Hierarchical resource management in grid computing

Korkhov, V.V.

Publication date
2009
Document Version
Final published version

Chapter 5. User-level scheduling of multi-job applications

5.1 Introduction

This chapter explores the workload and resource management for the layer of complex distributed application hierarchy corresponding to multi-job applications. Here we apply the Adaptive WorkLoad Balancing (AWLB) method presented in Chapter 3 to multi-job applications with divisible workload corroborating the technology independence of this method. We present a hybrid resource management environment, operating on both application and system levels, developed for minimizing the execution time of parallel applications with divisible workload on heterogeneous Grid resources. Compared to the solutions developed for parallel applications in the previous chapters, the approach presented here brings new possibilities and features. Integrated resource management on application and system levels enables more fine-tuned workload distribution, replacement of the workers between processing iterations and intelligent resource selection.

In this chapter we suggest an approach to the integration of a User-Level Scheduling (ULS) environment with the AWLB [66, 68, 73]. The AWLB ensures optimal workload distribution based on the discovered application requirements and measured resource parameters. The ULS controls the user-level resource pool, enables resource selection and controls the execution. The benefits of the integration are twofold: the AWLB is enriched with the capability to select resources most suitable for the application, and the ULS environment is equipped with an advanced strategy to optimize resource usage. The optimization of the workload performed by the AWLB is adaptable to the resource characteristics (CPU power, memory, network bandwidth, I/O speed, etc.) and to the corresponding application requirements. The ULS environment acquires the most appropriate resources to the user-level resource pool, and the AWLB controls the workload distribution. We perform the studies using a model application with tunable characteristics (communication to computation ratio, memory usage, communication topology, etc.) implemented in DIANE ULS environment [43, 88]. We analyze the results of performance comparison of default self-scheduling algorithm used in DIANE with AWLB-based scheduling, evaluate dynamic resource pool and resource selection mechanisms applied to it, and examine dependencies of

This chapter is based on: V.V. Korkhov, J.T. Moscicki, V.V. Krzhizhanovskaya "Dynamic Workload Balancing of Parallel Applications with User-Level Scheduling on the Grid" Future Generation Computer Systems, 25 (2009), pp. 28-34

application performance on aggregate characteristics of selected resources and application profile.

5.2 Integrated adaptive workload balancing and user-level scheduling environment

5.2.1 User-level scheduling features

Large Grid infrastructures, such as EGEE Grid [40], provide access to the computing resources at unprecedented scale. Designed for high-throughput applications, the Grid middleware and infrastructure comes with little support for the high-performance use-cases, especially the ones using a set of heterogeneous resources for a single application. Submitting, scheduling and mapping tasks on the Grid can take up to few orders of magnitude more time than the execution [43]. This is especially true for low-latency and short-deadline scenarios which are pervasive in a number of application domains from medical applications to physics data analysis. User-level scheduling (ULS) is one of the most promising ways to eliminate the difference in scale between short execution times and the large grid middleware latencies. The ULS system contains application-specific knowledge therefore it may provide customized resource selection and control mechanisms - a feature indispensable for enabling high performance applications on the Grid.

User-level (or application-level) scheduling is a virtualization layer on the application side. Instead of being executed directly, the application is executed via an overlay scheduling layer (user-level scheduler). The overlay scheduling layer runs as a set of regular user jobs and therefore it operates entirely inside user space. These overlay jobs are processed in standard queues of the resources and after being executed they provide immediate access to the actual applications. This approach is especially beneficial for the jobs with short execution times (the latency of waiting in the queue can exceed the execution time a lot) or for series of jobs, particularly in the case of jobs with input parameters that depend on the execution of the previous job in the series (which means that all the jobs can not be queued at the same time, and in the standard situation each job repeats the period of waiting in the queue on the resource). User-level scheduling does not require any modification of the Grid middleware and infrastructure nor the deployment of special services in the Grid sites, it provides immediate exploitation of the full range of a Grid sites which are available for a given user [44]. The only constraint is that actual applications have to be instrumented with additional functionality to support scheduling on the user-level.

Parallel data processing is often organized not within a single parallel application but in a set of separate programs that perform information exchange only at the start and finish of the execution or after each iteration in case of iterative execution. We call this type of software a *multi-job application*, and each job can be a separate parallel application in turn. Typically multi-job applications operate on a divisible workload: the amount of data that has to be processed by all the jobs can be shared in an arbitrary way between the jobs. Classical examples of plain multi-job applica-

tions in distributed environment are developed in the projects of distributed.net[131], SETI@home[138] etc. Here an enormous amount of workload is divided by portions between multitude of workers spread all over the world. Each worker processes its portion of the workload independently, reports the results back to the central server and gets a new portion. Usually the portions of the workload have fixed size for all the workers, and the speed and simultaneity of the processing does not play a significant role as all the initial workload should be processed only once. The situation changes when the results of processing generate new data which form new workload that should be processed again. Thus all the workload is generated iteratively, layer by layer. In this case the synchronization between different workers is needed, as the new workload can be formed only when the results of the previous iteration of processing is gathered from all the workers. From the application point of view the efficiency of distributed computations is characterized by minimization of the time taken for the execution. This demand to minimize the execution time of the iterative type of multi-job applications is met when all the workers finish processing of their portion of the workload from the same workload layer at the same time. This requires approaches and methods that ensure the simultaneity of data processing by the control of the data portion sizes assigned to different workers according to their capabilities.

