Abstract

We here present ATLS, a self scheduling scheme designed for execution of parallel loops in dynamic environments of non-dedicated networked computers. Since grid and volunteer systems based on desktop computers are proving themselves as real and powerful alternatives for parallel computing, new scheduling schemes, better suited for these environments, are required. Our proposal, by tracking several performance change ratios at runtime, is able to properly adjust the load distribution using no prior information of the loop features nor the involved processors. The results obtained during the experiments performed to validate ATLS show that it is possible to improve former contributions of well-known parallel loop scheduling schemes in dynamic environments. The implementation of the scheduler has been done for the R language but, as it is exposed, it can easily be adapted to any other language and parallel loop based application.

Keywords: parallel loops, self scheduling, desktop computing, dynamic loop scheduling

1. Introduction

The proliferation of multicore computers in desktop environments is introducing important changes in the way regular PCs are used to run applications. Since increasing the number of available processing units per processor is the current trend to increase computer performance, applications are being adapted with parallel technologies to take advantage of the new performance improvements. Additionally, as a consequence of the improved capacities of desktop computers, even in office environments, the aggregated computing power can provide similar performance levels that years ago were restricted to dedicated clusters, and that it is a tendency that over the coming years will become more evident with the increase of core densities per processor. In order to effectively take advantage of such aggregated computing power current and new methods have to be adapted and developed.

In our case, given that the most common source of parallelism in scientific applications comes from parallel loops we focus our work in this family of problems. A loop is called a parallel loop if there is no data dependency among all its iterations, i.e. any iteration can be processed in any order or even simultaneously.
Our contribution, presented here, is a new scheduling scheme called ATLS, which stands for Adaptive Turnaround-based Loop Scheduling, and is specifically designed for processing parallel loops in dynamic environments made up of volunteered computers, for example desktop computers, where its number and capacities are unknown before performing a calculation and, given the resources are not dedicated, can change in quantity and quality at any time during the computation. The implementation has been done extending the statistical language R [1], to enable the distributed execution of parallel loops in dynamic environments. This language has been selected since their users, specially the bioinformatics community, usually perform the same operations over very large sets of samples, making parallel loops a perfect match for our proposal. Nevertheless, our contribution and the results obtained during the experiments are exposed independently of the R language, enabling anyone with similar requirements and interest for parallel loops, and independently of their language of choice, to take advantage of our work.

The rest of the paper is organized as follows. Section 2 reviews the analyzed parallel loop scheduling schemes. Section 3 shows the design and implementation details of ATLS. Section 4 provides an experimental evaluation of our proposal. Finally, the insights into future research and conclusions are given in section 5.

2. Background

The problem of scheduling parallel loops in a parallel system has been extensively studied in the literature [2–7]. The common approach is to set up tasks with subsets of N iterations in chunks of different sizes which are dispatched to P processing nodes. The objective is to distribute conveniently the workload, i.e. all the iterations, among all the processors to minimize the overall execution time. In order to define a balanced distribution of the workload, several scheduling methods can be selected. These methods can be classified into two classes, accordingly to the time when the required information to define the distribution is obtained and used.

2.1. Nonadaptive scheduling schemes

Nonadaptive scheduling schemes define chunk sizes (K) based on the available information before running the loop, usually at compile time. These chunk sizes can be fixed for the whole computation or updated at runtime based on predefined rules.

Static scheduling (STATIC), suitable for homogeneous environments with fixed time iterations, generates chunks of size $K = N/P$ and has low overhead. Pure self-scheduling (PSS) generates chunks of just $K = 1$. This method achieves load balancing but also a likely high overhead depending on the communication required among processors. A variant called chunk self-scheduling (CSS) allocates a group of $K$ iterations, which is a constant that must be specified by the programmers. A high value of $K$ is likely to cause load imbalance and a small value can eventually produce too much overhead. Fixed size chunking (FSC) provides an optimal constant value for the chunk size that minimizes the execution time on homogeneous and equally loaded processors once all its parameters have been defined, which includes the standard deviation of the iteration time and the scheduling overhead. Defining these parameters can be complicated for some programmers. These schemes provide fixed chunk sizes suitable for homogeneous environments but in many cases its results are not satisfactory for heterogeneous environments.

