Grid Technologies for Task Parallelization of Short Jobs: Usability Study

---

TDT4590 - Complex Computer Systems,
Specialization Project

Atle Rudshaug

Supervisor: Anne C. Elster

December 19, 2007

Department of Computer and Information Science

Faculty of Information Technology, Mathematics and Electrical Engineering

NTNU
Norwegian University of
Science and Technology

Abstract

In this project, the Condor and Sun Grid environments are evaluated as tools to schedule computations with lots of small tasks. Our strategy is to collect multiple tasks into meta-tasks and compare their performance to single single task jobs. Other methods discussed, include altering source code for computing multiple tasks internally, tuning the Grid schedulers and implementing a Master-Worker paradigm. Part of Dagoc, a commercial application from the Oil/Gas industry, is used as the test case. Basic scenarios contain between 1-1000 tasks. Each task takes between 2 and 20 seconds and are currently calculated serially on one processor. The tasks use different sets of files for input data, which resides on a NFS server. Splitting such small tasks into individual jobs may, however, not be suitable for Grid environments. However, positive results are observed through different benchmarks, using our proposed meta-task execution. Two well known Grid middlewares, Condor and Sun Grid Engine, are used in the benchmarks and their ease of install and performance is compared in the process. The installation procedure for Condor is shown to be much simpler, however, Sun Grid Engine generally performed better in our tests. Appendices containing practical discussions on installation and Grid usage procedures as well as practical examples of job scripts and code for handling result files, are included as guides to future users.

Contents

1 Introduction

In recent years, multicore architectures and networks of workstations has become popular parallell computing platforms. To utilize these new environments fully, software has to be (re)designed. A serial code cannot utilize more than one core on a multicore processor. However, multiple independent serial applications will be able to utilize multiple cores, if executed at the same time. Serial code can also be rewritten to make use of multiple cores, through OpenMP or posix threads. Hopefully, these will be better tools to help the domain developers of the future.

1.1 Project Goal

The main focus of this project is to look at two main Grid middleware platforms and evaluate how easy and well these can be used to support scheduling applications with many individual small tasks. Using Grid technology, multiple cores can be utilized, even multiple computers with multiple cores in parallel. Methods how to minimize scheduling overhead using built in Grid middleware functions are also considered. Note that these methods do not include altering the source code in the application to a great extent.

A typical office environment is considered. Here, the collection of workstations will in most cases be quite heterogeneous. Some machines are upgraded while others are kept as is. Some nodes might use Windows or Mac OS X while others use different flavors of Linux. Binary compatibility between these platforms will induce problems. Sometimes between different flavors of Linux as well. An executable compiled for one flavor of Linux might return an error on another, because of missing or outdated libraries or other causes. A Grid must be able to handle this. If jobs are distributed carelessly the success of each job execution could end up being quite random. Without any constraints, the Grid engine will send jobs to arbitrary free nodes. The user will not know if their job will be executed successfully or if it will return an error. A common interface to these resources is provided by different Grid middlewares.

1.2 Our application

In this project, we consider a commercial application, Dagoc, which is a tool from the Oil/Gas industry. It is developed by a small, Norwegian software company called Yggdrasil AS. The part of the application we are looking at, is a typical Parameter Sweep Application (PSA) [10]. We consider the computation of different sized collections of small, equal-sized and independent tasks, commonly known as meta-tasks [16]. The meta-tasks are currently designed to be executed serially on a single workstation. Each task in a meta-task, computes a fixed amount of source data from a list of different input parameters. The final result of a meta-task is to be saved at the end. At first look, our application should be perfect for use in a Grid environment.

However, the individual tasks are quite small, between 2 and 20 seconds. An average task needs about 0.5MB of input data from multiple small files. The executable is about 22MB and needs about 1.5MB of custom libraries. If the program is to be executed in database mode, it will need access to an 1-2MB sqlite3 database file as well. For such short tasks, the scheduling and file distribution may become a dominant factor.

1.3 Outline

The remainder of this paper is organized as follows. Section 2 presents a general overview of Grid technology and a description of the two Grid middlewares used in this project. Different thoughts on how to solve the problem is presented in Section 3. Section 4 describes different issues related to the benchmarks and the respective Grid middlewares. Benchmarks and results are shown in Section 5. Section 6 concludes the project and discusses future work.

In Appendix A, issues concerning installation procedures for the respective Grid middlewares, are discussed. Appendix B, describes issues concerning job submission for both Condor and SGE. The example scripts in Appendix C, can be used as starting points for creating other, more complex scripts.

2 Grids

Grids are often referred to as High Throughput Computing (HTC). A Grid is a collection of different, privately owned, computer resources to form a type of heterogeneous, virtual supercomputer for providing computing power for large-scale jobs [1, 2] . Their job is to distribute a high number of jobs efficiently, through a common interface, and provide long lasting computation time.

A simple Grid can be formed by local workstations, for example inside an office environment. Every day there are hours of idle computer time during, for example, the lunch hour, staff meetings, after-office hours and at night. In these periods, the workstations can be used for various computations. If the local resources are not enough, the number of resources can be dramatically increased, by connecting the local Grid to remote Grids in other locations. The ultimate Grid would be the one with access to all the computing resources in the world. However, people are usually very reluctant to let other, unknown people use their hardware, at least while they are using it themselves. So what can be done to get permission to use these resources?

Introducing a computational economy [3], is suggested as a good motivator for people to share their resources over the Internet, making it a computational power Grid. This also opens for smaller companies to buy only the resources they need to get their current jobs done, and not make huge investments in own infrastructure and computing power.

Other features are needed in a Grid as well, to attract users and resource providers. These include a transparent interface for resource allocation and administration, fault tolerance and different security and authorization tools. A secure environment is important, so the providers know their resources will not be exploited [2].

Grids are, however, not to be confused with clusters. A cluster is typically a collection of identical nodes with the same processor and OS, typically containing a static number of nodes, all placed in the same physical location. A Grid is a heterogeneous system [18], with different types of nodes (e.g. computational or storage nodes), processors and OS's. Grids are dynamic in number and resources, while clusters are generally more static over time. Another important feature is that Grids can contain different HPC environments, such as clusters and supercomputers, in addition to other resources [13]. Thus, a Grid can represent a heterogeneous environment with the possibility to utilize the power of supercomputers for less embarrassingly parallel tasks as well.

Grids can be a cheap alternative to dedicated supercomputers, since a Grid can utilize idle time on already available workstations. These workstations can be very cheap and do not need special rooms or cooling facilities as large supercomputers do, unless a large number of nodes are clustered together. However, as Grids are heterogeneous systems, they are best suited for embarrassingly parallel jobs where the individual jobs are independent of each other during execution. There are several cases where a collection of heterogeneous workstations is not a suitable replacement, e.g., for fine grained parallel tasks with high dependency between processes. Here, each job is best run on its own cluster or supercomputer for optimal performance. Submitting multiple fine grained jobs simultaneously to a Grid with access to multiple clusters or supercomputers, can give a combination of coarse- and fine grained job execution. For example, a large job consisting of multiple independent fine-grained jobs, can be automatically distributed by a Grid and be run coarse grained on different clusters simultaneously [4].

Different tools have been developed to transparently handle the dynamic nature of Grid systems, as well as standards for developing Grid integrated applications. Efficient scheduling and execution of PSA's on a Grid is a big challenge for Grid developers. These applications often consist of a large number of jobs where the final result is dependent on all the individual results. These jobs must therefore be scheduled and distributed effectively, so not to delay the total execution time. Methods including re-use of common files between executions and adaptive execution to migrate jobs to provide better resources, are some suggested solutions [9, 10].

George Tsouloupas and Marios D. Dikaiakos [19] suggest a method for ranking resources in a Grid according to a ranking function. They have developed a tool called *SiteRank*, a module built on top of GridBench. With it, a user can rank all resources in a Grid with respect to a specific application. This tool can be used to better utilize the resources in a Grid for any kind of job, including the short ones presented in this paper.

An example of a large Grid can be seen at CERN. They are currently developing Grid tools for their Large Hadron Collider (LHC). They need an incredible amount of storage and computation power, and are connecting sites all over the world to their Grid to satisfy their need [7]. Without a Grid, it would be impossible to maintain the data and computation throughput necessary for the LHC project.

2.1 Grid features

2.1.1 Condor 6.8.6

Condor is a free Grid manager from the Condor team. It was born at the University of Wisconsin in the 1980's, as a combination of a doctoral thesis on cooperative processing, the Crystal Multicomputer and Remote Unix. It creates a High-Throughput Computing (HTC) environment by opportunistically utilizing workstations connected through a regular network, remote as well as local. The main features that makes this environment possible are ClassAds, Checkpointing & migrating and Remote System Calls [18, 15, 12].

The ClassAds system is a powerful mechanism for matching jobs to execution nodes. Users advertise their resource needs for a job and Condor matches them with the resource ads for the available workstations. This way, the necessary resources are acquired to best match each job.

Checkpointing is a system for transparently moving already running jobs from one workstation to another, if necessary. This will, for example, happen when a user returns from lunch and starts using his or her workstation. Condor will only schedule a job to a node which has been idle for a predefined amount of time, thus not bothering the owners of the respective resources. Each running job is regularly and transparently checkpointed during execution to make this possible. When a job migrates to another node, the new node can resume execution from the last checkpointed state.

