MQP GXS-0701

Grid Portal Application

submitted the Faculty

of the

Worcester Polytechnic Institute

in partial fulfillment of the requirements for the

Degree of Bachelor of Science

by

Montana Foertsch
Date:

Michael Rawdon
Date:

Richard Skowyra
Date:

Professor Gábor Sárközy, project advisor

Professor Stanley Selkow, project co-advisor

Abstract

This project concerns EMMIL (E-Marketplace Model Integrated with Logistics) and its viability as a grid application. The model was evaluated on a single processor and on the SEEGRID network using MTA-SZTAKI's P-GRADE Portal. A granularity heuristic was developed to guide the mapping of EMMIL datasets to processes. A portlet for P-GRADE Portal was also created to aid in data entry. Finally, pre-processing filters were added. These were designed to discard useless combinations and reduce overall computing time.

Acknowledgements

We would like to thank WPI and MTA-SZTAKI for providing us the opportunity to live and work in Budapest for seven weeks. We would further like to thank all members of the Laboratory for Parallel and Distributed Systems for their continued support and willingness to speak English when we were nearby. Drs. Péter and Livia Kacsuk did an exceptional job overseeing and guiding the direction of our project. Miklos Kozlovszky provided excellent counsel, and Gabor Hermann devoted many hours to assisting us in technical matters. Finally, we would like to thank Professors Gábor Sárközy and Stanley Selkow for making our presence here possible, and for the guidance they provided throughout the course of this project.

Table of Contents

Table of Figures

1 Project Statement

As much commerce becomes increasingly electronic, new marketplaces are necessary to ensure rapid and efficient transactions. Services such as Amazon.com and E-Bay have rapidly developed to facilitate such interactions between consumers and producers. However, business-to-business e-marketplaces are still an emerging field. A variety of factors, such as the quantity difference between retail and commercial orders, the nature of the products involved, and the relationship between buyers and sellers prevents a simple recycling of existing consumer models in many cases.

Specifically, methods are needed for automating the selection of sellers and shippers involved in a specific buy order. The E-Marketplace Model Integrated with Logistics (EMMIL) developed by Dr Livia Kacsukné Bruckner of Budapest International Business School is one such approach. In this model, a buyer places a buy order for an arbitrary number of products at arbitrary quantities. The system will then compute the best combination of sellers and logistics providers (shippers), attempting to minimize the total price paid [3]. (For a more in-depth explanation, see Section 2.5.) However, linear computations must be performed over many combinations of sellers, products, and shippers. On a single processor this might take an infeasible amount of time to arrive at an acceptable solution. Since the same algorithm is being applied to many sets of parameters, however, the problem lends itself to parallelization.

Our group worked with researchers at MTA-SZTAKI to implement a prototype EMMIL grid application. In addition, we had three independent objectives. First, the feasibility of using computational grids as an infrastructure for EMMIL marketplaces was determined through both single-processor and grid-based testing. Second, we created a portlet for P-GRADE portal to aid in rapid EMMIL data entry. Lastly, enhancements were made to the preprocessing algorithms in the EMMIL application. Product capacity limits were added, and a system of detecting and discarding hopeless combinations of seller, product, and shipper was implemented.

2 Background

Before discussing the methodology and results behind our project, some background information on both grid systems and the EMMIL model may be necessary. If the reader is already well-versed in these topics, this section may be considered optional. No data specific to our project are presented until Section 3. The origins and uses of grid systems are considered first, in Section 2.1. Problems that have arisen during their development and use are touched upon in Section 2.2. Next, a general overview of grid components and their layered representation is given in Section 2.3. Particularly common toolkits and libraries are investigated in depth in Section 2.4. Finally, the EMMIL system and its relationship to grid computing are explained in Section 2.5.