Parameter sweep applications can be also mentioned in this context if the parameter space under question is considered as the shared workload. In this case different parameters are distributed between the workers that report results back to the master to analyze if the search can be optimized and some parameter space regions can be excluded from the processing. An example of parameter sweep environment is Nimrod [136], and one of the core differences of our approach with parameter sweep applications is the difference in the goal. Parameter sweeps analyze the intermediate execution results to select the regions of parameter space that can be excluded from further processing, i.e. the workload is modified during run-time, without explicit attention to load balancing in case of heterogeneous resources. Our main concern is to ensure efficient execution of the existing workload in a heterogeneous environment while the workload is not modified.

5.2.2 Executing applications in the user-level scheduling environment on heterogeneous resources

Efficient execution of parallel applications on heterogeneous and dynamic Grid resources is a challenging problem that requires the development of adaptive workload balancing algorithms that would take into account the application requirements (initially unknown often) and the resource characteristics. Generally studies on load balancing consider distribution of processes to computational resources on the system/library level with no modifications in the application code [9, 55, 56]. Less often, load balancing code is included into the application source-code to improve performance in specific cases [102, 104]. Some research projects concern load balancing techniques that use source code transformations to speedup the execution [27]. In the proposed integrated system, a hybrid approach is employed, where the balancing

decision is taken in interaction of the application with the execution environment.

A number of semi-automatic load balancing methods have been developed (e.g. diffusion self-balancing mechanism, genetic networks load regulation, simulated annealing technique, bidding approaches, multi-parameter optimization, numerous heuristics, etc.), but most of them suffer one or another serious limitation, most noticeably the lack of flexibility, high overheads, or inability to take into consideration the specific features of the application. Moreover, all of them lack the higher-level functionality, such as the resource selection mechanism and job scheduling. By developing a hybrid resource management environment, we make a step forward towards efficient and user-friendly Grid computing.

One of the approaches to create a parallel implementation of a scientific application is to build a single parallel program that is executed on a set of resources and controls the proper distribution of the workload itself. In chapter 3 we discussed an adaptive load balancing algorithm for such type of applications and validated its MPI implementation [66, 68]. Another possibility is to share the responsibility for load balancing between the application itself and the environment that enables its execution on heterogeneous resources. This approach has the advantage of combined resource management on system and application level: the environment selects and acquires the resources according to application requirements while the application controls workload distribution on these resources [73]. The application consists of a set of parallel jobs that process the workload scheduled by the Master tightly connected to the User-level scheduling environment which is an intermediate layer between the system resource manager and the application. It operates the information about available resources combined with application specific information. For iterative simulations, each iteration is performed by a new set of computational jobs; and new more suitable resources can be used to improve the application performance. This is a crucial distinction of this approach compared to traditional parallel programs where resources are allocated once and fixed during the execution, i.e. cannot be replaced during run-time unless special migration libraries are used (e.g. Dynamite [55]). To support the replacement of resources during the application run-time, the concept of user-level adaptive resource pool is employed.

The user-level resource pool contains all the discovered and acquired by ULS environment resources suitable for the application. During the execution of the application the pool is periodically updated, resources can be added or removed. The suitability of resources is determined by the application requirements, and for traditional parallel computing applications it depends on the processing power and network connectivity correlated with the application characteristics.

After resources have been assigned to all the jobs, proper distribution of the workload to each job is necessary to eliminate possible load imbalance. The goal is to finish the execution of all jobs simultaneously. The distribution of the workload depends on resource properties and application characteristics; the method to assign the workload to the worker nodes (adaptive workload balancing algorithm, AWLB) is described in Chapter 3. The computation is performed as an iterative process; after each iteration the distribution of the workload is re-evaluated on the updated resources available, and the AWLB parameters are re-estimated.

To evaluate AWLB in ULS environment we modeled the Virtual Reactor application described in Chapter 3 as a multi-job application. Now the workload is processed not in a traditional parallel environment (MPI implementation) but by a set of independent jobs that receive portions of workload iteratively. Now each job receives a number of beams to process (see Figure 3.2 in Chapter 3), and reports back the updated values of data in the beam cells. New generated beams of computational cells form the new layer of the workload which is divided between the workers for further processing. The size of the data transferred between the workers and the master can be significant, the capabilities of the workers can be different and the workers can be replaced between the iterations - the significant new feature provided by ULS environment compared to the MPI implementation of the Virtual Reactor.

