Improved performance control on the Grid

BY M. E. TELLIER, G. D. RILEY* AND T. L. FREEMAN

Centre for Novel Computing, School of Computer Science, University of Manchester, Manchester M13 9PL, UK

In the 2005 paper by Mayes et al., a threshold-based performance control system (PerCo) was described and an initial experimental evaluation was presented. The objective of the current paper is to investigate the role of the threshold value in PerCo and to place the threshold-based rescheduling heuristic on a more principled footing. Simulation enables us to identify the ‘optimal’ threshold value for a particular application scenario, and we show that this optimal value results in a 10 per cent improvement in performance for the application considered by Mayes et al. Furthermore, we find that the execution time of this optimal threshold-based schedule is very close (within 0.5%) to the execution time that results from a linear programming optimal schedule.

Keywords: performance control; migration; distributed components; Grid

1. Introduction

In previous work (Armstrong et al. 2005), we have described the development of a performance control system (PerCo), which aims to improve the performance of component-based applications executing on distributed computing resources. PerCo is a sensor–actuator-based system that monitors the overall execution of a component-based application and takes advantage of mechanisms supported by the application, such as the ability to be checkpointed and restarted, to attempt to find a more efficient way to execute the application on the computing resources available. It uses a policy-driven, closed-loop feedback control model to drive the dynamic redeployment of application components.

In Mayes et al. (2005), experimental results, reproduced here in figures 1 and 2, were reported for the execution of multiple instances of a lattice–Boltzmann simulation code, LB3D¹, under PerCo, using a simple threshold-based rescheduling algorithm, with the threshold value set, somewhat arbitrarily, to a fixed value of 20 (the threshold value acts to bound the relative progress of the multiple instances). In this paper, we present the results of investigating the role of the threshold value and develop a technique, based on simulation of PerCo experiments, to find an optimum value for the threshold for a particular set of applications and computing resources. We explore the effect of the threshold value using a software simulator,

* Author for correspondence (graham.riley@manchester.ac.uk).

¹ This is an example of a scheduling problem where the instances are pre-emptable applications with known amounts of work, e.g. numbers of time steps.

One contribution of 16 to a Theme Issue ‘Crossing boundaries: computational science, e-Science and global e-Infrastructure I. Selected papers from the UK e-Science All Hands Meeting 2008’.
PERCoSim, capable of modelling the behaviour of our experiments. PERCoSim turns out to be a sufficiently accurate, and efficient, predictor of execution time for a range of application types and computing resources. The effect on execution time of varying the threshold value shows a number of local minima, each of which is predicted to deliver improved performance over the threshold value of 20 used in the original experiment. The result of repeating the original experiment but with PerCo using a threshold of 88, the ‘best’ threshold thus identified, is shown. This indicates a performance improvement of approximately 100 s or approximately 10 per cent over the original experiment.

In addition, we have developed an analytical approach to understanding the effect on performance of redeploying components across the computing resources available at run-time. This approach focuses on the way that the mapping of components to resources evolves throughout the execution. We call a mapping of components to resources a mode; a mode sequence is a collection of modes where the progression from one mode to the next in the sequence involves a redeployment of components, and the approach itself is termed mode sequence analysis since we study the space of possible sequences of modes available.

2. Problem motivation and background

Where a particularly complex scientific system is to be studied, a modeller may develop a simulation that integrates several separate executable models, each of which may have been designed and developed independently. The resulting
software architecture is that of a component-based application, where each component is a scientific model. Such components may represent different levels in a multiscale application, where each model acts at a different length or time scale; or they may represent related subsystem models that interact at the same level, where each model describes a distinct underlying physical phenomenon.

A representative selection of simulation scenarios identified as potential candidates for a component-based modelling approach (Armstrong et al. 2005) indicates a general characteristic of such applications to be that the constituent models simulate physical phenomena that develop over time. The resultant application codes are time-stepping codes, where a simulation progresses through a series of iterations.

Both the components of a Grid application and the Grid resources used to execute those components can change dynamically. Component processing requirements may differ over time depending upon the data being generated or manipulated, and the processing capabilities of the resources can change according to load and availability.

For an application to be flexible requires that the components be tolerant of interruptions to their execution and of changes to the underlying execution environment. A performance controllable coupled model is therefore said to be redeployable and malleable. A redeployable model is one that can be interrupted (‘pre-empted’) on one platform and restarted, from the same state, on another. A malleable model is one for which the configuration may be changed dynamically, e.g. the number of processors used.