2.1 Grid Systems

Grid Systems provide computing power to users while abstracting the details of hardware platform, operating system, and physical location of the machine or machines doing the computation. A common analogy for grid computing is the power grid: a device can be plugged into an outlet and connected to the electrical grid, but the user of this device is not concerned with how or where this power was generated [11]. Grid computing has not yet reached this ideal level of transparency, as some level of knowledge is still required by the user. Ian Foster, a leader in grid development, created a three point checklist to both define and evaluate a grid:

- *"coordinates resources that are not subject to centralized control ...*
- *... using standard, open, general-purpose protocols and interfaces ...*
- *... to deliver nontrivial qualities of service".* [2][9]

The first of these points is critical. Each node of a grid is not directly managed by a central authority, but instead can be made up of systems from multiple domains with different policies. The second point ensures that the grid is useful in the completion of a range of tasks, and not specific to any specific application. The final point emphasizes that the grid must be able to guarantee a certain level of service. This is also crucial, as without guaranteed quality of service the effectiveness of a grid system is lost.

2.1.1 History

The concept of grid computing arose from an earlier concept, which was developed in the 1980s. Metacomputing, like grid computing, is intended to make computing power from different resources transparently available to users. The major difference between grid computing and metacomputing is that the intent of the latter is to provide transparent access by clients to supercomputers, rather than to a heterogeneous network of machines (as is the case in grid computing) [24].

In 1995 a metacomputing infrastructure was developed that became the basis of many future grid systems. This infrastructure, known as I-WAY, was demonstrated with great success at the Supercomputing '95 conference where 60 different applications were demonstrated on the system [25]. I-WAY's success led the United States DARPA to fund a project to develop a toolkit for distributed computing. To head this initiative they selected Ian Foster, a lead researcher in the I-WAY project. This led to the development of the Globus Toolkit, which many current grid systems now use [2].

2.1.2 Uses

Grids have gained popularity in a number of different areas, ranging from academic to industrial and even entertainment settings. Grids provide a number of high level benefits to users, including increased agility in product development, reduced risk in decisions, and increased potential for innovation. These benefits are all a result of the large amount of parallel data that grid resources can process in a short amount of time, when compared with a traditional computer system.

A number of powerful grids have been developed in academic settings, which utilize the knowledge of computer systems within existing IT and computer science departments. Houston University Campus Grid is an excellent example of a grid in an academic setting; it makes use of a heterogeneous computational grid to help solve a wide range of computationally intensive jobs from a wide range of disciplines. These jobs include the modeling of atmospheric pollution, the analysis of seismic data for oil and geophysical service companies, and many others from disciplines such as mechanical engineering, computer science, and mathematics. The Ontario HPC Virtual Laboratory (HPCVL) is another interesting example of a campus grid, being composed of machines located at four different universities in Ontario. This grid helps to solve problems not just introduced by researchers from the universities involved, but also those which are considered important to the economy of Ontario and Canada as a whole. For instance, the HPCVL provides computing resources to the Sudbury Neutrino Observatory, whose work was selected by *Science* in 2002 as the second most important discovery of the year. These grids not only provide computational power to help solve important problems, but have also helped in the development of grid systems as a whole[2].

Bioinformatics is another area where grid computing has been widely used. Work in this field centers on the analysis of patterns within genetic databases to find previously unknown connections and similarities. Genetic databases are extremely large, and the comparisons lend themselves well to parallel computing. Bioinformatics has aided in the development of treatments for many diseases, including anthrax and smallpox[2]. Grid computing has also proved useful in the modeling and creation of new compounds to help to treat diseases, by testing the possible effectiveness of large numbers of synthetic molecules [2]. These processes have enabled researchers to solve many genetic and molecular problems much faster than was previously possible.

Grid computing has also proved effective in a wide range of commercial and industrial settings. The financial sector has effectively utilized grid computing to greatly speed up their ability to make financial predictions: computations that previously took over ten hours can now be completed in mere minutes. This allows traders to constantly evaluate the risks that they have taken, and continually reevaluate their position in the market [2]. Certain sectors of the manufacturing community are also implementing grid-based solutions. For example, the automotive industry now simulates a large number of collision tests over grid networks. Similarly the aerospace industry conducts many simulations on grid computing systems that could previously only be completed with expensive prototypes in wind tunnels [2]. Another interesting application of grid computing occurs in the electronic gaming industry, where scalable infrastructures have been developed for massively multi-player online games [2]. The benefits of grid computing have been felt in a wide range of industries, and its uses are only increasing as its influence spreads.

