. However, MPI already takes care of many IPC issues such as provision of buffers and protection against buffer overflow, automatic handling of possible data-type incompatibilities between applications and/or computer architectures, security of communications etc. If we continue to use sockets then to some extent we will have to re-invent the wheel and take care of these issues ourselves. MPI is also explicitly designed to achieve parallelism whereas sockets are not, and since our primary purpose is to achieve efficient parallelism MPI would undoubtedly be a better solution. Also, expertise in MPI is perhaps more widely available than expertise in using sockets, and if we want an application that can be safely used in a parallel environment and modified by others, perhaps for other architectures, it would be preferable and more flexible and extendable to use something such as MPI to manage IPC. The portability of MPI would even make it possible for some processors in a cluster to be dual-boot and booted in a different operating system, or perhaps running an emulator for anther operating system. Given these potential advantages it is our intention to use MPI wherever possible and retain custom socket interfaces only when absolutely necessary.

MPI can be used directly whenever a component is written in a suitable language, such as C, C++ or Fortran. It can also be used in other languages which can be linked with MPI libraries and where those languages can make calls to C or Fortran routines in these libraries. If a component is self-contained and is written in a language which can be linked with C, C++ or Fortran compiled code then it is also possible to write a wrapper in C, C++ or Fortran which contains the necessary MPI commands for parallelism and which can call and receive results from the legacy component linked with it. If the source code is not available it may be very difficult, or even impossible, to incorporate such a component. However, if the component is only available as an executable it is probably still possible to communicate with it by writing a wrapper program in C or C++ which calls the executable and receives results from it, but it will only be possible to use MPI for communication with other processes and not with the executable directly.

The biggest difficulty though is with languages which do not produce object code that can be linked with C, C++ or Fortran object code, and our greatest difficulty is in parallelizing the WFM itself where the main framework, which handles the instantiation and interaction of all components, is written in Smalltalk. Smalltalk generally runs on a virtual machine which interprets intermediate code that cannot be linked to other object code. In some circumstances some Smalltalk virtual machines can link with other libraries. Also Smalltalk can usually call and communicate with C routines, as long as the Smalltalk process is the primary process.

We have briefly looked into some of these issues but with the VisualWorks Smalltalk implementation that we are currently using we did not discover how to make a primary process in MPI/C communicate by using MPI with the Smalltalk as a secondary process. Also with VisualWorks Smalltalk we have not yet discovered how to link it with the MPI libraries so that Smalltalk itself can run MPI routines.

We will investigate Smalltalk further, and also other implementations of Smalltalk, such as Gnu Smalltalk [17], to see whether it is actually possible to overcome the difficulties we have encountered so far in trying to parallelize Smalltalk. Specifically we need to be able to link the Smalltalk virtual machine to external libraries such as the MPI libraries and use MPI to run and communicate with a Smalltalk application. Also, even though most Smalltalk implementations allow a C process to call Smalltalk routines is seems that this is only possible if Smalltalk is already running and communicating with a C process it started. It does not appear to be possible to have an independent C process communicate directly with Smalltalk. Thus unless it is possible for Smalltalk to interact with

MPI directly we believe that direct parallelism with MPI in Smalltalk will probably not be possible and the techniques we have described earlier may actually be the only way to achieve parallelism with Smalltalk running on a virtual machine.

3.3 Improving performance

We have mentioned earlier that as a result of the original structure of the WFM each paddock will have its own instance of a pasture model running as a separate process, and similarly each cow will also have a separate process running. The WFM was parallelized in this way for historical reasons because that is the original object-oriented structure of the serial version, and our first priority was to get the WFM parallelized as quickly as possible without having to rewrite it.

This is possibly not efficient as a large number of concurrent processes running on the same processor increases the amount of context switching which will have a detrimental effect on performance. Also, depending on how much memory each process uses and how much overall memory is available for each processor, this could significantly increase the amount of paging needed as context is switched, thus decreasing performance even further.