The application problem addressed in this paper consists of a static composition of multiple (identical) instances of a lattice–Boltzmann simulation code, LB3D (Harting et al. 2004), and a heterogeneous, but static, selection of resources2. Resource heterogeneity is reflected in the different platforms used (Linux, Solaris and IRIX) and also in the processing capabilities of those platforms for an instance of LB3D (there are two identical examples of a ‘fast’ resource, and two other resources with differing performances).

3. Performance control strategy

The redistributive performance control strategy outlined in §2 underwent an initial evaluation in Mayes et al. (2005). The results of this experiment, reproduced here in figures 1 and 2, show that PerCo can take advantage of the ability to redeploy application components (instances) among the available computational resources to deliver a significant gain in performance (approx. 25%) over the static schedule given in figure 1.

In the experiment, a simple threshold-based rescheduling algorithm was implemented in PerCo, which, in effect, attempted to load-balance the execution of the multiple LB3D instances by redeploying the instances whenever the difference in achieved execution progress3 between them, as measured by the number of time

---

2 The assumption of a static composition and static resources does not restrict the generality of the approach, since the threshold algorithm periodically revises the schedule and thus can respond to variations in instances and resources; see Mayes et al. (2005) for a wider discussion.

3 The achieved execution progress includes the costs of redeployment, as discussed in Mayes et al. (2005).

Phil. Trans. R. Soc. A (2009)
steps completed, reached a threshold. The new deployment was generated on the basis of the known relative performance of the applications on all available resources, and the application/resource pairings were swapped accordingly. If we let $TS_i(T)$ denote the time steps completed by component $i$, $i=1,2,...,n$, where $n$ is the total number of components and $T$ is the elapsed time, and let $\theta$ denote the threshold value, then redeployment at time $T$ is indicated whenever

$$\max_{i,j=1}^{n} |TS_i(T) - TS_j(T)| > \theta.$$

The overall effect of the threshold was therefore to influence the duration of each deployment. In the original experiment, the threshold value used to actuate a redeployment was set, somewhat arbitrarily, to a fixed value of 20 time steps. The experiment successfully demonstrated the potential to dynamically control the run-time performance of a component-based application, but the threshold strategy had two main drawbacks. Firstly, the algorithm implicitly assumes that the components are homogeneous. Secondly, the time-step threshold used to trigger a redeployment activity is a relatively arbitrary choice, which we examine in §4.

As published in Mayes et al. (2005), the control policy assumes a homogeneous application problem, where the total number of time steps for each component is the same, and the migration logic defines the threshold in terms of the number of time steps completed thus far by the components. In order to accommodate more general problems of heterogeneous components, we have revised the algorithm to determine the execution progress of each component relative to the progress of the other components, measured as the fraction of overall execution completed. If we let $TTS_i$ denote the total number of time steps to be taken by component $i$, then the threshold test becomes

$$\max_{i,j=1}^{n} \left| \frac{TS_i(T)}{TTS_i} - \frac{TS_j(T)}{TTS_j} \right| > \theta'.$$

The previously reported experiment has been reproduced using this revised policy, with a suitable value for $\theta'$, and is found to produce identical behaviour.

4. Experimental study

The experiment under consideration (i.e. concurrent execution of an assemblage of LB3D instances) follows the ‘parameter sweep’ paradigm and defines a use case of proven scientific value (Pickles et al. 2004). However, underlying the threshold-based control strategy is the assumption that the time-stepping components of an application should finish together to eliminate load imbalance at the end of the execution, and that this can be achieved by restricting the execution progress of any component relative to the others.

The resulting scheduling problem (i.e. pre-emptable components executing on machines that differ in their rates of processing—but these rates are constant for the duration of the experiment—with migration overheads) is well defined in the
literature on processor scheduling (see Kubiak et al. 2002). Thus, we face a known problem in the (new) context of dynamically changing resources and applications (i.e. performance control).

Several algorithms have been proposed in recent years to control the deployment of applications over distributed resources (see Maheswaran et al. (1999) and Zhao (2006) for surveys and comparative studies). All such algorithms seek to assign a set of tasks to a set of resources in a schedule that minimizes some objective function, e.g. the makespan, i.e. the time elapsed between the start of the first component and the completion of the last component. The goal of the threshold-based strategy is also to minimize the makespan; to that end, each redeployment acts to reverse any imbalances in the execution progress of the components. The threshold value used must implicitly balance the need to continuously maximize resource usage against the overheads associated with swapping components. The value used in Mayes et al. (2005) was based upon informed knowledge of the average execution performance of LB3D on each of the resources. Here, we explore the use of a simulator to automate the choice of an ‘optimal’ threshold value.