Remote System Calls technology, as with checkpointing, requires relinking of the job executable with specific Condor libraries. The Remote System Calls feature preserves the submitting node's local execution environment, by redirecting a jobs I/O mechanisms back to the submitting node. Thus, distributing the executables and its input files is not necessary prior to job execution, as this is handled automatically. It also gives a user access to the executing node without having a login account on it. There are however a number of limitations[1] to jobs which are to support checkpointing, including

[1] `http://www.cs.wisc.edu/condor/manual/v6.8.5/1_4Current_Limitations.html`

running them on Windows nodes. If for some reason the executable cannot be relinked to run in the standard Condor universe, e.g., no access to the source code, the executable can be run unaltered in the "vanilla" universe instead. However, when using this universe, file transfer has to be specified by the user in the submit script, as described in Appendix A.2.2.

To run jobs with dependencies, Condor includes a feature called DAG-Man. This is a Directed Acyclic Graph Manager, where rules for job dependencies and pre- and post processing scripts can be set up in a special file. The pre- and post scripts are run locally on the submit host. When this type of job is submitted, the DAGMan takes care of the order of execution according to the rules specified by the user. However, each job defined in a DAG still needs its own regular Condor submit script.

Distributed Resource Management Application API (DRMAA) 1.0 Java and C bindings are also supported. This API can be used to integrate Grid technology into applications, instead of manually submitting jobs through a console.

Many other projects are available for use with Condor, including a file handling system called Stork, a system monitoring tool called Hawkeye, a master-worker paradigm called MW, and more.

2.1.2 Sun Grid Engine 6 (SGE)

Sun Grid engine is a free Grid manager from SUN [5]. It is now an open source project, with support contracts available from Sun. One of the newest features in v6.1 is Resource Quotas, a feature for controlling resources in the Grid. Access rules to different parts of a Grid can be set up for users, groups, projects, etc. for fine grained control of the available resources. A Condor ClassAds alternative in SGE is Boolean operations. This is a tool for creating rules for specifying resource needs with AND, OR and NOT operations.

Job dependencies can be managed using the Grid Engine Array Job Interdependency (ARI) [2] feature. This, in combination with prolog- and epilog scripts, gives similar functionality for SGE, as DAGMan does for Condor.

Execution of parallel jobs (MPI or PVM) is supported through a dedicated interface, as with Condor. SGE also supports checkpointing and migration among other tools and functions.

A GUI interface for easy configuration and administration of queues, jobs and nodes is available for Linux. However, all features in this GUI are also available from the command line.

Distributed Resource Management Application API (DRMAA) 1.0 Java and C bindings are supported on the Linux platform, but not on Windows.

[2] http://gridengine.sunsource.net/news/GE61ARIsnapshot-announce.html

2.2 Benchmarking grids

There are not many tools available for benchmarking Grid environments. Their heterogeneous nature makes this challenging compared to traditional parallel, high performance systems. Liang Peng *et al.* [14], have done some work on benchmarking the performance between SGE and Globus in terms of CPU utilization and turnaround time. They noticed that the overhead introduced by the Grid middlewares was negligible for large problem sizes. In their case, the overhead actually changed very little even when the problem size grew significantly. For short jobs, however, they found that the overhead can be quite significant, sometimes half the total time for a job. They also found that the Globus middleware generally had more overhead than SGE. The significant overhead for small jobs is what we are considering in this project.

3 Implementation Ideas

In this section, different ideas on how the respective Grid middlewares can be used to support our application, are discussed. Different issues concerning the location of input and output files are also considered.

Our PSA may consist of thousands of permutations, where each permutation needs to be computed to find the final result. Luckily, Grid systems support methods for submitting multiple jobs automatically, as described below.

In a Grid environment, all nodes must have access, locally or remotely, to all resources needed by the tasks they are given. If not, the tasks will obviously return an error. In most cases, these files are located in remote places, e.g., on a NFS server for easy administration. For each task, these files have to be transferred to the respective execution nodes. Since the tasks in our application are so short, this extra file transfer overhead is factor to be considered. Since many files are common between the different tasks, methods for collecting multiple tasks in one job is the main focus in this project. A collection of multiple independent tasks is known as a meta-task [16] and will be used throughout this paper.

The following methods are considered to minimize overhead:

- Altering the source code to execute multiple calculations from input in the argument list
- Tuning the schedulers for faster submitting of jobs
- Bash script with multiple executions
- DAGMan and Master-Worker features in Condor

3.1 A golden rule

A golden rule is to exploit application domain optimizations before platform domain ones. Altering the source code of one application, is not a general solution across other applications, like the solutions discussed below. However, in many cases it should be possible to make an application execute multiple tasks internally, by altering the input arguments. When altering the code to include multiple computations, result comparison between tasks can be implemented as well, and only the best result would have to be returned to the submit node. However, how much time is really saved by making the application do multiple executions internally, compared to executing multiple single-executions in a bash script? Time will be saved by not having to start and stop the executable for every task. The question is if the time saved for each execution is noticeable compared to simply specifying multiple computations in a bash script.

This may come down to which is easier in the long run, depending on the application. Defining multiple runs in a script, each with different input, or altering the source code to open for calculation of multiple tasks internally, in one single executable.

3.2 Scheduler tuning

The default scheduler settings might not be optimal for every compute farm environment. Different actions can be taken to fine tune the schedulers for optimal performance in specific environments. The SGE scheduler supports different tools[3] for debugging and validation of scheduled jobs. These can be turned on or off, depending on the specific needs. When the Grid is in production state, these tools may not be necessary all the time and can be turned off by the administrator.

For example, configuring the SGE scheduler for immediate scheduling, will increase the throughput of the compute farm. The only limitation is the power of the machine hosting the master and scheduler. If this machine is overwhelmed by work, the scheduler can be configured to run jobs only in a fixed schedule interval, which also is the default setting.

In Condor, different parameters[4] can be tuned in the configuration files for faster scheduling. One of Condor's default behaviors, is not to schedule jobs to non-idle nodes. It also preempts and/or suspends jobs, if the current node becomes unavailable due to user interaction. These features can be disabled, if seen fit for the compute farm.

[3] http://docs.sun.com/app/docs/doc/820-0698/enfky?a=view
[4] http://www.cs.wisc.edu/condor/CondorWeek2007/large_condor_pools.html

3.3 Submitting multiple jobs in one submit file

With Condor, one can submit multiple jobs in one submit file simply by stating the number of jobs with the *"Queue"* command, e.g., *"Queue 50"* for 50 jobs. Each job can be identified with the $(CLUSTER) macro and sub-jobs with the $(PROCESS) macro. In this case, each sub-job gets a unique identifier from 0-49. Different input parameters can be defined in the submit script, by using different macros. Using this multi submit method is similar to submitting 50 jobs manually, thus it does not remove any scheduling overhead. It only saves the user time by instantly queuing X number of jobs automatically.

For SGE, the alternative is called Array Jobs. Array Jobs can be specified either in the submit script by adding *"#$ -t first-last:step"*, or as an argument to the SGE submit-to-queue binary *qsub*. Instead of the $(PROCESS) macro in Condor, SGE defines a set of environment variables for the array job, to identify the task and task range. To submit an array job from the command line, type the following when submitting the job: **qsub -t 1-10:2 script.submit**. This will queue 5 jobs with step 2. The tasks will get SGE_TASK_ID 1, 3, 5, 7 and 9. SGE_TASK_ID, SGE_TASK_FIRST and SGE_TASK_LAST are environment variables set by SGE for this particular array job. The scheduling process is similar to Condor's Queue X command. Each job is scheduled individually, but time is saved by automatic queuing of multiple jobs at the same time. The environment variables can be used to automatically select the correct input files for the respective tasks.

3.4 Bash script with multiple executions

The executable in Condor can either be a binary executable or a bash script. This is defined in the submit script with the argument *"executable = filename"*. For meta-tasks, multiple executions can be specified in a bash script and the binary can be transferred as an input file. Thus, the executable is only transferred once for the whole meta-task, and is reused by all the tasks specified in the bash script. Each task's result, will be appended to the specified output file if written to stdout. If the application creates any new files, these will also be transferred back to the submitter automatically by Condor. Thus, the execution of multiple tasks does not overwrite any intermediate results.

SGE's submit script is very similar to a regular bash script. SGE specific flags and options can be defined directly in this script with *"#$ -flag option"* notation. Multiple executions can therefore be defined directly with bash arithmetic's and submitted as is. However, SGE does not automatically transfer any input files. Its submit script only invokes the remote host's environment as if it was invoked locally. However, SGE supports prolog/epilog scripts that can perform any necessary processing, including file transfer, before or after job execution. These scripts are run on the execution host and not on the submit host, as for Condor's DAGMan. For SGE, the executable and input files can either be located on NFS for easy administration, transferred by a prolog script or located locally in the same path on every node for minimum network traffic.

3.5 Condor specific tools

3.5.1 DAGMan

DAGMan makes it easy to define job dependencies. The jobs in the DAG are regular Condor submit scripts and each job is scheduled individually. Therefore, DAGMan does not help to minimize scheduling overhead of jobs in any way. It can, however, help with post execution result gathering, as described in Section 3.6.2.

3.5.2 MW: Master-Worker

The master-worker paradigm [17, 6] can be very quick for collections of short jobs. The Condor implementation consists of a set of abstract classes, namely Task, Driver and Worker. The Driver sits below your application and manages a pool of Workers and set of user defined Tasks. The Workers pick up Tasks, does the user defined work on them, and returns result to the Driver. The implementation is specific to each application and will therefore involve altering the existing code, if not implemented during initial development of an application.

Implementing MW in our application falls outside the scope of this project, as we are mainly looking at ways to use Grid tools to optimize execution of our application, without modifying the source code.