Guided self-scheduling (GSS) [4] based methods can dynamically change the value of $K$ by generating decreasing size chunks. Their approach is to dispatch large chunks at the beginning of the calculation to reduce the overhead and gradually reduce the chunk size to align the finalization time of the involved processors. Trapezoid self-scheduling (TSS) [8] is a variation where chunk sizes decrease linearly, in contrast to the geometric decrease of chunk sizes in GSS. Based on the parameters $f$ and $l$ specified by the programmer, $TSS(f, l)$ distributes all the iterations, starting with a chunk of size $f$, and decreasing its size linearly until the last chunk of size $l$. Factoring self-scheduling (FSS) [3] proceeds in phases. During each phase, only a subset of the remaining iterations, i.e. a batch $B$, is divided equally among the available processors. The batch size is a fixed ratio of the unscheduled iterates. The ratio depends on the mean and standard deviation of the iteration execution times. When these statistics are unknown, the ratio 0.5 has been experimentally proved to provide reasonably good results. The next phase starts once all the chunks in the current batch are scheduled. Weighted factoring (WF) [5] incorporates, when available before the execution of the loop, information about the processing speed of the working nodes. In that method, the chunk size assigned to each
processor $i$ is readjusted accordingly to its relative speed, defined as a weight $w_i$ which is used as follows:

$$K_{WF} = w_i \times K_{FSS}$$

Using this method, faster processors obtain bigger chunks than slower ones. This scheduling scheme is suitable for heterogeneous environments since it adjust the assigned portion of the workload to each working node based on its processing capacity. However, it assumes that the relative speeds are constant throughout the execution of the loop, what can not always be assured.

In order to react against the changes originated within the worker nodes and its external environment that degrade the performance obtained with these nonadaptive methods, adaptive scheduling schemes have been proposed.

2.2. Adaptive scheduling schemes

Adaptive methods described in literature mainly evolve from the factoring scheduling schemes. Adaptive weighted factoring (AWF) [6] was originally developed to tackle with changes at runtime, not only from the processing speeds but also from the parallel loops themselves. AWF attempted to incorporate both sources of variability when determining the chunk sizes. Initially, chunk sizes are determined as in WF, but at the end of each time step, the processor weights $w_i$ are adjusted based on the information collected during the current and previous steps. Adaptive factoring (AF) [7] modifies the FSS method by using an estimation of the mean and standard deviation of the iteration times, which is updated dynamically at runtime.

Next section describes our proposal of a new parallel loop scheduling scheme for dynamic environments, ATLS.

3. Design and Implementation details

Since the R language interpreter is single thread and does not provide natively any mechanism for parallel computing, an R package must be used to extend its capabilities. In our case we have selected the R package R/parallel [9]. Although other packages exists that provide parallel support to R [10] like Rmpi [11], rpvm [12] or SNOW [13], we have selected R/parallel since its design and implementation is completely focused on parallel loops (running in multicore computers), what makes its adaptation for distributed computing more straightforward.

The package R/parallel implements a master-worker architecture. The master component runs on the client side, where the R script with the parallel loop is going to be run. Once a new computation begins, the master component requests the collaboration to a list of known volunteered computers where a listener has been previously installed. These computers, if possible, will answer back, requesting a task to collaborate in the calculation. Under the instructions of the selected scheduler, the master distributes the iterations among the worker nodes which later, once they have finished their portion of work, will return back the partial results and if pending will obtain an additional task with new iterations. Once all the iterations have been processed, the master reassembles all the partial results in final ones as in a sequential execution were obtained.

3.1. ATLS: Adaptive Turnaround-based Loop Scheduling

The reasons that have motivated each design decision are explained next, together with the details of our proposal, the scheduling scheme ATLS.