It is quite possible to redesign the parallel implementation to improve performance so that we do not have so many concurrent processes with their own memory space. This could be done in a number of ways, such as using shared libraries or linking the external component object code with the MPI/C manager program code. This would reduce the number of separate processes and thus context switching and paging. However, this would require some major re-structuring of the original framework which would take considerable effort, and it is important to consider first whether re-structuring the application will improve performance enough to justify the effort. For instance, the Linux cluster will usually not be running in a dedicated mode and will be shared with other users, so context switching, paging and other user's applications will be affecting performance in any case. Also, the amount of physical memory available for each processor is considerably more than was common when the models were first developed, so paging is much less likely to be an issue with the larger amounts of memory now available.

However, now that the application is parallelized we will carry out performance tests to establish how much context switching and paging do actually affect overall performance. Only if they have a significant effect will we study the WFM and see how much effort it would be to change the implementation as described above and whether this would result in any significant gain in performance.

Sometimes it is better to be pragmatic in these issues and accept good performance rather than the best performance because it would just take too much time and effort to achieve any significant gains, and resources could be better used elsewhere, such as in porting further components.

3.4 Extending the parallelism

Since one complete simulation of the WFM for a period of a year only takes less than 30s on an 800 MHz processor (for a modest size farm with relatively simple component models) this was considered sufficiently small granularity to give good performance for our parallel optimization application which may for instance need to run a million such simulations. Thus in our application each WFM simulation is currently executed completely on one processor, with the many simulations themselves distributed among the processors. i.e. the simulations are not as yet further broken down to execute components of a simulation on different processors.

This "embarrassingly parallel" solution is quite appropriate for our application where the simulations are for relatively small farms of perhaps 10 cows and 15 paddocks. However, in applications other than optimization researchers may want to study far larger farms with hundreds of cows and much larger numbers of paddocks. Since, for instance, the complex MOLLY cow simulation takes far longer than the simulation of pasture growth it may be preferable to allocate perhaps one processor to execute the WFM framework and all the pasture growth simulations, and all other processors to work only on MOLLY cow simulations.

Such "internal" parallelism of the WFM has not yet been implemented in our application as so far it has not been needed. However it is relatively easy to increase the parallelism of the WFM to allow various components to be run on different processors, as the framework and each component are independent entities. This can be achieved by using either sockets or preferably MPI. Thus, when we have addressed other more pressing issues we will extend the parallelism of the WFM in this way.

4 Conclusions

We have shown that in certain circumstances it can be relatively easy and reasonably quick to port and parallelize a complex multi-component system written in several computer languages from a Windows environment to a Linux system, provided the original components are reasonably well-designed and well-written (see Recommendations). Moreover we have also shown that by using pragmatic techniques this can be done in a relatively short time and reasonably efficient parallel performance obtained, even when the original computer languages used are not easily parallelizable. Although the work itself was spread over several months the total time needed for the porting and parallelization described in this paper was not more than two person-months of full-time work in total, which is considerably less time than it would have taken to rewrite such a parallel complex system from scratch.

We have shown that pragmatic techniques, such as first using files for IPC and then sockets, provide quick and relatively simple ways to parallelize an existing complex system as quickly as possible while maintaining the original legacy components unchanged.

When the amount of data to be exchanged is small it is appropriate to use sockets when it is not easily possible to use for instance MPI for IPC. Even though sockets are fairly low-level using them is not difficult and allows fast communication between processes both on the same processor and also between distributed processors. Using sockets is fast enough and flexible enough to allow communications between processes but not as ideal and general as MPI.

Also sockets are available on both Linux and Windows operating systems, which aids portability. For instance, if one of the components of an original system cannot be ported to Linux and perhaps has to run in a proprietary environment on Windows, then one or more processors in a cluster could run Windows and communication between the processes on the Linux processors and the Windows processors could be done using sockets if it cannot be done using MPI.

Thus we recommend use of MPI for communication between processes wherever it is possible. However, in cases where this cannot be achieved, such as with the Smalltalk virtual machine, then sockets are a suitable alternative.

We note also that it is probably easier to port legacy systems to a distributed architecture such as a Linux cluster, than to a shared-memory multiprocessor, because the individual architecture of each processor in a distributed system is essentially the same as that of a uni-processor.