3.6 Handling the input/output files for Dagoc

The amount of files that are needed for each task, as mentioned in Section 1.2, might become a factor for the total execution time of our small jobs. Obviously, the job executions would benefit from reusing as much of these files as possible on each node. One solution would be to have all the data and the executables locally on all nodes, but this would be difficult to administer. For easier administration all files could be put on NFS, but this might generate a lot of network traffic, since every calculation has to access it for its input and executable.

By collecting multiple tasks into meta-tasks, all common files would only have to be transferred to the execution node once, and could save a lot of traffic. Hence, each task in the job would only have to transfer a small amount of files, unique to that job, from the NFS server. All other files would be already available locally on the node for the duration of that particular job.

3.6.1 Condor specific file handling

In Condor, input files can be transferred from the submit host, by defining the necessary input files in the submit script. Using the option *"transfer_input_files = file1,file2,...."*, these files will be copied next to the executable in the execution node's spool directory. However, transfer of whole directory structures is not support, only lists of specific files. See Appendix C for a Condor submit script example with file transfer.

Since the "Vanilla" universe is used (see Appendix B.2.1), the options *"should_transfer_files = YES"* and *"when_to_transfer_output = ON_EXIT"*, are needed to specify that the executable and results are to be transferred between submit host and execute host.

3.6.2 Result gathering with DAGMan

When distributing our application onto a Grid, post processing to find the best result is necessary after all the calculations are finished . This can be set up using job dependencies in DAGMan. Each job has its own regular Condor submit file, like the one shown in B.2.2. The

A solution was implemented, where the post processing job was configured to use the "local universe". This forces the job to run on the submit host, where all the result files are located. This job contains a script that scans all the results files, extracts the results and puts them all in one single file. The corresponding input parameters are saved as well. Regular Linux tools such as 'grep' were used for this. Then, a short C++ program was developed that extracts the best result from the new single result file (See Appendix C). The input parameters for the best result will then be used to execute the task with the best result locally, on the submit node. This will save all the execution data in the database. All this post processing is done automatically, in sequence, by the last bash script.

3.6.3 Result gathering with SGE

Using the ARI functionality mentioned in Section 2.1.2, a post job result gathering script, similar to the one described above, can be used. However, if NFS home directories are not used, each execution host will have to transfer their results to the submit host, before the final result can be extracted. File staging[5] with epilog scripts can be used for this. File staging has to be enabled by the administrator. The epilog script can use different variables, set by SGE, to identify which files are to be sent where. An epilog script was

[5] http://gridengine.sunsource.net/howto/filestaging/filestaging6.html

written to transfer the output from SGE's array tasks back to the submitter automatically (see Appendix C).

3.6.4 Problems with NFS file access

Accessing SQLite databases on NFS may induce problems. In some perfect, up to date NFS setups it might work. Others, including ours, have issues with file locking and databases. Since our application uses an SQLite database for data access, this problem was encountered while having this database file remotely. A workaround in Condor, is to copy the database file as an input file by adding it in the submit script. For SGE, a prolog script can be used to transfer the correct database before the job is executed. The other alternative is to have a local copy on each node. This should not be a problem for the calculations in our case, as the intermediate tasks are not writing to the database, but only reading. However, administration might become cumbersome and jobs may at times give false results if some nodes are not updated correctly with new database files.

4 Model

In this project, the focus is on small office environments with limited resources. A small 3-node Grid is used in the benchmarks. The nodes are basic workstations connected through an Ethernet network. The Linux distribution Fedora 7 is used as operating system on each node. Furthermore, the workstations are quite different, as shown in Table 1.

Methods for distributing a collection of short independent tasks on multiple nodes, are considered. If using multicore nodes in a Grid, one would also be able to exploit all the cores available, without altering the source code of the application. This is possible due to the Grid engine's ability to schedule a job for each CPU on a node. Since HyperThreading technology is interpreted as an extra CPU by the operating system, the [P4Hyper] machine will be scheduled two jobs simultaneously, when used in a Grid. This might give a slight speedup, although not twofold since the extra CPU detected by the OS is only virtual.

All the nodes in the Grid are assumed to be idle during benchmarking. The default Condor setup excludes nodes, which have not been idle for a period of time, as candidates for jobs. Since one of the execution nodes is also used as the submit node, Condor is configured in testing-mode to remove the waiting time for this node. This will make Condor's environment more similar to SGE's. The nodes are not used for other tasks while benchmarking, so not to bias the results. All tasks used in the benchmarks will be given identical input. The calculated results can then be used to verify that the same calculation is executed in every task.

Job execution is handled differently by the two middlewares, as described in Appendix B.1.1. To support SGE's file locality concept[6], all the necessary input files, including the binary executable, are assumed available on NFS. Hence, each node can find all specified files using the same paths, provided by SGE's bash script. However, NFS file caching and buffering might bias the result in SGE's favor. Some common files, including the executable, might already be available in the local file buffer on the execution nodes for the following task, reducing file transfer. Since we are not using NFS mounted home directories, the results from SGE will be located on the local home directories for the submitting user on each execution node. This may become a significant factor in the benchmark results, since Condor automatically transfers all results back to the submitting node. Hence, two benchmarks will be run for SGE; one with and one without result file transfer back to the submit node. An epilog script is configured, in SGE's queue, to be used by each task. After a job completes, the epilog script is executed and the results are transferred back to the submitter using SCP. Password-less SSH keys were distributed among the nodes prior to job execution. Thus, secure authentication is handled automatically, without user interaction.

The SQLite database file needed by the application will, for SGE, be locally available on each node during execution. This is due to the NFS file locking problems described in Section 3.6.4. For Condor, its regular file handling will be used, and the database file is transferred with the job along with the executable binary or bash script. However, the textual input files for each task will be located in the same NFS location for both SGE and Condor. Thus, Condor will transfer the executable and database file from the submit node to the execute nodes, while SGE will transfer the executable from the NFS server. The NFS server is in our case the same as the submit node, namely the [P4Hyper].

After all results are copied back to the submit host, the final result can be collected using the method described in Section 3.6.2. For Condor, DAGMan can be used. However, DAGMan jobs are not scheduled instantly, but only after about 5 minutes by default. Thus, the benchmarks for Condor were run by simply submitting the computation job script directly, since this is scheduled instantly. The time taken to find the final result is therefore not included in the numbers for either Condor or SGE. However, the final script is run manually and the extra time used is given in the results.

4.1 Test parameter

Walltime is used as the parameter to compare the time taken to execute X very short tasks serially versus distributing them on a Grid. We are looking for speedup in the range of seconds and minutes, not clock cycles, hence the

[6] http://gridengine.sunsource.net/howto/nfsreduce.html

Table 1: Workstation specifications

	[Athlon64]	[P4Hyper]	[Athlon32]
Processor	AMD Athlon64	Intel Pentium 4	AMD Athlon
Extras	64-bit support	HyperThreading	N/A
Speed	3500+	3.0Ghz	2500+
RAM	2GB	1GB	1GB
Grid job	Submit/Execute	Master/Submit/Execute	Submit/Execute
OS	Linux Fedora 7		

choice of timing parameter granularity. The following tests were run:

- Time used serially on each of the nodes
- 3 runs of single task jobs on both Condor and SGE
- 3 runs of meta-tasks on both Condor and SGE
- 3 runs of single task jobs and meta-tasks on SGE, without result file transfer
- Altering the job-task ratio for 1000 jobs

The benchmarks are run 3 times to see if the [P4Hyper] machine will give any significant difference in the total execution time, as well as to verify the results. HyperThreading does not nearly give double the computation power, hence, two jobs running simultaneously on the [P4Hyper] machine might use longer time altogether than if executed on two different nodes. Another benchmark where the the job-task ratio for 1000 tasks is altered, is also performed to see if there is room for fine-tuning the amount of tasks sent to different nodes.

The number of jobs were not chosen through an empirical study, but arbitrarily only to compare the different benchmarks. The range was chosen between 10 to 1000 tasks, to cover our application's usual task range.

The actual timings are extracted from log files, capturing the submit time and the end time for the last task in the job. In Condor, the user specifies the log file name, in which the submit, start and stop times for each task in a job are recorded. A bash script is used to automatically extract and calculate the time used for the total job (See Appendix C).

SGE has a tool, *"qacct"* , which extracts data about jobs, including wall-time. Data from each job is piped to a file and another bash script is used to extract the timing results for the jobs (See Appendix C).

5 Benchmarks and Results

This Section shows the results from computing X small tasks serially, compared to distributed on a small three node Grid. The ideas from Section 3.3 and 3.4 were used. A discussion follows of the results from using the these methods in both Condor and SGE. The serial results are shown first for comparison.

As mentioned in Section 4, finding the final result was to be done manually for practical reasons. The time used to collect, compare and extract the best result from 1000 tasks, was found to take about 3 seconds. This is negligible, when the corresponding calculation time is in the range of over a thousand seconds. For 100 tasks, it took less than one second. Thus, this last result comparison is ignored in the results.

5.1 Serial execution

Table 2 shows that the serial execution time varies by about 27% between the fastest and the slowest node. The average time taken between the three was used later in the comparisons. One single task is shown to take between 3 and 4 seconds. The same task was used in all benchmarks and this was verified by checking the result of the tasks.