Since the quantification of the computing power with a positive constant is still an open problem [14], performance indicators based on intrinsic characteristics of computers, such as processor frequency, can not completely be trusted. Performance results based in predefined metrics which are true for a given application and computer can change if any of these elements is also changed. Moreover, when running on non-dedicated environments, there are multiple factors that have a negative effect over the raw computing power. These factors range from network congestion to computer overloading. Due to that, it is quite feasible that at some instants, relatively small nodes perform faster than more capable ones. For these reasons we have chosen the task turnaround completion time as the only performance indicator we can really trust.

The performance ratio is based on the task turnaround completion time, and is calculated as follows: $\nu = \frac{T}{K}$, where $T$ represent the time elapsed since the task was scheduled to a worker until it was successfully completed and returned back. The variable $K$ represents the number of iterations assigned to that task, i.e. the chunk size. The average
performance ratio, $\nu_{avg}$, given $n$ finalizations and the current performance ratio $\nu$ is updated as shown in (1). Also the minimum $\nu_{avg}$ is kept for each processor for further calculations in the variable $\nu_{min}$. Our proposal to mitigate the negative effects of unpredicted changes in the worker node performance is the utilization of a system-wide confidence indicator, $CI$, based on the changing ratios of the performance index of each participating node. The initial value of the confidence indicator, like any of the following variables except if stated otherwise, is $CI_0 = 0$, since all nodes are unknown at the beginning and therefore untrusted. As the computation evolves, its values are updated as follows:

$$CI_n = CI_{n-1} + w_i \cdot (C_r - CI_{n-1})$$

Here two new variables are introduced. The relative weight of the worker node $i$, $w_i$, and the current system changing ratio, $C_r$. The idea is to update the previous value of $CI$ with the difference between the previous $CI$ and the current system changing ratio, adjusted with the relative weight of worker node. By doing so we adjust the system variable $CI$ only with the proportional contribution of each worker. The value of $w_i$ and $C_r$ is is updated whenever a working node returns successfully a job (ontime) or when its finalization deadline is reached (overdue) as follows:

$$\nu_{avg} = \frac{\nu + \nu_{avg,j} \cdot (n-1)}{n}, \quad w_i = \frac{\nu_{avg,j}}{\sum_{j=1}^{p} \nu_{avg,j}}, \quad C_r = \frac{\sum_{i=1}^{p} \nu_{avg} \cdot \alpha_i}{\sum_{i=1}^{p} \nu_{avg}}$$

(1)

The variable $\alpha_i$ represents the current changing ratio for the processor $i$ while the variable $\alpha_{avg}$ represents its average changing ratio and it is updated at each finalization event, positively when a task is returned ontime or negatively when its estimated finalization time, i.e. its estimated deadline, is reached. The calculation of $\alpha_n$, at the $n$th time, is as follows:

$$\alpha_n = \begin{cases} \alpha_{n-1} + \frac{1}{j} & \text{if a task is finished} \\ \frac{1}{j} \alpha_{n-1} & \text{if a deadline is reached} \end{cases}$$

The event when a task is done is worker driven. To estimate the finalization time for a task, a deadline $\lambda_i$ is defined at the time of dispatching a task to the worker $i$. In an utopic scenario, knowing the real performance ratio $\nu_{real}$, the constant overhead of the system $h_{ct}$, the overhead per iteration $h_{iter}$ and the future delay possibly introduced by the environment, also constant $\epsilon_{ct}$ and per iteration $\epsilon_{iter}$, the ideal finalization time $\lambda'_i$ can be calculated as follows:

$$\lambda'_i = (\nu_{real} \cdot K) + (h_{iter} + \epsilon_{iter}) \cdot K + h_{ct} + \epsilon_{ct}$$