5.2.3 Adaptive load balancing algorithm with resource selection in the user-level scheduling environment

The outline of the integrated solution for adaptive workload balancing in ULS environment is presented in Fig. 5.1 as a meta-algorithm based on the concepts developed in [73]. All the processing is performed in the framework of the ULS environment; two levels are distinguished: User-level resource pool and Application level. ULS environment is responsible for managing the resource pool to supply the application with appropriate resources. In turn, the application updates the environment with changing requirements. The explanation of the steps illustrated in Fig. 5.1 is provided below.

Resource pool level: In parallel to the application execution the resource pool is being monitored and updated by the ULS environment.

- Step R1. Update the pool: Discover available resources using Grid information services, acquire them to the pool if they meet application requirements. Check the resources in the pool for availability, remove no longer available ones.

- Step R2. Benchmark resources: Measuring the computational power and memory available on the worker nodes in the resource pool, network links bandwidth, hard disk capacity and I/O speed. In a more generic sense of "resources", some other metrics can be added characterizing the equipment and tools associated with a particular Grid node.

- Step R3. Rank resources: Update and re-order list of acquired resources (used by the resource selection procedure in Steps A2, A3). The priority of ranking parameters is dependent on the type of application. For traditional parallel computing solvers the first ranking parameter is the computational power (CPU) of the processor, the second parameter being the network bandwidth to this processor. For memory-critical applications, memory is the top-priority metric. For a large emerging class of multimedia streaming applications, the network bandwidth and the disk I/O speed would be the key parameters. In most cases memory ranking is an essential complimentary operation, since available memory can be a constraining factor defining if the resource can be used by the

application or not. The same goes for the free disk space parameter that can constrain the streaming applications that damp data on hard disks.

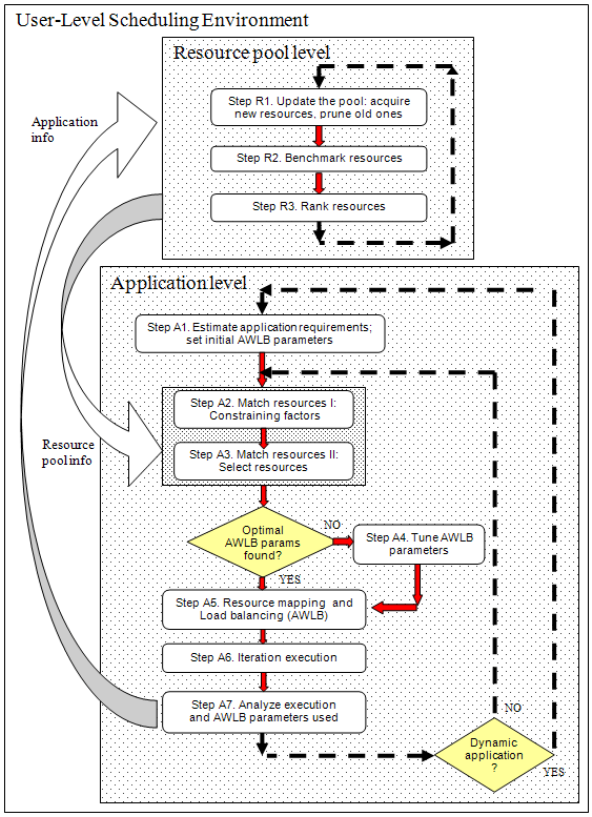

Figure 5.1: Schematic view of iterative execution of parallel application using adaptive workload balancing algorithm in ULS environment with dynamic resource pool. Resource pool level: in parallel to the application execution the resource pool is being monitored and updated by the User-Level Scheduling environment. Application level: the load balancing, resource matching and task mapping is performed on this level, in coordination with the Resource pool level.

Application level: The load balancing, resource matching and task mapping is performed on the application level, in coordination with the Resource pool level.

- Step A1. Estimate application requirements; set initial AWLB parameters: Application requirements are used to set initial values for AWLB parameters (see Section 3.3.2); additional constraints are set (e.g. minimal memory required). The AWLB parameters are automatically tuned during runtime (Step A4).

- Step A2. Matching resources I. Constraining factors: This is the first stage of checking the suitability of the available resources to the given application. It is based on the analysis of the results of Steps R2, R3 and A1 (or A7 in iterative process). In our computational application example, memory can be the constraining factor: in case of insufficient memory on some of the processors, they must be disregarded from the computation. Steps A2,A3 use information from the resource pool and send back the application requirements and requests on chosen resources. The requests are then processed by the ULS environment to book the resources.

- Step A3. Matching resources II. Selecting resources: This step provides the means to select the best-suited resources for each of the computational tasks. It consists of two basic functionalities: finding an optimal number of processors and the actual resource matching. The resource matching procedure (to be distinguished from process mapping) shall take into account the application requirements derived in Step A1. The suitability of a resource is strongly dependent on the application characteristics: the communication-bound applications will achieve a better performance on faster links even with slower processors, and the computation-intensive applications will not care about the network bandwidth. Another question is how many worker nodes shall be assigned to a multi-job application. The answer depends on the application characteristics again: for a majority of embarrassingly parallel applications (employing the resource farming concept), the more processors the better. On the other hand, for a wide class of typical parallel applications (characterized by a speedup saturation with a growing number of parallel processors), an optimal number of processors can be estimated based on the measured resource parameters and the application fractional communication overhead.