2.1.2.1 Parameter Studies

Grid systems play a particularly important role in parameter studies: applications with a single algorithm that should be executed over a large set of input parameters. As this type of task can easily be split into a number of smaller tasks capable of running in parallel, it is a natural fit for grid computing. A number of projects have provided implementations of parameter studies within grid systems. One notable implementation was developed by Worcester Polytechnic Institute students and SZTAKI scientists for the P-GRADE portal [15]. It integrates grid workflows and parameter studies into a single concept [18]. This is described in more detail below.

2.2 Problems in Grid Development

Computational grids must overcome a large number of difficulties in order to be seen as reliable and efficient from the user's standpoint. Grids must allow "flexible, secure, coordinated resource sharing among dynamic collections of individuals, institutions, and resources" [7]. This sharing of resources requires a term to describe the different entities involved in the net of resources provided and used. Virtual Organizations (VO) represent a body of individuals or institutions that consumes resources, provides them, or both. VOs on the grid host resources that include both computation and storage elements. Virtual Organizations must also recognize each other, and have specific rules defined regarding which other VOs are allowed to access their shared resources and in what manner these client Virtual Organizations may use these resources [7].

This restriction on the resources available to VO's requires a central security authority that will define resource availability for any Virtual Organization. A Certificate Authority (CA) fills this need. Within a Grid, a Grid Security Infrastructure (GSI) provides security functionality [27]. This GSI provides a common security method among all of the grid resources. Entities within this grid are provided with certificates that allow them to authenticate themselves to other entities which in this case can be users or processes. Gateways can convert certificates between different infrastructures and site resource domains [27].

CA's in a Grid generally validate user certificates, although they may also create these certificates. As long as a VO trusts the CA who signed off on a user's certificate, that VO will trust that certificate. This alone does not allow the user access to a VO's resources. A user begins to attempt access to VO resources by authenticating itself to a Community Authorization Service (CAS), which returns assertions to the user defining how that VO's resources may be used. The user then presents this assertion to a VO resource along with a request for resources. The resource checks the CAS policy statement against its local policy statement, and may then allow the client to utilize the resource [27].

2.2.1 Resource Management

Remote resources are required for many grid applications. After security concerns are taken care of, the grid must still manage the location of remote files, transfer content to active jobs, control load balancing for grid tasks, handle any generated errors, and always appear reliable and efficient from a user's perspective [6].

Physical file systems on the grid exist at remote locations in the same way computing elements do. As a great deal of information can be generated by grid tasks during execution that may need to be consumed again at a later point, it is infeasible to send this information to the client and then request it back again. Remote file systems provide large storage space for storing large batches of output data in addition to these temporary files. The system for file handling by the Globus Toolkit is described in Section 2.4.2.2.

2.2.2 Load Balancing

Load balancing among computing elements of grid resources is another important issue that grids need to deal with. Predicting the future computational needs of processes is extremely difficult, and due to the rate at which loads on a resource can change load balancing systems must respond very quickly to prevent further processes from being allocated before the client realizes the resource is completely occupied.

2.2.3 Information Management

Due to the dynamic nature of the grid information services are required to keep all grid users updated with the status of a grid. New resources may be added as new VOs join a grid, or resources may leave in the event of a crash or disconnect. Tasks being executed may change their resource requirements, such as increasing the strain on resources or reducing it. A discovery service is required to discover currently available resources. Schedulers are required to determine the best resource(s) to allocate to a task. Application adaptation systems watch running applications and resource availability and modify applications behavior to improve system performance [5].

These information services necessarily generate very large quantities of information that may take a long period of time to be received by those agents affected and it may be impossible for the grid to delay actions and await a response. These information services must also be able to adapt to failures in the grid system. Information systems have to be distributed in order to handle possible node failures within the grid [5].

2.2.4 Quality of Service

Grids have difficulty enforcing a standard quality of service. Since jobs running on a grid can behave in unpredictable ways, their performance can affect other jobs which are using the same resources in unforeseen ways. Different resource types can also affect the performance of grid tasks. Many applications are written to run on specific systems, and have issues running on many different environments within a grid. Grids attempt to solve these issues by using virtual workspaces [19].

Virtual workspaces are abstractions of execution environments that can be made available to authorized clients. These workspaces allow programs designed to run on a certain system to be run on grid resources configured differently. One option is for the resources themselves to reboot and load a new configuration more suitable to the task. Host machines can also run a Virtual Machine (VM) that can support this abstraction of another system by emulating instructions issued by the user's task. This VM can allow a user to create a custom environment, and also allows strict enforcement of resource usage. A VM is configured with a set available memory and disk space which can keep resource usage of unpredictable grid tasks in check [19].