However, except for the value of the chunk size $K$, for the other variables, given the variable nature of the dynamic environments, it is not possible to make any safe estimation of their values. Our proposal to overtake this problem and calculate the finalization time $\lambda_i$ is based on the following equation:

$$\lambda_i = (\nu_{avg} \cdot K) + (\xi_i \cdot K) \cdot 0$$

Given the method we use to calculate the performance ratio, it is clear we are introducing an error by including in this ratio a fraction that corresponds to the system overhead and other external factors originated in other elements of the environment with no direct relation with the ongoing calculation. Nevertheless, as discussed previously, we don’t have any other observation to support a different performance ratio. But, we can also observe the differences between the estimated deadlines and the real elapsed times. Any time a task finishes, before or after its deadline, the time difference, positive or negative, corresponds either to the system overhead or the external factors, for the current or the previous performance ratio values, and can be recovered and kept aside for further estimations. The variable $\xi_i$ is defined as the burden ratio. With this variable we maintain the maximum ratio of extra time per iteration observed during the ongoing calculations. We assume that this extra time, until a higher value appears, can be used to accomodate the estimations of future completion times and, as a consequence, we avoid to declare a scheduled task as lost before it is really needed. Its values are calculated, based on the finalization time, as follows:

$$\xi_n = \begin{cases} \frac{T_{\text{end}} - T_{\text{start}}}{K} & \text{before } \lambda_i(\text{ontime}) \\ \nu - \nu_{\text{min}} & \text{after } \lambda_i(\text{overdue}) \end{cases}, \quad \xi_i = \begin{cases} \xi_i' & \text{if } \xi_i' > \xi_i \\ \xi_i & \text{if } \xi_i' \leq \xi_i \end{cases}$$
Before we can define the equations to obtain the batch and chunk sizes, as also used in the AWF method, we still have to describe an additional concept to understand our proposal, the relative capacity of the system at a given time $n$ or $RC_n$. This variable represents the number of iterations that can simultaneously be performed by all the available working nodes within a time-frame so all nodes finalize at the same time. The basic idea is that, assuming there are enough iterations for all the available nodes, in order to obtain the best efficiency, all the available resources must be used at the same time. Therefore, at least 1 iteration must be assigned to the slowest node. During the time-frame required for the slowest node to process its assigned iteration, the rest of the working nodes are able to perform one or more iterations. $RC_n$ represents therefore the sum of the iterations each worker is able to perform during the time-frame defined by the slowest node. The batch $B$ and chunk size $K$, at the $n$th time, can be defined as follows:

$$B_{ATLS_n} = \left\lfloor \frac{\text{remaining}}{RC_n} \right\rfloor \ast C_{I_n} \ast \frac{\#\text{workers}_{active}}{\#\text{workers}_{expected}}, \quad K_i = \begin{cases} 1, & \text{1st time} \\ 2, & \text{2nd time} \\ \left[B_{ATLS_n} \ast \left(\frac{\text{available}}{v_i}\right)\right], & \text{otherwise} \end{cases}$$

The last variables introduced with the fraction $\frac{\#\text{workers}_{active}}{\#\text{workers}_{expected}}$ are used to limit the total number of scheduled iterations by the proportion of active workers. By doing so, given that the workers after the request for collaboration arrive at unknown times, we reserve a portion of work avoiding to schedule too many iterations before all the workers have answered to the request for collaboration.

Next section describes the experimental results obtained during the validation of our proposal.

4. Experimentation

The experiments have been performed in a controlled scenario made up of 5 identical computers connected by a gigabit network. Each computer has 2 quad-core processors providing a total of 8 processing units and 16 GB of RAM. The operating system installed is Red Hat Enterprise Linux Server release 5 and R version 2.10.0. One computer is used as the master component, from where the user launches his or her programs, while the other four are used as the volunteered computers with the worker components, providing a total of 32 worker nodes.

The program run at the experiments is an R script which, besides of the R/parallel package, it also uses the package Biostrings from the software project for the analysis and comprehension of genomic data Bioconductor [15]. The script loads a large set of DNA sequences (from ~1000 to ~100000 sequences) and given a set of provided labels, it identifies the individual donor of each sequence and clips the original sample sequence. Since each operation is performed independently from the others and there is no dependency between each other, it can be performed using a parallel loop. During the experiments several parameters have been modified to evaluate its influence on the design of the proposed scheduling scheme, basically: the number of iterations, the number and type of workers and the schemes set up at the scheduler.