The scheduling problem has been shown to be NP-complete (Ullman 1975), in both its general form as well as some restricted cases. For this reason, most scheduling approaches use either heuristics or guided random searches to generate close approximations to the optimal schedule(s) (Kwok & Ahmad 1999; Di Martino & Milliotti 2002). Such approaches have the advantage of being simple and directly applicable to practical scheduling problems. However, undertaking any analytical study of the generated schedules is too complex for all but the simplest of problems. An experimental approach is preferable but equally impracticable, given the need to evaluate algorithms with respect to diverse hardware configurations, long-running application problems and a requirement for repeatability.

Given the real need to evaluate, compare and rate the efficacy of different scheduling algorithms, simulation is a popular approach for experimental study.

PerCoSim is a new toolkit that provides a set of core abstractions for modelling Grid scheduling problems and for simulating scheduling algorithms; the details of PerCoSim will be given in Tellier (2009). The basic architecture reflects that of PerCo (Mayes et al. 2004). PerCoSim performs combined continuous/discrete event simulation, with distinct mechanisms of simulation for the Scheduler, Application Performance Steerer and Component Performance Steerer elements of PerCo. This approach differs from existing Grid simulators, such as Song et al. (2000), Casanova (2001) and Buyya & Murshed (2002), and gives rise to an efficient and flexible tool for modelling scheduling behaviour.

As a platform for experimentation, PerCoSim affords the ability to undertake scheduler performance evaluation in a repeatable and controllable manner, unlike real-world experiments. PerCoSim is most usefully viewed as a tool for modelling behaviour as opposed to delivering guaranteed performance predictions.

As motivated by §3, we know that there must exist a threshold value for the given problem, which results in the schedule that most closely approaches an optimal load-balanced solution. Any non-optimal schedule will result in the raggedness in completion times visible in figure 2, as progressively more resources sit idle towards the end of execution. PerCoSim was used to predict the application run-time for every possible threshold value for the given problem, resulting in the graph of figure 3. The accuracy of this output was validated by

Phil. Trans. R. Soc. A (2009)
running a number of ‘real’ PerCo experiments over all boundary conditions of the PERCOSIM results (determined as the local minima and maxima in figure 3), resulting in figure 4. (An indication of the value of PERCOSIM is given by the fact that generating the data to support figure 3 using PerCo would have required over 80 hours, whereas PERCOSIM produced the data in less than a second.)

To analyse these results, we introduce the concept of mode sequences. We consider a set of components $C_i, i=1,2,...,n$, that together form the component-based application, and a set of (virtual) computational resources $R_j, j=1,2,...,m, m \geq n$, on which the components can be executed. The resources are assumed to be dedicated for the duration of the experiment and thereby the processing rates are constant. In general, for example due to varying background load, this assumption may not be valid, leading to a more general performance control problem, as discussed in Mayes et al. (2005). We assume that each (virtual) resource may host at most one component, but that a machine may host multiple virtual resources, and that, at any given time, each task may be executed by at most one virtual resource. For simplicity, we assume that any of the resources can host any of the components. We define a mode to be a mapping of all the components of the application to resources. We denote the $k$th mode $M_k$ as $M_k = \{R_{k_1}, R_{k_2}, ..., R_{k_n}\}$, where the $R_{k_i}, i=1,2,...,n$, are assumed distinct. This notation indicates that component $C_1$ executes on resource $R_{k_1}$, $C_2$ executes on resource $R_{k_2}$ and so on. Each mode is referenced by a unique mode index.

A mode sequence $MS$ (of length $s$) is given by $MS = \{M_1, M_2, ..., M_s\}$. The modes in the sequence are not required to be distinct, although the successive modes in the sequence are distinct. A change from one mode to
another requires the swapping of components between resources. Each mode in a mode sequence is maintained for a particular time duration, following which the next mode in the sequence is entered.

To illustrate mode sequences, table 1 lists the sequence of length 10 that results from the original experiment, with the threshold value of 20. Note that, out of 24 possible modes, only seven are used and three are duplicate modes; the latter implies that there must be three unnecessary swaps\(^4\) generated by the threshold-based algorithm with the particular threshold value of 20.

Let us analyse figure 3 in terms of mode sequences. Different sequences exist within different ranges of threshold values. Figure 5 includes the threshold range 47–65 associated with the four-mode sequence MS = \{6,9,19,16\}, as representative of the general behaviour reflected in figure 3.