Table 2: Results for serial execution

Machine/nTasks	10	50	100	200	500	1000
Athlon64	30	150	299	599	1498	2988
P4Hyper	37	186	371	741	1853	3702
Athlon32	38	192	382	764	1915	3826
Average(sec.)	35	176	350.67	701.33	1755.33	3505.33

5.2 Results for Condor

The first Grid benchmark was simply submitting all the tasks in single task jobs, using Condor's *Queue X* command mentioned in 3.3. There is no significant difference in the total time used by the different runs, as shown in Table 3. Thus, the discussion about the HyperThreading capability of one of the nodes from Section 4.1, seems not have any significant implication for single task jobs. However, since all three benchmarks are identical, the scheduling should be very similar in each run.

Since each task only takes 3-4 seconds, scheduling and file transfer overhead might add a significant delay to the total job execution time. The results in Table 4, compared to Table 3, show that this is indeed the case. In the meta-task test, multiple tasks were sent in fewer jobs, lowering the

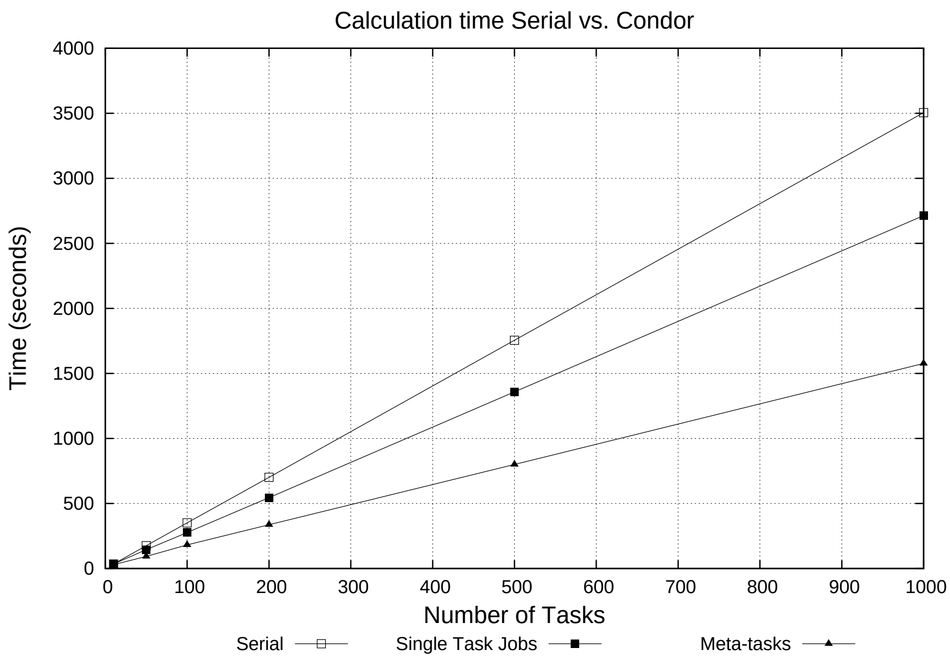

Figure 1: Calculation time serial vs. Condor

Table 3: Condor single task job execution time

Run/nJobs	10	50	100	200	500	1000
1	34	149	275	543	1353	2713
2	34	142	277	552	1362	2744
3	35	143	278	540	1358	2742
Average(sec)	34.33	144.67	276.67	545	1357.67	2733

Table 4: Condor meta-task execution time

Run/Jobs x Tasks	5x2	5x10	10x10	10x20	10x50	10x100
1	30	91	183	338	799	1577
2	30	94	181	336	802	1575
3	30	95	181	335	800	1577
Average(sec)	30	93.33	182.67	336.33	800.33	1576.33

Table 5: Speedup using Condor compared to serial execution

nTasks	10	50	100	200	500	1000
Single task jobs	1.02	1.22	1.27	1.29	1.29	1.28
Meta-tasks	1.17	1.89	1.93	2.09	2.19	2.22

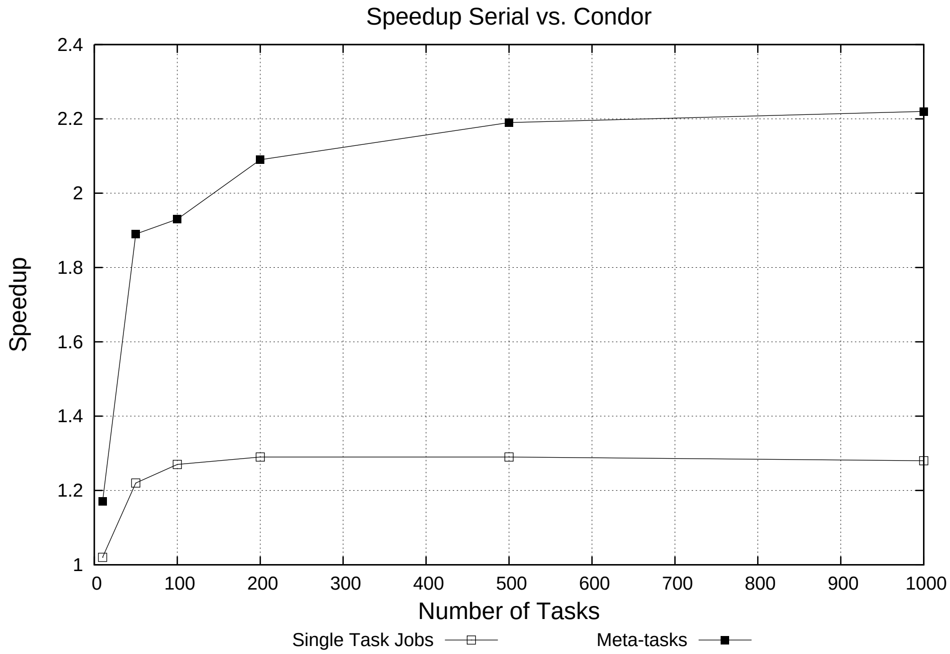

Figure 2: Speedup serial vs. Condor

scheduling and file transfer delays considerably as the task count increased. There is an even smaller difference in time between the different runs in this test than the former. One should think that if the [P4Hyper] machine was given two 100-task jobs while another node is idle, would give more difference. Disabling the HyperThreading feature altogether, showed little difference in the timing results for Condor. Furthermore, more benchmarks with different job-task ratios, should be run for a more secure conclusion. This will be considered future work.

Table 5 shows the speedup of the two former benchmarks. Sending each task as a single task job peaks at about 1.29 speedup, which is not very good keeping in mind the use of three times the computing power. Meta-tasks, however, show a much higher speedup. Thus, it seems that for such short tasks as in our case (3-4 sec), one can gain a lot from submitting multiple tasks together, when using Condor. A speedup of 2.2 is seen for 1000 tasks compared to serial execution, which in turn is a speedup of factor 1.73 compared to single task jobs. Fewer result files are transferred back, though their size are larger according to the number of tasks in the respective jobs.

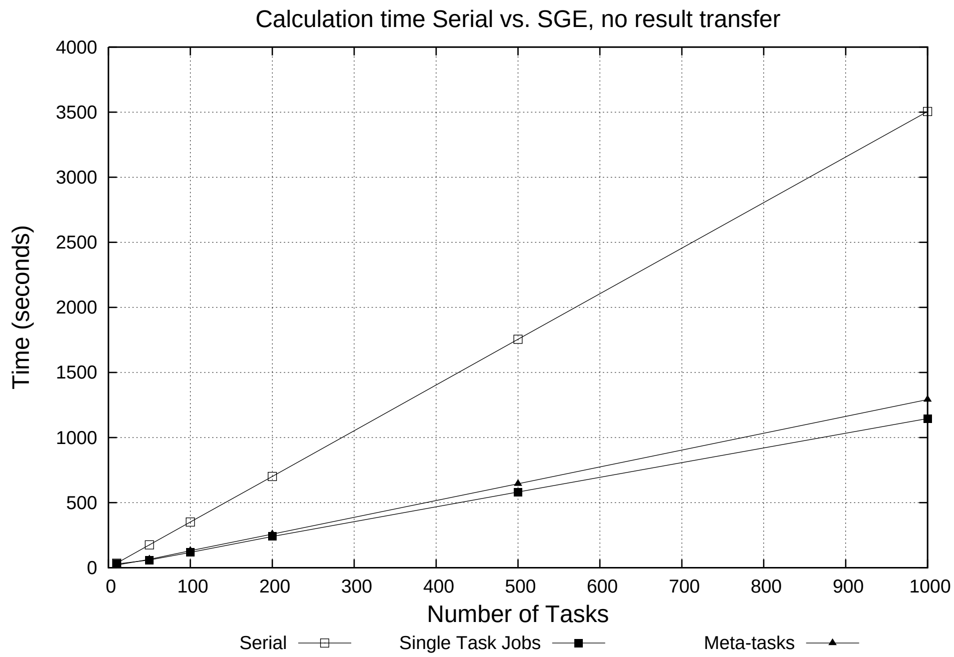

Figure 3: Calculation time serial vs. SGE (no result transfer)

5.3 Results for Sun Grid Engine (without result file transfer)

Table 6 shows very good results for single task jobs on SGE, even faster than Condor's meta-task executions. Compared to the average serial calculation time, a speedup factor of the number of nodes used in this small Grid is seen. This shows that all the extra file transferring done by Condor creates a significant amount of overhead. Apparently, SGE has nearly nonexistent overhead for these particular tasks when not transferring the results back to the submitter.

Table 7 shows an interesting result. It actually shows slower performance for meta-tasks than for single task jobs. Some tweaking of the job-task ratio was performed and generally showed that the more jobs submitted (with fewer tasks), the closer the timings came to the single task jobs. Thus, it seems that for SGE, single task jobs submitted as array jobs, will perform equal to or better than meta-tasks when not transferring the results back to the submitter after each execution.