4.1. Static Environment with Homogeneous Workers

Although ATLS has been designed for dynamic environments, first we must determine that its performance is satisfactory also in static environments. The test-bed described without modifications serves our requirements. Figure 2 shows the obtained total execution time and its corresponding speedups for different setups. As it was expected for an environment without changes, the simple scheme STATIC provides the best performance. In that case the schemes AF and AWF, designed for dynamic environments, provide respectively an average result 10.36% and 8.88% worse than STATIC while for ATLS the difference is smaller with a result of 4.68%. FSS and WF, best fitted for static environments, obtain results 2.93% and 3.16% worse than STATIC, performing slightly better than ATLS in that case. Finally, it can be observed that the efficiency (speedup/number of workers) is very similar for ~10000 and ~100000 iterations, decreasing down to ~50%. Taking that into consideration, next experiments are focused in using 32 workers and ~100000 iterations.
4.2. Static Environment with Heterogeneous Workers

Here we evaluate the results using different performance ratios per worker. To achieve this the program `cpulimit` [16] is used to limit the maximum percentage of share of processor a worker process receives when running. Using this tool the processor usage has been limited for the 32 workers with the following values: 8x 90%, 8x 75%, 8x 25% and 8x 10%. To simplify the comparison of results, PSS and FSS have been discarded given its results does not provide additional information for our comparison. The rest of the setup is maintained from the previous experiments.

The scheme AWF has shown the best performance in a static environment with heterogeneous workers, closely followed by WF (0.47% worse). With a 10.23% worse than AWF, STATIC is penalized for assigning the same task size to non-equal workers. For the case of AF the results are similar, a 10.31% worse. The main reasons for this behaviour are observed in Figure 2. First, almost all workers are processing ∼3000 iterations, like in the case of WF, indicated with an orange box. However, for AF, indicated with a red square, an additional reason is indicated. There is one worker with almost the double of iterations assigned. Since there are 8 workers with the highest limitation and only one has received such task, we can deduce the decision of scheduling such amount of iterations is not performance related. In fact, it is related with the sequence in time at which workers arrive and start its collaboration with the master. AF is too eager to schedule most of the iterations at the beginning, before all workers arrives, and schedules bigger tasks assuming no more workers are expected. The result is a worker that delays the computation.

For the case of ATLS, it performs on average just a 2.06% worse than AWF in this experiment. Although, as it can be observed in green boxes, the scheduler with ATLS has correctly identified two classes of workers (i.e. with high and low CPU limits), for static environments, the AWF scheme, less sensitive, ends up with better results.

4.3. Dynamic environments

Next we evaluate the results in different environments where the performance of workers varies during the computation. Again we use the `cpulimit` tool, but this time programming not only the share of CPU but also the instant in time at which the restriction will start and end. By doing so we are able to, modifying dynamically the worker performances during the computation, emulate the external interferences that eventually restrict their processing capacities, quite common in shared multi-processing computers. Three environments have been set up: step-down, step-up and mixed. At the first environment, step-down, all workers start at the same full capacity, but after 240 seconds their performance is limited during 900 seconds with the following values: 8x 50%, 8x 25%, 8x 10% and 8x 5% (number of workers x limitation). For the second, step-up, the opposite limitation have been programmed. All workers start
with the same restrictions, but after 240 seconds the processes run at full power. For the third one a mixture of behaviours have been programmed (see Figure 3). This includes one machine with 8 step-down type limitations, one with 8 step-up type limitations, one with 8 pulse type limitations (i.e. concatenations of steps) with constant limits during 120 seconds per pulse and a last one with 8 pulse type limitations with variable limits.

The results after several runs can be observed in Table 1. For the step-down experiment the best result is obtained by the scheme AWF, followed by ATLS with a result 4.33% worse. As expected, the worst result is obtained by STATIC, performing a 9.04% slower. For the step-up case, the best scheme is ATLS followed by AWF which performs a 4.43% slower. Finally, for the mixed environment, the best result is obtained again by ATLS followed by AWF, but this time a 14.09% slower for this experiment.