Linux clusters are now relatively cheap and readily available and are ideal for this sort of system simulation. However, much modelling and simulation in the past has been done on single processor machines under other operating systems. Thus porting such simulation models to high performance Linux clusters is likely to be of increasing importance in the future and we hope our experience is useful when porting and parallelizing such legacy code.

5 Recommendations

We also make the following recommendations to those developing computer models as part of their research, especially those models that have the potential to be re-used as components of larger systems and those which may be parallelized. (And if they do not have this potential for re-use why are they being developed at all?)

Therefore, in the interests of increasing portability, reusability and extendability of computer models, we suggest the following:

- If possible use the OO paradigm when designing and developing software, as objects map well onto objects and systems in the real world, and the OO approach will facilitate future reuse and extension. If not using an OO language then still design the software with the OO paradigm as far as possible, making all code units modular, passing data into these units with parameters and returning results from each routine.
- If at all possible use standard languages, such as C++, C, Fortran that are widely available for various computer architectures and operating systems. (You cannot necessarily predict that your model will never be used on another operating system.) Of these it is strongly recommended that you use C++ if at all possible as it is object-oriented. Avoid using C or Fortran if you can. However, if you really have to use Fortran then it is advisable to use Fortran 77 for which compilers are widely available at no cost on many platforms. Currently there are no free compilers for Fortran 90 or Fortran 95 for Linux, although there have been in the past and some versions are

presently under development. (Researchers often have little money and may not be able to afford to buy commercial compilers.)

- Anticipate the possibility that the application may need to be parallelized and note that MPI can most easily be used with C, C++ or Fortran applications. Although, as we have shown, applications written in other languages can still be parallelized, especially if they can be linked with a C wrapper, it is still easier to do if MPI can be used directly.
- Avoid proprietary and specialised environments such as ACSL, Microsoft, Borland. Developing software in such environments will limit the re-usability of the model, partly because it becomes too expensive to convert the software to a new environment, and partly because of the cost of licence fees, which may be prohibitive, particularly for instance in a parallel processing environment where the licence fee may be per processor.
- If you must use a proprietary environment for development, such as for instance Microsoft Visual C++, use standard features and libraries in these languages and avoid using implementation-specific libraries. (The documentation should indicate whether features are standard (usually ANSI) or implementation-specific.)
- If it is useful to use for instance a specialised simulation or statistical or mathematical environment or language for developing a model because of the features it provides, try and choose an environment which will allow the export of the model in standard C or Fortran code that can be compiled and which will run independently and does not have to run in a proprietary environment. Then this exported code could be incorporated as a component model for another system.
- If it is essential to use specialised libraries, such as for instance mathematical or engineering libraries, choose those which are well-known and widely available on different platforms, such as NAG [18] or Portland Group. [19] Otherwise develop your software so that it would be relatively easy to use alternative libraries if necessary.
- Use the Model-View-Controller [20] concept when developing software, where the user interface is completely independent of the code implementing the actual model, with these two layers connected by an intermediate controller or transaction layer. The many advantages of this approach are well-known, such as for internationalisation, but in the modelling context some of the main advantages are that it increases the portability of the code for the model. Most incompatibilities between implementations on different operating systems and architectures occur in the graphical user interface which may involve considerable change when porting an application. However, usually the code for the model itself will port with minimal trouble providing it has been written in a standard language using standard libraries. This separation of the model from the GUI is also important in modelling as quite often it will be necessary to run the model independent of its GUI, such as for instance when it becomes a component of a large system or in parallel processing. If this approach is not taken it can take considerable work to separate model code from GUI code.

- Avoid using global data. Not only does this make it difficult to keep modules independent in the application but it can contribute to difficulty in parallelization process.
- Use standard data types such as integers, IEEE floats and doubles and be sure that interacting applications know whether they are using for instance 1, 2 , 4 or 8 byte integers. Also be sure to be aware of whether the code is using either 1-byte or Unicode chars and be consistent. Avoid using implementation-specific data-types such as the Pascal 6-byte Real.
- If the model may be parallelized for a heterogeneous system be aware that there may also be issues with whether integers are little-endian or big-endian. If you can use MPI for inter-processor communication this will automatically be taken care of. However, if you are using sockets you will need to take care of it yourself, although there are functions available for use with sockets that will help with this [14].
- And finally, unless you have the necessary computing skills yourself, we strongly encourage you to employ someone who does have them. High performance computing is quite a complex area and you will get better results if you use an expert in this area.