  Steps A2, A3 use information from the resource pool and send back the application requirements and the requests on chosen resources. The requests are then processed by the ULS environment to book or release the resources.

- Step A4. Tune AWLB parameters: The AWLB parameters are tuned based on the execution analysis Step A7 to provide better workload distribution. Being an adaptive heuristic, AWLB requires several steps of computations to estimate optimal values of the parameters on a given resource set. If the resources change between the iterations, re-estimation of AWLB parameters is required.

- Step A5. Resource mapping and load balancing: Actual optimization of the work-load distribution within the parallel tasks is performed, i.e. mapping the processes and workload onto the allocated resources. This Step is based on the Adaptive Workload Balancing Algorithm (AWLB) described in Section 4. It includes a method to calculate the weighting factors for each processor depending on the re-source characteristics measured in Steps R3-R5 and application requirements estimated in Step A1, A7.

- Step A6. Iteration execution: Perform an iteration of calculations and data

exchange with the workload distribution defined in Step A4.

- Step A7. Analyze execution and AWLB parameters: Measure the execution time of one iteration (or simulation time step) with current AWLB parameters. The idea is to quantitatively estimate the requirements of the application based on the results of resource benchmarking (Step R2) and measurements of the application response. This Step measures the application performance on a given set of re-sources. The performance information is used to re-estimate AWLB parameters according to the latest iteration performance data (Step A4).

  In case of dynamic resources where performance is influenced by other factors, a periodic re-estimation of resource parameters and load re-distribution shall be performed. This leads to repeating all the meta-algorithm Steps except of Step A1 (see the dashed arrows in Figure 5.1). If the application is dynamically changing (for instance due to adaptive meshes or different combinations of physical processes modeled at different simulation stages) then the application requirements must be periodically re-estimated even on the same set of resources.

5.2.4 Resource pooling and selection

Resource pooling provides for the acquisition, maintenance and refinement of a set of Grid resources. The user-level environment controls the resource pool and maintains a desired amount of resources with certain parameters best fitting the application requirements.

The refinement of resource selection in the pool is based on the basic optimality principle in divisible load scheduling problems which states that to obtain optimal processing time all the participating processors must stop computing at the same instant in time [111]. While the above claim seems to have an intuitive validity, the optimal processing time can be achieved by distributing the load only among the "fast" processor-link pairs. A reduced network can then be obtained after eliminating the slow processor-link pairs and the load is distributed among the remaining processors using the optimality principle.

Based on the above it is reasonable to say that although the optimality principle remains valid for even an arbitrary network topology, the optimal time performance depends crucially on the selection of a proper subset of the available processors. Thus, using a larger set of nodes may yield an inferior performance compared to an optimal subset of nodes among which the load is distributed according to the optimality principle.

Division of resources to fast and slow can be done after introducing a ranking algorithm. Evidently, the meaning of "fast" and "slow" for a resource can depend on the type of application it is supposed to run. Thus we have to introduce a metric for appropriate resource ranking dependent on f_c of a particular application (as defined in Section 3.3).

To rank the resources we selected a metric similar to the one used for processor weighting in AWLB:

$$r_i \sim p_i(1 + f_c/\mu_i) \tag{5.1}$$

where r_i is the rank of processor i.

For the application to run the first M processors with highest rank are selected where M is defined as the number of processors that give a reasonable speedup. M is defined during a procedure of subsequent application iterations with increasing number of processors used, starting from 1 and growing up to the value not giving a significant application speedup growth any more:

$$\begin{array}{l} M = 1 \\ while(t(M+1)/t(M) < 1 - \varepsilon) \\ M = M + 1 \end{array} \tag{5.2}$$

where $t(m)$ - execution time of an iteration on m processors with highest rank, ε - threshold of minimal acceptable speedup growth. The rank r_M is called border rank.

To support dynamic behavior of resources the value M is evaluated each time a resource with rank $r > r_M$ joins or leaves the pool. In case of such resource joining it is inserted into the sorted resource list as: $\{r_0, .., r_{i-1}, r, r_i, .., r_{M-1}, r_M\}$, thus r_M is excluded from the first M processors with highest rank. To evaluate new speedup behavior on the updated resource pool the algorithm 5.2 is applied again (starting from M resources) to determine the current value for M. Similar procedure is performed when a resource i ranked $r_i > r_M$ leaves the resource pool. The sorted resource list looks like $\{r_0, .., r_{i-1}, r_{i+1}, .., r_M\}$, and the new value of M is updated to be equal to $(M - 1)$. The algorithm 5.2 is applied to find out if the speedup dependency has changed, and the value M and the border rank can be updated.