5.4 Results for Sun Grid Engine (with result file transfer)

In this benchmark, an epilog script is used to transfer the result files for all tasks back to the submit host. This benchmark is run for better comparison to Condor, as Condor transfers all result files back to the submit host

Table 6: SGE single task job execution time (no result transfer)

Run/nJobs	10	50	100	200	500	1000
1	26	59	117	249	588	1145
2	31	59	116	227	587	1145
3	30	60	118	242	571	1147
Average(sec)	29	59.33	117	239.33	582	1145.67

Table 7: SGE meta-task execution time (no result transfer)

Run/Jobs x Tasks	5x2	5x10	10x10	10x20	10x50	10x100
1	21	64	130	259	644	1293
2	21	65	130	259	644	1287
3	20	65	131	258	648	1294
Average(sec)	20.67	64.67	130.33	258.67	645.33	1291.33

Table 8: Speedup using SGE compared to serial execution (no result transfer)

nTasks	10	50	100	200	500	1000
Single task jobs	1.21	2.97	3.00	2.93	3.02	3.06
Meta-tasks	1.69	2.72	2.69	2.71	2.72	2.71

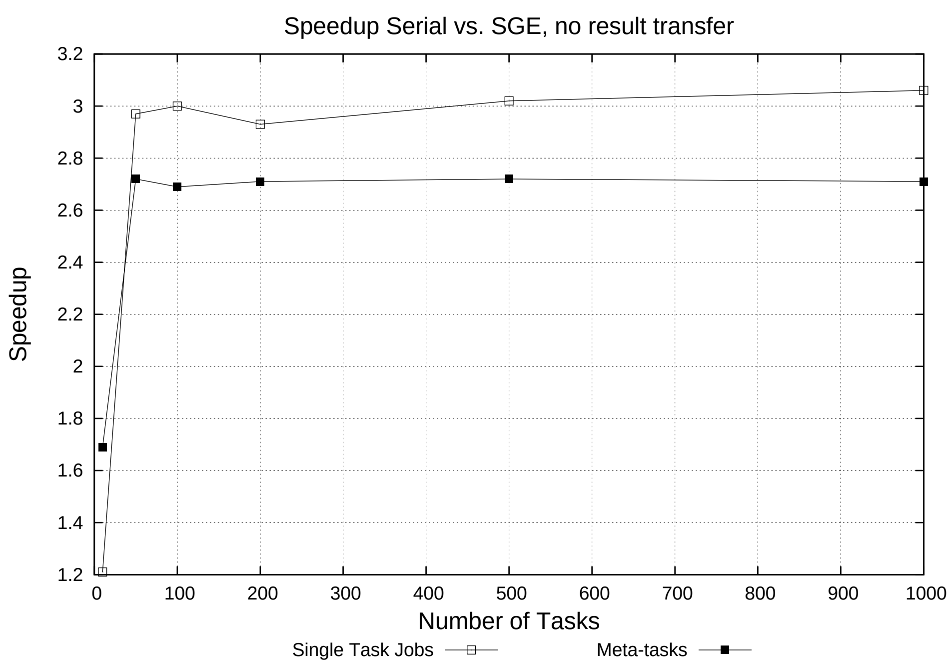

Figure 4: Speedup serial vs. SGE (no result transfer)

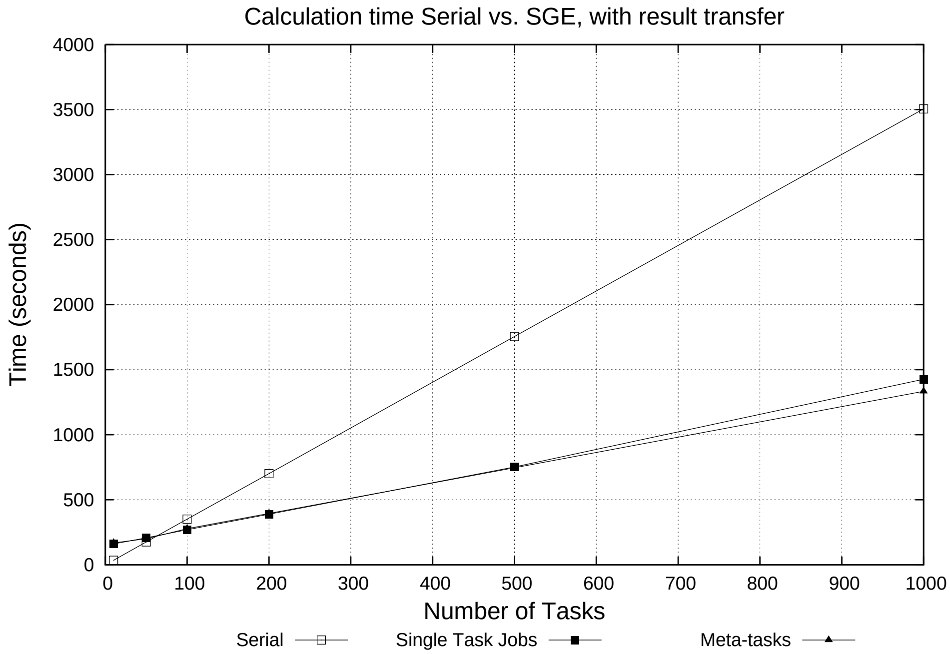

Figure 5: Calculation time serial vs. SGE (with result transfer)

automatically. Both stdout and stderr will be transferred for comparison, although one can choose not to transfer the stderr files if they are not needed. Actually, since the epilog transfer script is a regular bash script, one can choose to transfer whatever, in whichever way found suitable.

The results in Table 9, shows the execution time with result transfer back to the submit host. Compared to the single task jobs without result file transfer from Table 6, the numbers are generally higher, especially for small task collections. It is actually slower than serial execution for 10 and 50 tasks.

In Table 10, it is observed that the meta-tasks with the chosen job-task ratios, perform almost equally well as single task jobs. This result is quite different from the former benchmark, shown in Table 7, where much slower performance was observed for meta-tasks than for single task jobs. Thus, it seems that the extra file transferring levels out the performance between the two.

In Table 11, it is observed that when transferring result files for SGE, the performance is almost identical for both meta-tasks and single task jobs. In Section 5.5, it is observed that this was arbitrary, and that it is possible to tweak the submit scripts in favor of meta-tasks.

Table 9: SGE single task job execution time (with result file transfer)

Run/nJobs	10	50	100	200	500	1000
1	160	207	267	388	750	1354
2	161	207	270	389	752	1461
3	161	206	270	390	753	1465
Average(sec)	160.67	206.67	269	389	751.67	1426.67

Table 10: SGE meta-task execution time (with result transfer)

Run/Jobs x Tasks	5x2	5x10	10x10	10x20	10x50	10x100
1	168	200	277	396	746	1334
2	169	200	278	394	746	1335
3	168	200	278	395	745	1332
Average(sec)	168.33	200	277.67	395	745.57	1333.67

Table 11: Speedup using SGE compared to serial execution (with result transfer)

nTasks	10	50	100	200	500	1000
Single task jobs	0.22	0.85	1.30	1.80	2.34	2.46
Meta-tasks	0.21	0.88	1.26	1.78	2.35	2.63

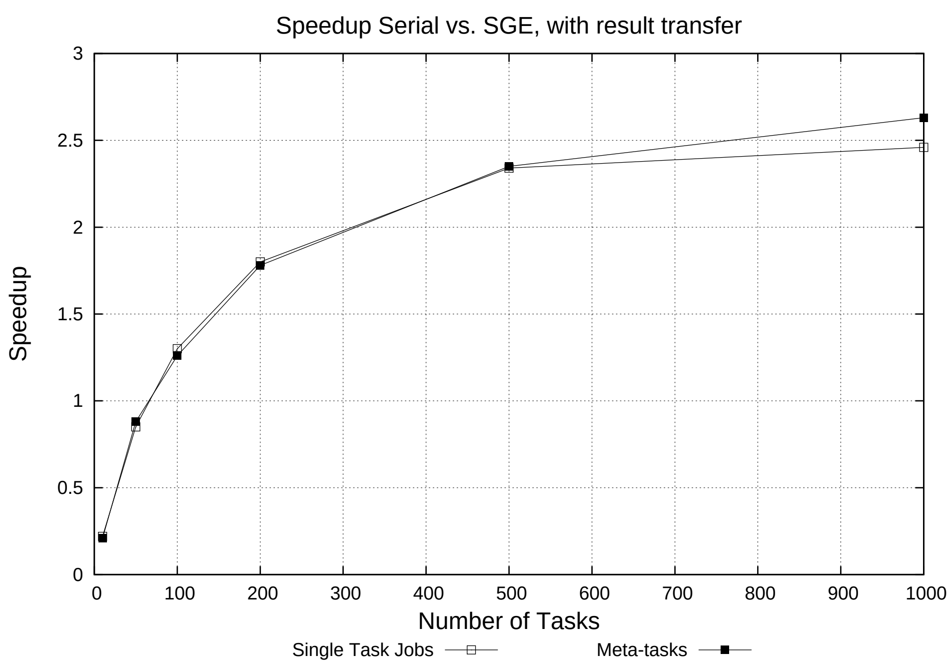

Figure 6: Speedup serial vs. SGE (with result transfer)

Table 12: Results for 1000 tasks when altering the job-task ratio

Grid/Jobs x Tasks	10x100	20x50	40x25	100x10
Condor	1576	1477	1462	1535
SGE (file transfer)	1333	1290	1246	1263

5.5 Results when altering the Job-Task ratio

Table 12 shows that there is room for fine tuning the job-task ratio for better performance. It was observed, during the benchmarking, that the [P4Hyper] machine was usually the last machine doing computations, on two jobs simultaneously. Hence, for jobs with large amounts of tasks, the two other machines were idle for a long time. Thus, it seems that Hyper Threading might actually give worse performance altogether, when used on nodes i a Grid. However, this machine could also be looked at as two slower machines, since each of its two jobs take about twice as long to finish as one on the [Athlon64]. Thus, it seems that this environment could benefit from dynamic scheduling, giving larger jobs to more powerful machines.