For each threshold value in the range 47–65, the corresponding schedule (a schedule is defined to be a mode sequence together with associated execution times for each mode of the sequence) is a variant of MS = \{6,9,19,16\} with a unique distribution of execution time to modes. Within the set of schedules, there must exist at least one that will minimize the load imbalance in the final mode of the sequence, \(M_{16}\). For this range, the threshold value of 61 gives this optimal schedule (listed in table 2), thus producing a local minimum when plotted in figure 5.

\(^4\)These swaps are deemed unnecessary under the assumption of dedicated, i.e. constant, resources.

Table 1. Schedule (mode sequence & associated timings) generated where \(\theta = 20\).

<table>
<thead>
<tr>
<th>mode index</th>
<th>mode</th>
<th>time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>{R(_1),R(_4),R(_3),R(_2)}</td>
<td>54.653</td>
</tr>
<tr>
<td>9</td>
<td>{R(_2),R(_3),R(_1),R(_4)}</td>
<td>100.88</td>
</tr>
<tr>
<td>19</td>
<td>{R(_4),R(_1),R(_2),R(_3)}</td>
<td>90.645</td>
</tr>
<tr>
<td>16</td>
<td>{R(_3),R(_2),R(_4),R(_1)}</td>
<td>105.307</td>
</tr>
<tr>
<td>20</td>
<td>{R(_4),R(_1),R(_3),R(_2)}</td>
<td>111.635</td>
</tr>
<tr>
<td>9</td>
<td>{R(_2),R(_3),R(_1),R(_4)}</td>
<td>106.641</td>
</tr>
<tr>
<td>19</td>
<td>{R(_4),R(_1),R(_2),R(_3)}</td>
<td>90.847</td>
</tr>
<tr>
<td>10</td>
<td>{R(_2),R(_3),R(_4),R(_1)}</td>
<td>111.972</td>
</tr>
<tr>
<td>19</td>
<td>{R(_4),R(_1),R(_2),R(_3)}</td>
<td>111.635</td>
</tr>
<tr>
<td>15</td>
<td>{R(_3),R(_2),R(_1),R(_4)}</td>
<td>45.731</td>
</tr>
</tbody>
</table>

Figure 5. Detail of figure 3, including MS = \{6,9,19,16\} in the threshold range 47–65.
As the threshold value increases beyond 61, the application spends more time in each of $M_{6}$, $M_{9}$ and $M_{19}$ before a swap condition is reached, and the time spent executing in $M_{16}$ decreases commensurately. Eventually, at a threshold value of 66, $M_{19}$ never reaches a swap condition (because the component imbalance never reaches 66 time steps) before execution completes. At this point, $M_{16}$ becomes redundant and the lower boundary of the new range of threshold values delimiting MS=$\{6,9,19\}$ is found. The associated schedule for the threshold of 66 is given in table 3.

Table 2. Optimal schedule within the range $47 \leq \theta \leq 65$ generated where $\theta = 61$.

<table>
<thead>
<tr>
<th>mode index</th>
<th>mode</th>
<th>time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>${R_{1},R_{4},R_{3},R_{2}}$</td>
<td>165.875</td>
</tr>
<tr>
<td>9</td>
<td>${R_{2},R_{3},R_{1},R_{4}}$</td>
<td>303.694</td>
</tr>
<tr>
<td>19</td>
<td>${R_{4},R_{1},R_{2},R_{3}}$</td>
<td>282.462</td>
</tr>
<tr>
<td>16</td>
<td>${R_{3},R_{2},R_{4},R_{1}}$</td>
<td>138.402</td>
</tr>
</tbody>
</table>

Table 3. Schedule generated where $\theta = 66$, demonstrating load imbalance in $M_{19}$.

<table>
<thead>
<tr>
<th>mode index</th>
<th>mode</th>
<th>time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>${R_{1},R_{4},R_{3},R_{2}}$</td>
<td>180.472</td>
</tr>
<tr>
<td>9</td>
<td>${R_{2},R_{3},R_{1},R_{4}}$</td>
<td>327.58</td>
</tr>
<tr>
<td>19</td>
<td>${R_{4},R_{1},R_{2},R_{3}}$</td>
<td>467.705</td>
</tr>
</tbody>
</table>

As the threshold value increases beyond 61, the application spends more time in each of $M_{6}$, $M_{9}$ and $M_{19}$ before a swap condition is reached, and the time spent executing in $M_{16}$ decreases commensurately. Eventually, at a threshold value of 66, $M_{19}$ never reaches a swap condition (because the component imbalance never reaches 66 time steps) before execution completes. At this point, $M_{16}$ becomes redundant and the lower boundary of the new range of threshold values delimiting MS=$\{6,9,19\}$ is found. The associated schedule for the threshold of 66 is given in table 3.