The method to select the threshold ε is typical for the tasks with stepwise approximation: the closer approximation to the optimal value is desired, the smaller the threshold should be. On the other hand, the price of achieving maximal speedup is the occupation of larger number of processors, and the gain of couple of percent in the speedup might not worth acquiring another worker node. In our experiments we set ε equal to 0.1 which was based on the empirical consideration that 10 percent of gained speedup is still worth acquiring another worker. Additional experiments on the influence of ε on the performance of the system can be carried out, but they are not covered in the scope of this thesis.

5.3 DIANE environment for user-level scheduling

To implement and validate the hybrid AWLB+ULS approach described in Section 5.2.3, we chose DIANE - DIstributed ANalysis Environment [86, 129], which is a realization of user-level scheduling environment developed at CERN. The framework provides the execution environment for parametric parallel applications i.e. the applications which are not communication-bound and for which the communication occurs in regular patterns. This covers a broad class of applications including parameter sleep, data-analysis if data locality is assumed, Monte-Carlo simulations

etc. The communication backbone of the environment is based on Master/Worker model however customization allows to achieve more complex task synchronization patterns. DIANE allows to plug-in user-defined scheduling algorithms and failure-recovery strategies. DIANE layer runs as a set of regular user jobs, and therefore it operates entirely inside the user space. User-level scheduling does not require any modification to the Grid middleware and infrastructure, nor the deployment of special services in the Grid sites, thus it provides immediate exploitation of Grid resources available to the user. Lightweight, transient services such as Job Master or Directory Service are run locally on user-controlled computer and may be enabled or disabled at any time.

DIANE has been used with a number of applications, from black-box executables (e.g. image processing [17], telecommunications [84], regression testing and data analysis [121]) to interfacing the applications at the source-code level (e.g. medical physics and bio-informatics [82]). DIANE allows to increase the application performance, minimize the feedback latency [43], and improve certain Quality of Service parameters of the Grid, such as reliability and predictability.

DIANE provides the software plug-in framework and uses Ganga [87] as a job abstraction and management layer. DIANE user-level scheduling exploits the concept of late-binding also known as place-holders or pilot agents similarly to Condor-G glide-ins [39]. The Grid jobs run generic agents which get the workload from a scheduler - a Job Master service. Deferring the mapping of workload until the runtime helps to short-circuit the overhead of hierarchical scheduling and to accurately react to the dynamically changing characteristics of the resources or of the application. The execution of jobs is controlled by the Job Master service running on local user computer.

Typically the Grid worker nodes have the constraints of outbound-only direct connectivity with the outside networks. Certain subsets of Grid sites may provide inbound connectivity on certain ports in order to support the cross-cluster MPI applications [135]. However such support is not ubiquitous and may not be relied upon for an average Grid user in an average VO. The worker node cross-talk is still possible in DIANE however the communication is routed via the Job Master service. In DIANE model the output of one job may trigger execution of another job if decided by the Job Master. This is the only allowed way of inter-worker communication. This model is more efficient for applications with low f_c parameter because the communication even between neighboring workers is routed via a distant point in the wide area network. On the other hand the AWLB methodology may be applied directly because the resource parameter μ is evaluated always against the same worker endpoint (Job Master). Thus the μ is invariant of any communicating worker node pair.

The experiments were carried out on the EGEE grid testbed [40], in Geant4 VO [133]. The model application is implemented as a python-based plug-in for DIANE environment. The master-worker model of execution is employed, where the master selects the amount of workload to be processed by a worker, sends the data to workers and receives the results.

5.4 Simulation results and discussion

5.4.1 Adaptive workload balancing and self-scheduling comparison

To validate the methodology of integrated AWLB+ULS approach, we experimented with a model application with a synthesized workload and tunable communication to computation ratio f_c. The experiments compare performance results achieved using the AWLB algorithm and dynamic resource pool with the results shown by the standard DIANE job dispatching technique - self-scheduling (also called a FIFO scheduling algorithm). In self-scheduling all the workload is divided into jobs of equal size, typically the number of jobs exceeds the number of available worker nodes. As soon as a worker becomes available, the next job from the list is assigned to it. In AWLB all the workload is divided into the number of available workers, and the size of the workload assigned to each job is calculated by the heuristic algorithm, using the resource and application characteristics.