In any multi user Grid environment, however, the idle machines would be used to compute tasks from other jobs, by other users. Thus, when speaking of overall throughput in a Grid, this discussion is not equally important. The otherwise idle nodes will be utilized as long as there are other jobs in the queue.

Furthermore, the trend today is multicore processors without Hyper Threading technology, eliminating the discussion altogether. However, the dynamic scheduling idea still stands.

5.6 Result summary

Table 13 summarizes the speedup from all our benchmarks. It was observed, that the speedup quickly peaks at around 3, using SGE on three heterogeneous nodes. However, when looking at the details of this particular benchmark, this speedup did not include the transfer of result files from the execution nodes back to the submit node. When including the file transferring into the equation, the speedup, for SGE, was 2.46 for 1000 single task jobs. Another interesting result was actual slowdown when submitting a small number of tasks on SGE. Condor, however, was observed to have much better performance for small number of tasks.

The results show that SGE was about twice as fast as Condor for single task jobs, when transferring the results back to the submitter. Overall, Condor had bad performance when distributing the tasks in our application, using single task jobs. Using three nodes, only a peak speedup of 1.29 was observed. For comparison, an extra test was performed for Condor. This test

Table 13: Summary of speedup from all benchmarks
(Originally shown in Tables: 5, 8 and 11)

Benchmark/nJobs	10	50	100	200	500	1000
Condor single task	1.02	1.22	1.27	1.29	1.29	1.28
Condor meta-task	1.17	1.89	1.93	2.09	2.19	2.22
SGE single transf.	0.22	0.85	1.30	1.80	2.34	2.46
SGE meta-task transf.	0.21	0.88	1.26	1.78	2.35	2.63
SGE single no-transf.	1.21	2.97	3.00	2.93	3.02	3.06
SGE meta-task no-transf.	1.69	2.72	2.69	2.71	2.72	2.71

used the same bash script as for SGE's single task jobs, removing the transfer of the database file, as opposed to the regular Condor execution. This gave a speedup of 1.43 for 1000 tasks, which was only slightly better compared to the original Condor benchmark. Thus, it seems that transferring the binary from the submit node or fetching it from an NFS server, including removing the database transfer overhead, yield only slightly better performance.

In Section 5.5, it is observed that there is room for fine-tuning the job-task ratio for meta-tasks. A performance increase of 7-8% was observed, using 40 jobs with 25 tasks compared to 10 jobs with 100 tasks.

The benchmarks in this paper did not include the collection of the end results. The user will expect the same end results in the database as for serial execution without having to manually enter it. Without any means of automatically extracting and saving the end result from the distributed calculations, users may become more reluctant towards using Grid technology. The time saved in distributing calculations is lost in collecting and extracting the end result. For Condor, using DAGMan possible solution, where one could add a result gathering job, as dependent on the rest of the calculations. For SGE, the newly released ARI functionality could be used to add post jobs dependent on an array of jobs. These methods are only proposed in this project, and not thoroughly tested.

6 Conclusions and Future Work

In this project, we have seen how a commercial application, developed serially without any initial thought of parallelism or distributed calculation functionality, can benefit from being used in a Grid environment. Two well known Grid middlewares, Condor and Sun Grid Engine, were considered in the process, and ease-of-use evaluated. A discussion of installation procedures and problems can be found in the Appendices as well as practical job submission examples for the respective middlewares.

Due to the short execution time of our tasks, different methods for minimizing scheduler overhead were proposed, including altering the source code of our application to make it execute multiple tasks internally, tuning the Grid schedulers, and implementing the Master-Worker paradigm.

The first alternative would entail altering the input parameter list and the source code corresponding to the task computations, to make the application execute multiple tasks internally. This was believed not to have considerable speedup compared to our multiple task job proposal. The only time saved was assumed to be the starting and stopping of the executable for each task, and, if internal result comparison was implemented, fewer files would have to be compared by the last result script. However, this was only an assumption and needs further evaluation before a concrete conclusion can be taken.

In this project, however, only methods using regular Grid submit scripts were analyzed. Thus, all speedup results were gained without altering the source code in any way.

Grid schedulers have multiple parameters and features which can be fine tuned in different ways. By removing unnecessary features and fine tuning different timing constraints, scheduler overhead can be reduced. Removable and tunable features include scheduler monitoring, job validation, load adjustments and different scheduling timings. More information can be found on the web pages for respective Grid systems.

Implementing the Master-Worker paradigm proposed for Condor, is considered to give easy access to a heterogeneous environment. The MW-paradigm describes three classes that would have to be implemented, namely Driver, Task and Worker. These classes are used to describe, generate and execute tasks coherently and fast. MW is shown to be easily implemented in certain serial applications, with good results [6]. However, this solution was found to be outside the scope of this project.

The results gained in this project, show that the effort needed for installing a local Grid system in an organization, may be well worth it. Automatic distribution of tasks to nodes with idle CPU cycles, would give effective utilization of already available computing resources. For certain applications, no source code needs to be altered to make good use of a Grid environment.

6.1 Future work

In future work, dynamic scheduling of the meta-tasks, sized to better fit the different nodes in the Grid, would be of interest. From Table 2, it shows that the Athlon64 machine is about 27% faster than the Athlon32. Could the Athlon64 be sent bigger meta-tasks than the Athlon32 to make them finish at the same time? Or will it level out automatically when enough jobs are in the queue?

Methods for dynamic scheduling is found important for heterogeneous environments, like Grids. Different methods are proposed to handle different aspects of heterogeneous environments. These include, handling dynamic network bandwidth, decreasing makespan of meta-tasks of different size, and on-line dynamic scheduling algorithms, using dedicated scheduling processors [16, 8, 11]. Combining *SiteRank* [19] with a method for dynamic sizing of meta-tasks, is an interesting idea for future work.

References

[1] J.H. Abawajy. Job scheduling policy for high throughput computing environments. *Parallel and Distributed Systems, 2002. Proceedings. Ninth International Conference on*, pages 605–610, 17-20 Dec. 2002.

[2] Mark Baker, Rajkumar Buyya, and Domenico Laforenza. Grids and grid technologies for wide-area distributed computing. *Softw. Pract. Exper.*, 32(15):1437–1466, 2002.

[3] R. Buyya, D. Abramson, and J. Giddy. Economy driven resource management architecture for computational power grids, 2000.

[4] EGEE. http://egee.cesnet.cz/en/info/applications.html.

[5] Sun Grid Engine. http://www.sun.com/software/gridware/.

[6] Goux, J.-P., Kulkarni, S., Linderoth, J., and Yoder, M. An enabling framework for master-worker applications on the computational grid. In *The Ninth International Symposium on High-Performance Distributed Computing*, pages 43–50, May 2000.

[7] GridCafé. http://gridcafe.web.cern.ch/gridcafe/gridatcern/lcg.html.

[8] B. Hamidzadeh, D. J. Lilja, and Y. Atif. Dynamic scheduling techniques for heterogeneous computing systems. *Concurrency: Practice and Experience*, 7(7):633–652, 1995.

[9] J. Herrera, E. Huedo, R. S. Montero, and I. M. Llorente. Execution of typical scientific applications on globus-based grids. In *ISPDC '04: Proceedings of the Third International Symposium on Parallel and Distributed Computing/Third International Workshop on Algorithms, Models and Tools for Parallel Computing on Heterogeneous Networks (ISPDC/HeteroPar'04)*, pages 177–183, Washington, DC, USA, 2004. IEEE Computer Society.

[10] Eduardo Huedo, Ruben S. Montero, and Ignacio M. Llorente. Experiences on adaptive grid scheduling of parameter sweep applications. *pdp*, 00:28, 2004.

[11] Hung-Yuan; Liu Kang-Yuan; Chang Gei-Ming; Lien Chin-Chih Lee, Liang-Teh; Chang. A dynamic scheduling algorithm in heterogeneous computing environments. *Communications and Information Technologies, 2006. ISCIT '06. International Symposium on*, pages 313–318, Oct. 18 2006-Sept. 20 2006.

[12] Miron Livny, Jim Basney, Rajesh Raman, and Todd Tannenbaum. Mechanisms for high throughput computing. *SPEEDUP Journal*, 11(1), June 1997.

[13] Liang Peng, Lip Kian Ng, and Simon See. Yellowriver: A flexible high performance cluster computing service for grid. In *HPCASIA '05: Proceedings of the Eighth International Conference on High-Performance Computing in Asia-Pacific Region*, page 553, Washington, DC, USA, 2005. IEEE Computer Society.

[14] Liang Peng, Simon See, Jie Song, Appie Stoelwinder, and Hoon Kang Neo. Benchmark performance on cluster grid with ngb. *ipdps*, 18, 2004.

[15] Condor Project. http://www.cs.wisc.edu/condor/.

[16] Prashanth C SaiRanga and Sanjeev Baskiyar. A low complexity algorithm for dynamic scheduling of independent tasks onto heterogeneous computing systems. In *ACM-SE 43: Proceedings of the 43rd annual Southeast regional conference*, pages 63–68, New York, NY, USA, 2005. ACM.