To better understand these results we can observe Figure 4. Marked with an orange box it is indicated the first main batch of tasks. By observing the height of the box and its vertical position we can guess how eager is each scheme. STATIC dispatches almost all the iterations at the first round with equal sizes (there is one task left due to rounding adjustments). AF also schedules most of the tasks too early. At the other side, ATLS is more conservative at the beginning and schedules considerably less iterations per task with a wider distribution of the number of tasks. In the middle we found WF and AWF with similar distributions. The purple arrows give us another indication. Its length shows the proportion of time of each individual experiment where the scheduler is just waiting, since all the iterations have already been dispatched. This is important since the sooner the scheduler runs out of iterations the sooner it will lose its ability to react against environmental changes. For this indication, it can be observed that ATLS is the scheme that more delays the dispatch of the last tasks, resulting in a better ability to adapt to unexpected changes.

Table 1: Average $\mu$ and percentage of standard deviation $\%\sigma$ of the total execution times obtained during the experiments.

<table>
<thead>
<tr>
<th>Scheme</th>
<th>Step-Down $\mu$</th>
<th>Step-Down $%\sigma$</th>
<th>Step-Up $\mu$</th>
<th>Step-Up $%\sigma$</th>
<th>Mixed $\mu$</th>
<th>Mixed $%\sigma$</th>
</tr>
</thead>
<tbody>
<tr>
<td>STATIC</td>
<td>1543.27</td>
<td>0.24%</td>
<td>1005.58</td>
<td>0.62%</td>
<td>1432.10</td>
<td>0.62%</td>
</tr>
<tr>
<td>WF</td>
<td>1483.58</td>
<td>3.27%</td>
<td>1025.18</td>
<td>8.97%</td>
<td>1203.00</td>
<td>1.55%</td>
</tr>
<tr>
<td>AWF</td>
<td>1415.35</td>
<td>4.36%</td>
<td>988.77</td>
<td>0.72%</td>
<td>1197.37</td>
<td>1.15%</td>
</tr>
<tr>
<td>AF</td>
<td>1521.38</td>
<td>0.85%</td>
<td>1213.52</td>
<td>3.05%</td>
<td>1746.67</td>
<td>15.23%</td>
</tr>
<tr>
<td>ATLS</td>
<td>1476.66</td>
<td>0.53%</td>
<td>946.25</td>
<td>1.13%</td>
<td>1049.53</td>
<td>1.46%</td>
</tr>
</tbody>
</table>
Finally, the green ellipses indicates the number of tasks finalizing close to the end. In a perfect situation all the scheduled tasks should finalize exactly at the same final time so we ensure all the available nodes are effectively used. ATLS shows the highest density of tasks finalizing close to the end while STATIC and AF waste a considerable amount of time waiting for the last task.

As the contents of this section have showed, the proposed scheduling scheme, ATLS, while providing satisfactory results in environments with none or low frequency of changes, is better suited for dynamic environments with a high frequency of changes than the others schemes analyzed. It is also worth to mention that the scheme AWF, although not as well suited as ATLS for highly changing environments, it also provides very good results in dynamic environments, in some cases even better than ATLS. Nevertheless, the experimental results show that ATLS provides good performance figures in all the scenarios evaluated, outperforming the others in dynamic environments like the one used at the last experiment due to the reasons exposed in this section.

5. Conclusions and Future Work

In this paper we have described in detail a proposal of a new scheduling scheme for parallel loops, ATLS, which as the experiments show provides, in average, better results than previous well-known contributions in dynamic environments of non-dedicated computers. The design of the scheme, since it is only based in the information obtained monitoring the turnaround completion time of each task scheduled to each worker, allows its implementation in any running platform. However, still better performance can be obtained. In case an external source of information from where insights about future events was available (e.g. the queue of a job scheduling system), it would be possible to react beforehand against incoming changes. Additionally, as it has been observed, before a distributed computation finishes there is a fraction of time where all the iterations have been scheduled and slower nodes must be awaited. A rescheduling component can be added to take advantage of the idle nodes which eventually can process the pending iterations before the slower nodes finish. The same rescheduling component can also be used to provide fault tolerance to the system and handle faulty nodes. Our future work will be focused in this direction.
Appendix A. Acknowledgments

This research has been supported by the MEC-MICINN Spain under contract TIN2007-64974.

Appendix B. References