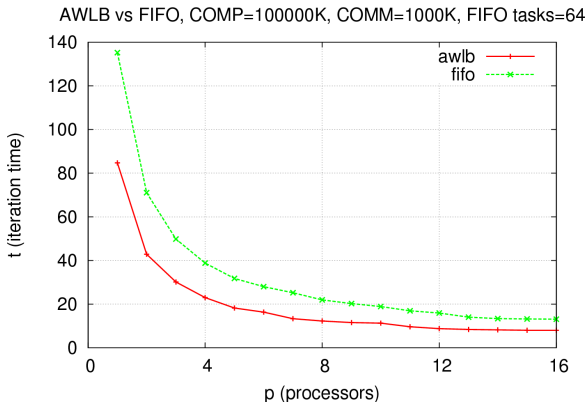

Figure 5.2: Comparison of AWLB and self-scheduling algorithms: Runtime dependency on the number of processors acquired. $f_c = 0.01$

Figure 5.2 presents a comparison of execution times of one iteration with the AWLB and self-scheduling (FIFO) algorithms. In this figure COMP is the total amount of computational operations (in Flops) and COMM is the total amount of communications for each simulation (in bytes transferred). In this experiment the number of processors used is increased by 1 processor for each iteration. In all cases, the AWLB significantly outperforms the self-scheduling almost twice. In some cases the gain can be up to several times.

Table 5.1 illustrates a typical example of AWLB parameters on a set of heterogeneous resources. Each worker is ranked according to the resource parameters and application characteristics (see Section 5.2.4). In the table, PROC and NET are the relative processor and network capacity of a worker (results of benchmarking). The

worker	PROC	NET	R ($\times 10^{-2}$)
1	57.9	4.3	1.1
2	42.4	13.1	1.9
3	52.3	28.8	3.7
4	54.8	28.9	3.8
5	55.5	28.7	3.7
6	96.0	16.4	2.8
7	42.4	28.9	3.6
8	42.4	28.5	3.6
9	41.2	28.4	3.5
10	53.0	8.9	1.5
11	53.2	28.3	3.7
12	33.6	3.2	7.0
13	41.8	28.9	3.6
14	42.1	28.1	3.5
15	49.9	42.6	5.2
16	47.0	27.8	3.5
17	53.8	27.9	3.6
18	22.2	7.4	1.0
19	55.2	28.4	3.7
20	57.8	4.3	1.1
21	39.5	28.0	3.5
22	64.9	11.9	2.0
23	29.9	9.8	1.4
24	41.5	28.9	3.6
25	41.1	28.8	3.6
26	41.1	28.6	3.6
27	41.0	28.5	3.6
28	41.1	19.0	2.5
29	45.9	42.0	5.1
30	41.7	28.6	3.6
31	40.9	29.1	3.6
32	41.4	28.7	3.6

Table 5.1: Sample distribution of processor ranks (R) by AWLB on a set of heterogeneous workers.

application values COMM and COMP are the same as in Fig. 5.2.

Figure 5.3 shows how the application communication to computation ratio f_c influences the execution time for different workload distribution algorithms. During this experiment the computational load (COMP) was kept constant on a fixed set of 16 processors, while the amount of data transferred between the master and the workers (COMM) is varied. For all types of simulations, the AWLB algorithm is significantly faster than the self-scheduling.

Figure 5.4 presents the statistics of an actual run using a dynamically populated resource pool. The core feature of the ULS environment is the ability to change resources during the execution runtime, such that every iteration can run on a different set of resources. Figure 5.4a shows how the resources are gradually added to the resource pool, depending on their availability. We can also notice that some workers are removed from the pool: the number of workers is not growing steadily, but

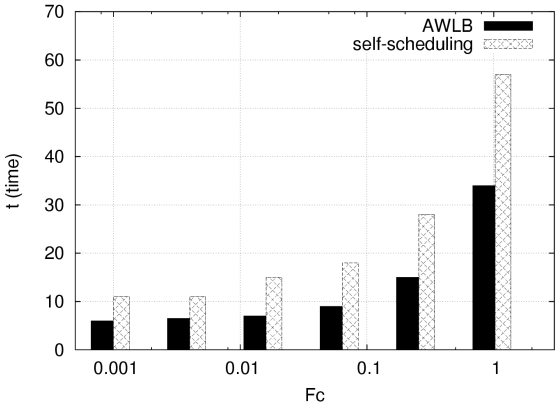

Figure 5.3: Sample dependency of the runtime on communication/computation ratio f_c for 16 workers and 64 FIFO jobs (COMP=30M, variable COMM).

experiences dips every few iterations.

In Figure 5.4b, the basic resource matching and selection mechanism is demonstrated (Step A3). All the workers acquired at each iteration (shown in Figure 5.4a) are used for the execution, and the execution time depends on the number of available workers at the moment. Notice that at iterations number 9,13,16 when some resources left the pool, the execution time increases.

5.4.2 Adaptive resource selection

To illustrate the adaptive resource selection algorithm presented in the Section 5.2.4 we analyze the execution of the model application on the dynamically acquired Grid resources. The distribution of the resources obtained for the experiments is presented in Figure 5.5a. Each point on the plot reflects a single available worker with corresponding processor performance and network connectivity to the master.