[17] MW Team. http://www.cs.wisc.edu/condor/mw/.

[18] Douglas Thain, Todd Tannenbaum, and Miron Livny. Distributed computing in practice: the condor experience. *Concurrency - Practice and Experience*, 17(2-4):323–356, 2005.

[19] George Tsouloupas and Marios D. Dikaiakos. Grid resource ranking using low-level performance measurements. In Anne-Marie Kermarrec, Luc Bougé, and Thierry Priol, editors, *Euro-Par*, volume 4641 of *Lecture Notes in Computer Science*, pages 467–476. Springer, 2007.

A Setting up the Grid

A.1 Sun Grid Engine 6.1

SGE master host and execution host was successfully installed on Fedora 7 by following the installation manual found on the Sun web page [5]. After working out the problems mentioned in A.1.3, the installation was pretty straight forward. There were a lot of steps to set up different things and it is a good idea to have a plan or basic idea of the Grid before installing.

To install an execution host, it is necessary to copy all the installation files from the master host to the execution host after the master host installation. This way the execution host will get the correct settings set up for the master host.

A.1.1 Possible node configurations

There are several different node functions available for SGE:

- Master host: this is where the Grid engine runs.
- Shadow master host: this is a backup host if the master host fails. There can be several shadow hosts in a Grid.
- Administration host: nodes that can do administrative tasks on the Grid.
- Submit host: a node which can submit and control jobs.
- Execute host: a node where jobs are executed.

A.1.2 Access to files

Each node in a SGE Grid, needs all executables and input files locally or on a NFS/AFS mount. There is no automatic file transfer in SGE, like in Condor. However, prolg- and epilog scripts can be defined for a queue, where file transfer or other operations can be defined. Results are copied to the respective user's home directory. SGE expects the home directories to be mounted on NFS. If they are not, all results will be copied locally on the execution node. An epilog script can then be used to automatically transfer all results back to the submit node. An example of such a script is shown in Appendix C.

A.1.3 Problems

Some problems were encountered while installing the master host:

- SGE needs the libXm.so.3 library, which can be found in the OpenMotif package, for its GUI application. OpenMotif-2.3.0.0.fc7.ccrma.i386.rpm for fedora core 7 was installed which had the newer version, libXm.so.4, of the library. I had to make a symlink to this file for the SGE GUI to work (since it is a newer version, the linker is happy):

  ln -s /opt/openmotif/usr/lib/libXm.so.4.0.0 /usr/lib/libXm.so.3

- After unpacking the files, the command **setfileperm.sh $SGE_ROOT** is to be run to set the right permissions. This failed because of a wrong

GLIBC version in Fedora 7. To fix this, open the file *"$SGE_ROOT/util/arch"* and edit line 248 from 3|4|5) to 3|4|5|*) and run the script again. (NB! This problem did not appear on a machine running Kubuntu)

A.1.4 Windows restrictions

Windows machines cannot run as master hosts, shadow master hosts or scheduler. Windows is therefore limited to execution and submit hosts. Certificates (Certificate Security Protocol (CSP)) are also necessary for communication between master host and windows execution host. The GUI tool *qmon* and DRMAA are not supported either.

A.2 Condor-6.8.6

I installed Condor with a rpm package on Fedora 7, with a tar.gz package on Kubuntu and with a MSI package on WindowsXP. The only prerequisite on Linux was an older version of the libstdc++ library. This can be installed with **yum install compat-libstdc++-33** for Fedora 7 or **apt-get install libstdc++5** on Kubuntu. The installation was easily completed by following the installation manual found on the Condor web page [15]. The installation of a 4 node Condor pool, including a WindowsXP node, was an hours process. On SGE, it was more a days process, for a Grid beginner.

Registering an execute and submit node on the master host in Linux, simply enter the following command on the host to add:

./condor_configure –central-manager=host@domain.com –type=execute,submit

Make sure all the nodes networking is set up correctly before running this command. If not, the request might not be detected by the master host.

A.2.1 Possible node configurations

- Master host (even Windows nodes, unlike SGE)
- Execute host
- Submit host
- Or any combination of these

A.2.2 Access to files

In Condors standard universe, access to files, input and output, is handled automatically through remote system calls. In other universes, vanilla, Java and MPI, access is presumed, as default, to be through a shared file system on UNIX machines. If no shared file system is available, file transfer has to be specified in the submit files by the user. Add the following lines to the submit script to enable file transfer (other options are available):

```
should_transfer_files = YES
when_to_transfer_output = ON_EXIT
transfer_input_files = file1, file2, ...
```

A.2.3 Heterogeneous nodes

In a heterogeneous Grid, it is beneficial to have as much information about each node as possible. If a job has specific needs, special care should be take to which node the jobs are sent to. Condor solves this with the ClassAds system. Each node has a set of parameters (e.g. Total Memory, OpSys, Architecture, Disk Space), which can be used as requirements for jobs. Condor administrators can specify their own ClassAds for each node, as well. Inserting the following in a submit file, states that the job needs the Linux OpSys and flavor RedHat9 *(OPSYS_FLAVOR="RedHat9" has to be defined in the execution nodes configuration file for it to be eligible)*:

REQUIREMENTS = (OpSys == "LINUX") && (OPSYS_FLAVOR =?= "RedHat9")

This is useful if the executable is compiled only for a specific flavor of Linux, with possibility to fail if executed on, for example, Debian. A neat trick to make use more nodes, would be to have different executables for different OS's and flavors, and specify which executable should be transferred to the different nodes. The following line will choose the correct executable for the specified requirement.

Executable = exec.$$(OpSys).$$(OPSYS_FLAVOR)

The executables must have names similar to *exec.LINUX.RedHat9* or *exec.LINUX.Debian,* for this to work.

A.2.4 Problems

I installed a net install of Debian on a 3rd node and ran into a problem with ip-addresses and hostnames. The installation defined the IP address for the node's hostname, in the */etc/hosts* file, to a local one (127.0.1.1). This caused the master to block access for the node, because it had an unknown domain address. Commenting out this line and letting DHCP take care of hostname and corresponding ip-addresses, fixed the problem.

A.2.5 Windows restrictions

It is not possible to condor_compile Windows applications. As a result, remote system calls and checkpointing is not available on this platform. Therefore, Windows jobs have to run in the "vanilla" universe. The following can be added to the submit script, to run on Windows machines:

```
universe = vanilla
requirements = (OpSys == WINNT50)
```

B Using the Grid

B.1 Sun Grid Engine 6.1

To start the SGE daemons, enter the following commands as root:
$SGE_ROOT/name-of-cell/common/sgemaster start
$SGE_ROOT/name-of-cell/common/sgeexecd start

B.1.1 Submitting Jobs

Submitting jobs to SGE, is done by sending batch scripts to the master server with the command **"qsub /path/to/script.sh"** or through the GUI interface *qmon*. A computer is not allowed to submit jobs, unless it is registered as a "Submit Host". Once this is done, jobs sent from this node will be accepted and put in a job queue.

The results are handled differently than in Condor. Condor copies all results back to the same folder, on the node it was submitted from, simply by running in the *"standard"* universe, or by defining file transfer in the submit script. SGE copies the results to the owners home directory. Thus, SGE assumes the users home directories are NFS mounted. If the home directories are not NFS mounted, the results are copied locally on the execution node that completed the job. This can be tricky, as you don't know which node your job is executed on. Epilog scripts can be used to remedy this, as described in Appendix A.1.2.

SGE does not have automatic job executable transfer, like Condor. Each job executable must therefore be available on every node, in the path given in the bash script. The easiest way to make sure this is the case, is to have all the files available on AFS of NFS. However, it is possible to manually define transfer of executables in a prolog script, if necessary.
Example of a simple SGE submit script:

```
#!/bin/sh
#
# This is a simple example of a SGE batch script
# SGE specific options starts with '#$'
```

```
#$ -S /bin/sh
#$ -o output_file_name
/path/to/executable arg1 arg2
```

B.2 Condor-6.8.6

To start Condor, enter the following commands as root:
$CONDOR_ROOT/sbin/condor_master

B.2.1 Preparing jobs for execution

To use remote system calls and checkpointing/migration in Condor, the executable has to be relinked with the Condor libraries. Here a problem was encountered on both Fedora 7 and Kubuntu: "ERROR: Internal ld was not invoked! Executable may not be linked properly for Condor!". A solution was not found and the jobs were run in the "vanilla" universe instead. Condor_compile had problems locating some application libraries, thus a static link of the executable could maybe have fixed the problem.

B.2.2 Submitting Jobs

Submitting jobs is done with the command **"condor_submit job.cmd"**. The .cmd file contains different job settings, including input/output file locations, files that have to be transferred to the execution node, the required architecture for running a binary, etc. Setting up correct constraints and requirements for a job, will help make sure the job is executed successfully. Copying directories of input files, is not supported in the current version of Condor. For jobs with directories as input data, a shared file system can be used for input files instead. ClassAds can be used to find a node which has access to the specific remote location.

Example of a simple Condor submit script, that copies the executable to the execute node and returns the results to the submit node (notice the use of the "standard" universe):

```
#
# Test Condor command file
#
universe = standard
executable = name_of_executable
output = executable.out
error = executable.err
log = executable.log
arguments = arg1 arg3
queue
```

See manual for more options:
`http://www.cs.wisc.edu/condor/manual/v6.8/condor_submit.html`

B.2.3 Different settings

Condor can either be set up to run jobs on any node, idle or not, or to nodes which have been idle for more than 15 minutes (either no keyboard, no mouse movement or CPU idle time). When the node is no longer available, the job can be checkpointed and kept on the same node until idle again, or the job can be sent to another idle node. During testing, it is recommended use Condor in testing mode, to disable the "wait for idle" settings. See the manual for details on how to do so.