2.3 Components of a Grid System

This section provides a brief overview of grid architecture. The application layer is discussed first, and portals as a method of human interface are explained. A method of classifying them is then presented. The application layer section concludes with an explanation of workflows in grid application development. A section on the middleware layers follows, wherein the services residing on the Collective, Resource, and Connectivity sub-layers are detailed. Finally, low-level details of grid systems are considered in a section on the Fabric layer.

2.3.1 Application Layer

The application layer consists of the applications which are run on a grid system. These include commercial, scientific and engineering processes. The application layer makes use of an abstraction of the lower grid computing layers to provide computing services to the programs running on it. A number of popular applications have been ported from traditional computing environments to provide grid support, including Mathematica, DB2, the Websphere Application Server, the Sun Grid Engine Enterprise Edition, and a number of other critical products from key software vendors [1][14][23]. Most current grid applications are developed privately and used for specific research applications, as discussed in Section 2.1.2.

2.3.1.1 Portals

The use of portals is a common method used to provide application layer access to users of grid systems through web based interfaces. Some popular grid portals are:

- Pegasus [10]
- GridFlow Portal [16]

- P-GRADE Portal [17]

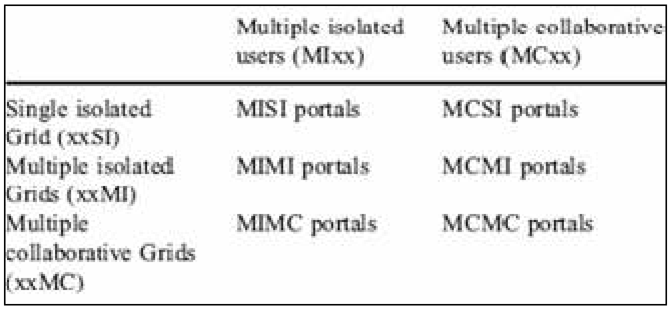
Each of these portals provides users access to a number of services, including workflow management, certificate management, job submission and visualization. Workflow management allows the grid user to specify the order and parallelism of an application. The Pegasus portal requires users to develop their workflows outside of the portal and then upload them, while Gridflow and P-Grade both provide integrated environments for workflow development. Each of the three portals allow for the submission and monitoring of workflows, allowing the user to view the status of submitted jobs whether they are queued, running, completed, etc. One may also view file output and logs. Additionally the three portals allow the users to download their certificates to the portal, thus granting them access to resources to which they are entitled. Through these functions users are able to run, authenticate and visualize the results of their application on the grid.

2.3.1.1.1 Portlets

P-GRADE is an extension of the open source portlet-based portal model Gridsphere, giving users of P-GRADE the ability to extend its functionality with portlets [17]. The intention of portlets is to provide "pluggable user interface components that provide a presentation layer to Information Systems," according to Java Specification Request (JSR) 168, the initial specification of portlets by Sun Microsystems [1]. Portlets allow developers to create JSP pages, which integrate tightly with Java code that extends the Gridsphere portlet model. This allows developers to handle web based events in Java using a traditional Java event-based model [1]. [1] Within P-GRADE a number of portlets have been created, including portlets to visualize grid status and to allow users to run parameter studies on the grid [13],[14].

2.3.1.1.2 Types of Grid Portal

Workflow oriented grid portals are defined by two values. This classification system is represented by the chart in Figure 1. The first value is the number of users who can access the same application and modify it. The second value is the number of grids that a portal is connected to and that the portal can execute jobs on. The first level of the chart shows the cases with users able to access only one grid. In the second level, users are able to see and access resources on multiple grids. All workflow parts are still only executed on one grid. The third level allows for workflow segments to be divided and run on multiple resources [17].



	Multiple isolated users (MIxx)	Multiple collaborative users (MCxx)
Single isolated Grid (xxSI)	MISI portals	MCSI portals
Multiple isolated Grids (xxMI)	MIMI portals	MCMI portals
Multiple collaborative Grids (xxMC)	MIMC portals	MCMC portals

Figure 1 - Workflow Portal Classification [17]

The current state of grid portals does not allow collaborative development of applications and programs on the grid. This is believed to be an important next step in grid development. Future users of the grid in industry will likely wish to have multiple users contributing to a single application project [17].