To check the behavior of different types of applications we modeled different amount of computations and communications (i.e. different f_c). The resource ranking used for resource selection (Section 5.2.4) is based on application properties, thus the rank of a single resource depends on the application it is used to execute. Figure 5.5b illustrates resource ranking for different application types (i.e. different values of f_c) for the same processor/network parameter distribution shown in Figure 5.5a. Namely, in Figure 5.5a the absciss axis marks the ordered number of a worker in the resource pool, the white triangles represent the processor capacity of the worker (measured on the left ordinate axis), the black triangles represent the network connectivity to the worker (measured on the right ordinate axis). Figure 5.5b and illustrates different values of weights given to the same worker in case of different assumptions about the application parameter f_c. The placement of the figures on the same vertical line and the same scale of the absciss axis allow to easily follow the dependency of weight

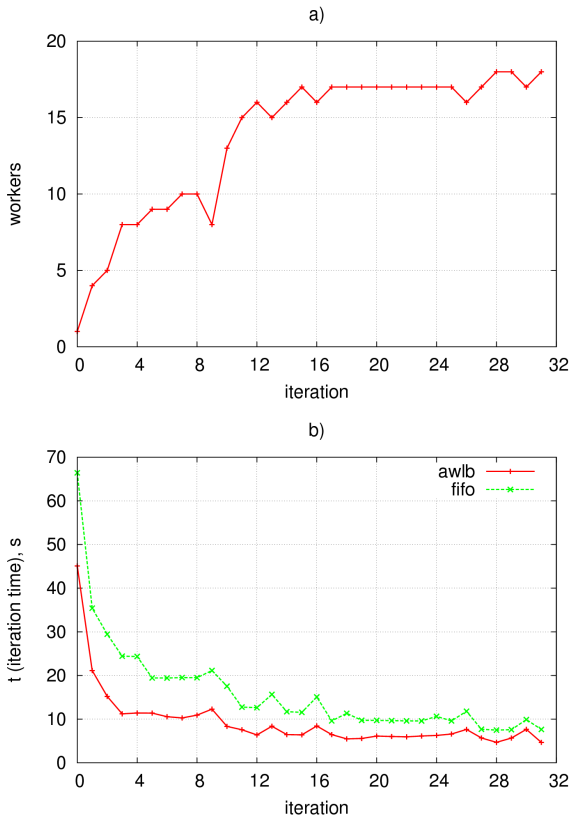

Figure 5.4: a) Dynamic resource pool population; b) Sample execution times using a dynamic pool.

volatility on processor/network pairs for each worker.

The general idea of resource selection is to provide the best resource subset from the set of available resources to ensure the fastest possible execution of the application. To estimate the performance gain from the resource selection procedure, we analyzed execution times on the best, average and worst sets of available resources. The number of workers was fixed and the resources were selected from the top, middle and bottom of the ranked resource list. Thus the performance was estimated for the same number of workers with the highest, average and lowest ranks. Figure 5.6 illustrates the efficiency of the resource selection algorithm by presenting the resulted performance difference.

The number of processors to use for efficient execution of a parallel application with divisible workload depends on both application and resource characteristics, and it is difficult to predict the speedup saturation point (the number of workers with highest

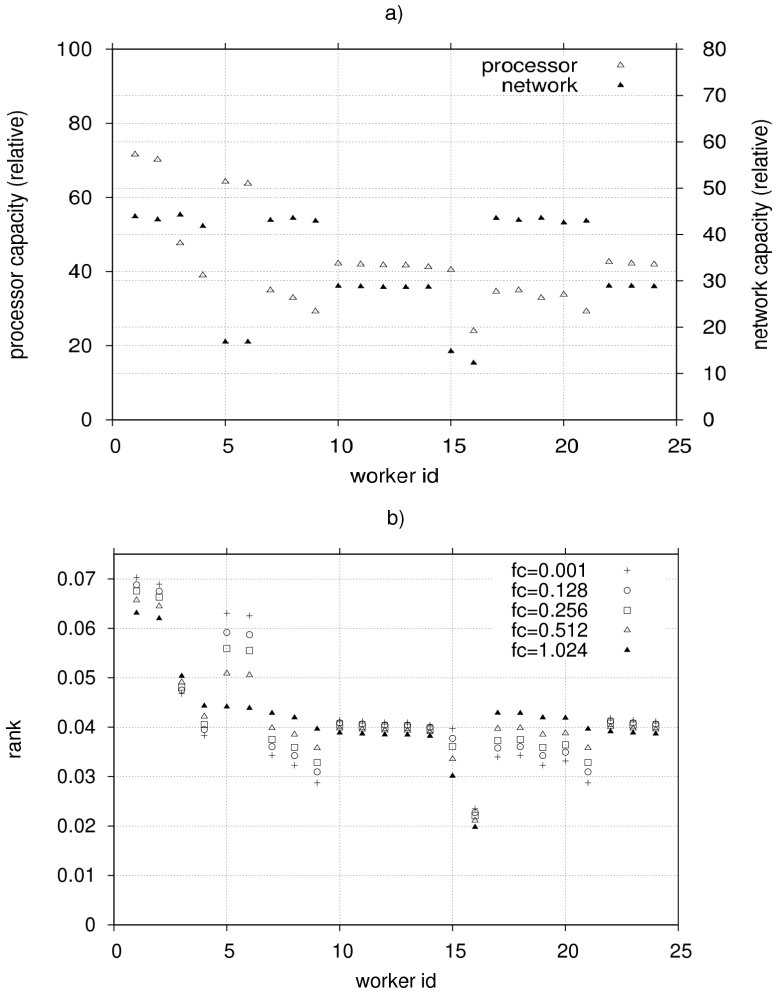

Figure 5.5: a) Sample resource distribution: processor and network capacity for the workers in the resource pool; b) Resource ranks for the workers (dependency on f_c).