C Scripts and Code Listings

Here follows a listing of all scripts used in benchmarking Condor and SGE. The last listing is the C++ program used to extract the best result from the pre formatted result file, described in Section 3.6.2.

Listing 1: Job script for SGE

```
#####################
# Single Task Job Script for Sun Grid Engine
# Single job submission:
#  qsub this_script.sh
# Array job submission:
#  qsub -t [t_first]-[t_last]:[t_stepsize] this_script.
   sh
#####################
#!/bin/sh
# Request Bourne shell as shell for job
#$ -S /bin/sh

v1=$(date +%s)
# Run dagoc
for ((i=0; i<[set_num_tasks_here]; i+=1)) ; do
/mnt/project/dagoc -c -start=1 -stop=10 \
"/mnt/project/setups/TEST_sge.sup"
done

# Print Job Data
echo start=$SGE_TASK_FIRST stop=$SGE_TASK_LAST \
step=$SGE_TASK_STEPSIZE id=$SGE_TASK_ID

# Print time taken
v2=$(date +%s)
let v3=$v2-$v1
echo "Seconds used for this task: " $v3
```

Listing 2: Result transfer epilog script for SGE

```
#####################
# result_transfer_epilog.sh
# Transfers SGE array results to host
#####################
#!/bin/bash
echo SGE_HOST: $SGE_O_HOST
echo HOSTNAME: $(hostname -s)
if [ "$(hostname -s)" != "$SGE_O_HOST" ];
then
 echo "Transferring result to host..."
 scp $SGE_O_WORKDIR/dagoc.sh.o$JOB_ID.$SGE_TASK_ID \
     $SGE_O_WORKDIR/dagoc.sh.e$JOB_ID.$SGE_TASK_ID \
     $SGE_O_HOST:/$SGE_O_WORKDIR
 rm $SGE_O_WORKDIR/dagoc.sh.o$JOB_ID.$SGE_TASK_ID
 rm $SGE_O_WORKDIR/dagoc.sh.e$JOB_ID.$SGE_TASK_ID
fi
```

Listing 3: Script for extracting runtime for SGE

```
####################
# Get time used from Sun Grid Engine
# Useage:
# qacct -j [job_id] > sgejobsummary.txt
# ./this_script.sh sgejobsummary.txt
####################
#!/bin/bash

#Get job submit time
t0=$(grep "qsub_time" $1 | head -n 1 | \
grep -o [0-9][0-9]:[0-9][0-9]:[0-9][0-9])
s0=$(date --date=$t0 +%s)

#Get last job end time
t1=$(grep "end_time" $1 | \
grep -o [0-9][0-9]:[0-9][0-9]:[0-9][0-9] | \
sort -r | head -n 1)
s1=$(date --date=$t1 +%s)

#Get total time used for array job
let s=$s1-$s0
echo "File:" $1
echo "Time:" $s "seconds"
```

Listing 4: Condor single task jobs submit script

```
####################
# dagoc.condor
# 100 Single Task Jobs Condor command file
####################
universe      = vanilla
executable    = dagoc
output        = dagoc.out.$(CLUSTER).$(PROCESS)
error         = dagoc.err.$(CLUSTER).$(PROCESS)
log           = dagoc.log.$(CLUSTER)

REQUIREMENTS = (OpSys == "LINUX") && (OPSYS_FLAVOR =?= "FC7")
should_transfer_files = YES
when_to_transfer_output = ON_EXIT
transfer_input_files = /mnt/project/dbases/TEST.db
arguments = -c -start=1 -stop=10 /mnt/project/setups/TEST.sup
queue 100
```

Listing 5: Condor meta-task submit script

```
####################
# 10 Multi(10x)Task Job Condor command file
####################
universe      = vanilla
executable    = multiTaskJob.sh
output        = dagoc.out.$(CLUSTER).$(PROCESS)
error         = dagoc.err.$(CLUSTER).$(PROCESS)
log           = dagoc.log.$(CLUSTER)
REQUIREMENTS  = (OpSys == "LINUX") && (OPSYS_FLAVOR =?= "FC7")

should_transfer_files = YES
when_to_transfer_output = ON_EXIT
transfer_input_files = dagoc, /mnt/project/dbases/TEST.db
arguments = 10
queue 10
```

Listing 6: Condor meta-task bash script

```
####################
# MultiTaskJob .sh
####################
#!/bin/bash
v1=$(date +%s)
echo "Start: " `date`
for ((i=0; i<$1; i+=1)) ; do
    ./dagoc -c -start=1 -stop=10 /mnt/project/setups/
        TEST.sup
done
v2=$(date +%s)
let v3=$v2-$v1
echo "Finished: " `date`
echo "Seconds used: " $v3
```

Listing 7: Script for extracting runtime for Condor

```
####################
# Get time used from Condor log file
# Useage:
# ./this_script.sh nameOfLogFile.log
####################
#!/bin/bash

# Get submit time and last job end time
t0=$(grep "Job submitted" $1 | head -n 1 | \
grep -o [0-9][0-9]:[0-9][0-9]:[0-9][0-9])
t1=$(grep "Job terminated" $1 | tail -n 1 | \
grep -o [0-9][0-9]:[0-9][0-9]:[0-9][0-9])

# Convert to seconds and find total time
s0=$(date --date=$t0 +%s)
s1=$(date --date=$t1 +%s)
let s=$s1-$s0

# Print results
echo "File:" $1
echo "Time:" $s "seconds"
```

Listing 8: Condor DAGMan script

```
######################
# DAGMan script for
# post processing
######################
Job A dagoc.condor
Job B post.condor
Parent A CHILD B
```

Listing 9: The 'post.condor' script

```
######################
# post.condor
# Post processing job for condor
######################
universe   = local
executable = post.sh
output     = post.out.$(CLUSTER)
error      = post.err.$(CLUSTER)
log        = dagoc_dag.log
arguments  = 1 10 GROUP.FIELD.FGasa
queue
```

Listing 10: The result extraction bash script for single task jobs

```
####################
# post.sh
# Post processing script
# (Single task jobs)
# Arguments:
# 1: start 2: stop
# 3: result_tag 4: res_file_name
####################
#!/bin/bash
echo Collecting results from output files
s0=$(date +%s)
if [ -f result.post ]
then
  rm result.post
fi
for i in $( ls $4*); do
  echo JobID: $i >> result.post
  grep Setup $i >> result.post
  grep $3 $i >> result.post
done

echo ----------------------------------------
echo Getting best result from collection
echo ----------------------------------------
./findBestResult $1 $2 $3 result.post

echo ----------------------------------------
echo Running dagoc with the best setup
chmod 775 postDagmanScript.sh
./postDagmanScript.sh
rm postDagmanScript.sh

echo ----------------------------------------
echo Done!
s1=$(date +%s)
let s=$s1-$s0
echo "Time used for post processing: " $s "seconds"
```

Listing 11: Code for extracting and comparing results

```
/*####################
* findBestResult.cpp
*/####################
#include <sstream>
#include <iostream>
#include <fstream>
#include <string>
using namespace std;
//Input args: start stop resultTag
int main (int argc, char **argv) {
 if(argc != 5)
 {
  fprintf(stderr, "Wrong # of arguments!\n");
   return 1;
 }
 //Get input parameters
 int start = atoi(argv[1]);
 int stop = atoi(argv[2]);
 string tagResult = argv[3];

 string tagSetupFile("Setup file:");
 string line;
 double bestResult = -1.0;
 string setupFilePath;
 ifstream myfile (argv[4]);
 if (myfile.is_open())
 {
  while (! myfile.eof() )
  {
   getline (myfile,line);
   if(line.find(tagSetupFile,0) != string::npos)
   {
    //Save temporary best setup file path
    string tmpSetupFilePath = line.substr(tagSetupFile.length());

    //Read next line if "Setup Path" tag found
    getline(myfile,line);
    if(line.find(tagResult,0) == string::npos)
    {
     cout << "Did not find result tag for" << tmpSetupFilePath  << endl;
     continue;
    }
```

```
      //If "Result Tag" found parse the result
      double tmpRes = -1.0;
      string res = line.substr(line.find(":",0)+1);
      istringstream i(res);
      if(!(i >> tmpRes))
      {
       cout << "Res: " << res << endl;
       cout << "Could not parse result" << endl;
       continue;
      }

      //Save new best result and setup file path
      if( tmpRes > bestResult)
      {
       bestResult = tmpRes;
       //Remove last point
       setupFilePath = tmpSetupFilePath.substr(0, tmpSetupFilePath.length()-1);
      }
     }
    }
    myfile.close();

    //Create script with final dagoc execution
    ofstream outputFile;
    outputFile.open("postDagmanScript.sh");
    if(outputFile.is_open())
    {
     outputFile << "#!/bin/bash\n";
     outputFile << "./dagoc -c -start=" << start << " -stop=" << stop << " " << setupFilePath << " > DAGOCRESULT.RES\n";
     outputFile.close();
    }
    else cout << "Unable to open output file" << endl;

    //Print results
    cout << "Best result: " << bestResult << endl;
    cout << "Setup Path: " << setupFilePath << endl;
   }
   else cout << "Unable to open input file" << endl;

  return 0;
}
```