Note that the time spent executing $M_{19}$ increases dramatically, yet no swap condition is reached. In these circumstances, the load imbalance in $M_{19}$ is maximized and the lower boundary of each threshold range corresponds to the most inefficient distribution for that particular mode sequence. This results in the sharply defined ‘sawtooth’ behaviour of figure 3.

Figure 3 reveals a number of local minima, each of which is predicted to deliver improved performance over the threshold value of 20 used in the original experiment. The result of repeating the experiment, on the same dedicated set of resources, but using the PerCoSim-predicted optimal threshold of 88, is shown in figure 6. This figure indicates a performance improvement of approximately
100 s, or approximately 10 per cent, over the original experiment. Thus, the use of PERCoSIm provides an efficient method to identify an optimal threshold value for the problem at hand.

5. A linear programming solution

Section 4 has shown that, for the given problem, the threshold-based strategy in PERCoSIm acts as an efficient heuristic to identify the best scheduling threshold value. However, the evaluation has revealed that, as each mode sequence in any range of threshold values differs from that defined in the preceding range solely by the elimination of the final mode, the strategy in fact considers only a relatively limited subset of all the possible mode sequences available. The threshold-based strategy is thus shown to generate schedules that approximately minimize the makespan. We therefore wish to evaluate the effectiveness of this strategy, and introduce an alternative (linear programming, LP-based) approach capable of identifying the optimal schedule for problem types similar to the one under consideration.

For a given mode sequence of a given length, the LP algorithm determines the optimum time to spend executing each mode of the sequence (i.e. the optimal schedule) to minimize the makespan of the overall application, but note that swap costs are ignored. (Including non-zero swap costs inevitably leads to a (more difficult) nonlinear optimization problem.) In the results given below, we deal with swap costs by adding a fixed cost per swap.

For example, for an n-component application, consider the mode sequence MS of length s given by MS = \{M_1, M_2, ..., M_s\}. We use the LP approach introduced by Bianco et al. (1995) to determine the time to be spent executing each of the modes M_k, k = 1, 2, ..., s.

For an n-component application, executing on m resources, there are t_m = m!/(n - m)! possible modes, and t_s = t_m!/(t_m - s)! possible mode sequences of length s, where we impose the restriction that no mode can be repeated in the mode sequence. We need to solve the LP problem for each of these t_s mode sequences, and select the mode sequence that gives rise to the shortest makespan. The process is then repeated for different lengths of mode sequences, in each case identifying the mode sequences that minimize makespan.

For the original experiment, the optimal schedule generated by the LP algorithm is given in table 4, and the corresponding predicted optimal run-time is 863.96 s (without swap overheads).
This schedule, when implemented in PerCoSim and parametrized with experimentally determined values for PerCo swap costs, results in a predicted application run-time of 886.11 s, and the corresponding PerCo experiment, implementing the optimal LP-generated schedule, delivers an actual application run-time of 887.07 s. The LP algorithm thus shows that, for the application scenario under examination, the threshold-based heuristic generates a best schedule (with run-time of 890.37 s) that compares very favourably to the LP solution.

6. Conclusions and future work

The use of PerCoSim has been shown to improve the performance of a PerCo experiment by optimizing an existing redistributive control strategy. We have also presented mode sequence analysis as a unifying principle for evaluating scheduling algorithms for component-based applications, and used an LP-based technique to demonstrate the properties of optimal mode sequences to maximize resource usage. In doing so, we have also shown that alternative scheduling techniques can be used in PerCo.

The extension of this work is to apply mode sequence analysis to investigate alternative scheduling algorithms as adaptive PerCo performance control policies within an online predictor framework. PerCoSim is lightweight enough to function as a real-time predictor within an executing PerCo-controlled simulation. A predictive PerCoSim will allow adaptive variants of other scheduling algorithms to be implemented and evaluated, including the many existing scheduling algorithms for applications structured as directed acyclic graphs (see Kwok & Ahmad 1999). Our primary interest here is in studying the impact of dynamic execution conditions on those algorithms to determine which are the most useful in computational Grid settings.

The work of the first author has been supported by the award of an EPSRC Doctoral Training Studentship; the work of the second author has been supported by the BBSRC e-Science project: IntBioSim (BBS/B/16011).

References