ranks used) in advance. The adaptive resource selection (Section 5.2.4) performs the analysis of speedup growth with addition of every new worker to the set of workers used, the threshold ε can be explicitly set by the user. The figure 5.7 presents sample executions of applications with different f_c on the same resource pool and ε equal to 0.1. As it can be predicted, the speedup of the application with higher communication demands saturates on smaller amount of workers, which is tracked by the resource selection algorithm. Thus the resources not acquired from the resource pool to provide the marginal speedup growth can be more useful for another application that accesses

the same resource pool.

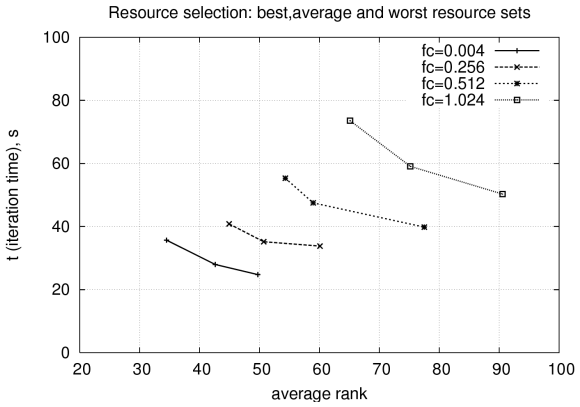

Figure 5.6: Resource selection: comparison of best, average and worst ranked resources performance. On X-axis the average rank of each resource set used for the execution is shown.

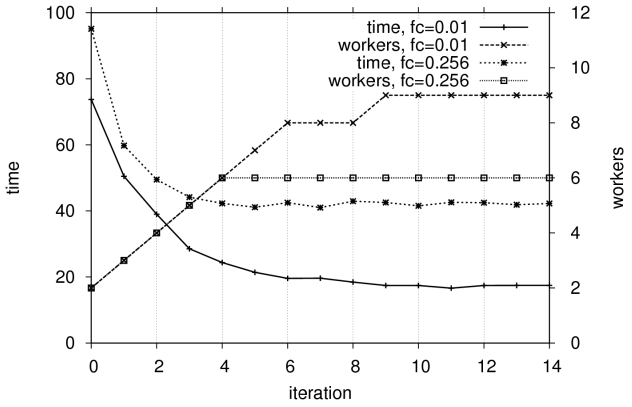

Figure 5.7: Adaptive selection of amount of workers to use, $f_c = \{0.01,0.256\}$

The experimental results presented above mostly address the cases when application parameter f_c is known. But this is not the case for real applications frequently. The AWLB algorithm is designed to be able to figure out the proper value of f_c automatically during several initial iterations of execution. The detailed description of this functionality is given in [68] and chapter 3.

5.5 Conclusions

Evolution of the Grid paradigm to support High Performance Computing is indispensable for a large number of applications which require on-demand access to the Grid resources. In this chapter we proposed an approach to enhance the quality of handling multi-job applications in Grid environment by integrating the Adaptive Workload Balancing Algorithm (AWLB) developed for parallel applications on heterogeneous resources and User-Level Scheduling (ULS) environment. The latter gaps the missing link between applications and Grid resource managers: this user-level middleware is a customizable, application-centric scheduler and application hosting environment. Dynamic benchmarking of resources and estimation of the application characteristics is used to optimize the usage of a dynamic user-level pool of Grid resources maintained by the ULS. We devise a generic recipe on how to solve the workload balancing problem on the Grid for multi-job applications. To prove the concept we performed tests with a synthetic application with configurable requirements using EGEE Grid resources and the AWLB algorithm incorporated into the DIANE user-level scheduler. We present experimental results and discussion on the way to manage the workload of divisible load parallel applications on the Grid, compare different workload distribution methods, illustrate the usage of dynamic resource pool and application performance dependencies with adaptive resource selection.

This chapter builds an intermediate step between parallel applications in Grid environment discussed in chapters 3 and 4 and multi-component distributed applications discussed further. We investigated the management of multi-job applications facilitated by User-Level Scheduling environment that in the aggregate formed an integrated solution for combined resource management on application and system levels. In the next chapter we focus the attention on the multi-component applications that do not share the workload (like it was discussed in all the previous chapters) but process it in a pipeline manner – scientific workflows.