2.3.1.2 Workflows

Workflows are an abstraction that allows users to develop applications able to run on grids and take advantage of the parallel processing power of such systems. Workflows are a series of nodes of a flow diagram that designate the binaries to run for each node. Input and output data are also abstracted through the concept of ports. When creating workflows the user needs only to know how the parallelism of their application should be structured, as the workflow manager provides for the abstraction of most grid details.

When developing grid applications the type of parallelism is an important consideration. A classification system developed by Flynn, based upon the number of programs run and the data input, is useful for this. The first application type in this system is Single Program Single Data (SPSD), is an application which takes in a single data set and produces a single output set. SPSD applications lend themselves easily to grid applications, as their workflow consists of only a single node. These applications are only effective when large numbers of data files exist, as a grid will effectively increase their throughput [2].

The next classification is Single Program Multiple Data. These applications split large data sets into smaller sets, each of which can be processed by the same program and then compiled together after processing. This workflow of such a system is fairly easy to handle, as the user only needs to consider the parallel processing of a single node. SPMD applications are the most popular application group currently being used on grid systems today [2]. An example of such an application is SETI@home, which splits large data sets collected by radio telescopes into smaller data sets. These are then processed through a volunteer-based grid [14]. Many bioinformatic applications also fall into this application category [2].

The third category is Multiple Program Multiple Data. These are applications consisting of multiple data sets that can be processed concurrently by multiple programs, which can assemble the output after completion. The workflow of MPMD applications are more complex than the earlier two groups, as the parallel processing of multiple nodes must now be considered. Successful porting of this application group can again cause performance gains, as well as having the added benefit of matching nodes to optimal resources[2].

The final classification group is Multiple Program Single Data, this is a case in which a single data set must be processed by multiple applications. These applications again can be developed for grid applications, but this application type is rarer than the others[2].

2.3.1.3 P-GRADE

The P-GRADE portal is designed to facilitate the construction of grid workflows and their execution on the grid. In this capacity, it can emulate any of the above configurations. The portal provides a graphical environment for the construction of workflows. This portal application also takes care of many of the necessary background operations required to run applications on the grid. The P-GRADE portal keeps track of the state of the grid environment and handles certificate management. In addition, the portal monitors the progress of running workflows [17].

The P-Grade portal provides users with the ability to develop and run grid applications. It does this by providing the user with the following functions:

- Defining a grid environment
- Creating and modifying workflows
- Managing Grid Certificates
- Controlling the execution of workflows

- Monitoring and visualizing the execution of workflows [17]

The first function, defining a grid environment, is set up by the Portal Administrator, who must define the VOs which are accessible to the portal. The administrator will also associate computational resources to each VO. The individual users of the portal do not have the ability to change the VOs that are accessible to the portal, but are allowed to add and remove individual computational resources to their resource list [17].

Creating and modifying workflows on the P-GRADE portal is handled using a Java Web-Start application that can be opened from the portal. This application allows the user to define the nodes of the workflow as well as the connections between their input and output files. Upon completion of grid workflow development, the application can then be uploaded back on to the portal along with any necessary binaries and input files [17].

Managing Grid Certificates is essential to the use of the P-Grade portal. Certificates grant users access to the grid resources which they have permission to execute their code on. Users can download their certificates from a proxy server on the portal. Once the certificate is on the portal, the user must set it for one or multiple VOs. After the completion of this step the user will have the ability to execute their code.

The P-GRADE portal enables users to execute completed workflows which they have developed or uploaded on to the portal. The portal will also provide details to the user about the status of each node of their workflow, indicating, if it has been submitted, is scheduled, running, completed, in error or requires rescuing. The portal will also display the contents of standard output, as well as any error log that is produced by their processes [17].

2.3.2 Middleware Layers

Precisely what components of a grid system fall into the category of middleware tend to vary by architectural and design needs. A grid devoted to computation, for example, would include a different set of services than a grid devoted to data storage and accessibility. There are, of course, a core set of components that will be present in nearly any grid system implementation. There are generally the services and protocols necessary to map application-layer jobs to fabric-layer resources while remaining free of application specificity or physical architecture requirements.