6 Acknowledgments

We are most grateful to Dexcel Ltd, Hamilton, NZ for financial support for this project, and to Dr M E Wastney, Mr K P Bright and Dr C Palliser of Dexcel's modelling team for their technical support and encouragement. Dr R A Sherlock of SmartWork Systems Ltd provided advice and assistance in connection with the Smalltalk porting. We also acknowledge the support of Lincoln University who provided the original funding to acquire the Linux cluster.

References

[1] Dexcel Ltd., Hamilton, New Zealand, http://www.dexcel.co.nz/

[2] Sherlock, R.A., Bright, K.P. and Neil, P.G. An object-oriented simulation model of a complete pastoral dairy farm. *MODSIM97 – Proceedings of the International Conference on Modelling and Simulation, Modelling and Simulation Society of Australia Inc*, eds McDonald, A.D., Smith, A.D.M. & McAleer, M., Hobart, Australia, pp 1154-1159, 1997.

[3] Sherlock, R.A. & Bright, K.P. An object-oriented framework for farm system simulation. *MODSIM99 – Proceedings of the International Conference on Modelling and Simulation, Modelling and Simulation Society of Australia and New Zealand Inc.*, eds. L. Oxley, F. Scrimgeour, and A. Jakeman, Hamilton, New Zealand, pp 783-788, 1999.

[4] *Cincom Smalltalk Website*, http://www.cincom.com/scripts/smalltalk.dll/index.ssp

[5] Baldwin, R.L. *Modeling ruminant Digestion and Metabolism*, Chapman and Hall, London, 1995.

[6] McCall, D.G. *A Systems Approach to research planning for North Island Hill Country*, PhD thesis, Massey University, Palmerston North, New Zealand, 1984.

[7] Mitchell, Gauthier. *Advanced Continuous Simulation Language (ACSL) User manual/reference*, Version 11, Mitchell and Gauthier Associates, Inc., Concorde, MA.

[8] Neil, P.G., Sherlock, R.A. and Bright, K.P. Integration of legacy sub-system components into an object-oriented simulation model of a complete pastoral dairy farm. *Environmental Modelling and Software*, **14**, pp 495-502, 1999.

[9] Neil, P.G. Distributed simulation using DCOM, *Distributed Computing,* May 99, 15-18, 1999. (formerly available at http://www.distributedcomputing.com, but no longer available)

[10] Lile, J.A., Bright, K.P., Palliser C.C., Prewer W. and Wastney M.E. Modelling dairy production in different regions of New Zealand. *Proceedings New Zealand Society of Animal Production*, **62**, pp 7-11, 2002

[11] *The MPI Home Page*, http://www-unix.mcs.anl.gov/mpi/mpich/

[12] *The MPICH Home Page*, http://www-unix.mcs.anl.gov/mpi/mpich/

[13] *Object Management Group (OMG) CORBA Home Page*, http://www.corba.org/

[14] Stones, R., and Matthew, N. *Beginning Linux Programming*, 2nd ed., Wrox Press Ltd, 1999.

[15] *VisualWorks Internet Connectivity Cookbook*, Cincom Systems Ltd., Cincinnati, Ohio, 2001. http://www.cincom.com/newsmalltalk/prodinformation/pdf/vwiccb.pdf

[16] Stevens, W. R. *UNIX Network Programming: Networking APIs: Sockets and XTI*, Volume 1, 2nd edit., Prentice Hall, 1998

[17] *Gnu Smalltalk Online Reference Manual*, http://www.gnu.org/software/smalltalk/gst-manual/gst.html

[18] *Numerical Algorithms Group (NAG) Home Page*, http://www.nag.co.uk/

[19] *Portland Group (PGI) CDK Cluster Development Kit - Software for Linux*, http://www.pgroup.com/products/cdkindex.htm

[20] Howard, T. *The Smalltalk Developers Guide to Visual Work*s, SIGS Books, New York, NY, 1995