Foster et. al. sub-divide middleware into the collective, resource and connectivity layers (See Figure 2). As the highest level of the three, the Collective layer is tasked with collecting and managing "interactions across collections of resources."[7] Two large categories of service are generally found here: Information Management and Data Management. The former is responsible for scheduling, monitoring of resources, diagnostics, and resource discovery, among others. Services handled by the latter include movement of data, managing of replicated files, and quality-of-service for large file transfers.

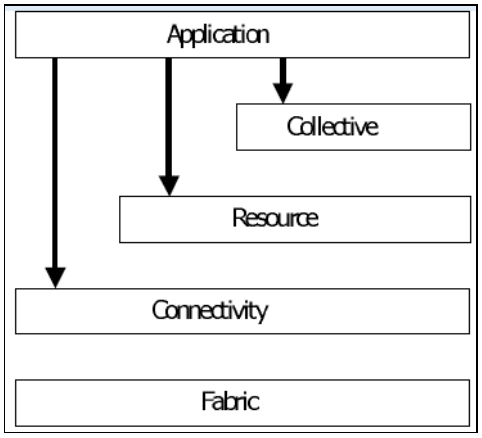


Figure 2 - Layered Grid Architecture [7]

Note that this layer is devoted largely to management. Actually moving files or allocating hosts for a job must be done by the Resource Layer, which implements “protocols (and APIs and SDKs) for the secure negotiation, initiation, monitoring, control, accounting, and payment of sharing operations on individual resources”[7]. While the Collective layer managed interactions between resources, this layer provides the software tools to actually carry out those interactions.

Information protocols, for example, are “used to obtain information about the structure and state of a resource,” including such considerations as “its configuration, current load, and usage policy” [7]. Services that fall into this category are generally concerned with discovery or querying of a resource. Purposes may be related to scheduling, diagnostics, job monitoring, or resource allocation. Note that collective-layer utilities often use these in combination with resource-layer management protocols to perform their overall task. Unlike information protocols, these are used to “negotiate access to a shared resource, specifying, for example, resource requirements… to be performed, such as process creation, or data access” [7].

So far we have avoided how, precisely, resource-application or inter-resource communication occurs. This is handled by the Connectivity layer, which “defines core communication and authentication protocols required for Grid-specific network transactions” [7]. Since development of the Internet has led to a variety of strong and well-tested protocols for data exchange, these are often employed in grid environments on the connectivity layer. Other options do exist, of course, and may be more suitable for certain grid applications.

Authentication on grid systems normally employs a Public-Key Infrastructure to manage user identity and privileges. The specific implementation will depend upon the "type of grid topology and the data the security will be protecting"[14]. Military networks have a much higher level of necessary security than, for example, a university research site. Similarly, the need for authentication vs. authorization must be considered. If users share a largely equivalent set of privileges, verifying that a user is in fact whom they claim to be may be more important than ensuring they have the right to use a given resource.

This leads to another important security trade-off, unique to PKIs. All certificates must be validated and issued by a central Certificate Authority (CA), as without a trusted third party certificates may be forged. Although several possibilities exist, two options immediately present themselves. Either one could use an existing CA, or create one specific to one's organization or grid. The former has the advantage of being established, with known procedures and levels of trust. However, commercial CAs may not be an affordable option. Government and institution-based CAs require, at minimum, membership in their organization. The latter option brings with it several questions:

- Where will my CA be deployed and how will I manage it?
- Do I have the necessary processes in place to administer my own CA?
- What are the responsibilities for managing my own CA? [14]

A poorly managed CA has the potential to compromise one's entire security infrastructure. Of course, a well-managed certificate authority can provide a powerful authentication mechanism.

2.3.3 Fabric Layer

The grid fabric layer consists of the actual resources whose jobs and communication are managed by higher layers. Note that these may be heterogeneous. It does not matter whether a given entity is a workstation, cluster, or supercomputer as long as all present the same interface to the connectivity layer. Of course, not all resources need be computational in nature. Storage resources are also common, and some grids may also have sensors capable of generating data to be processed (e.g. radio telescope observatories).

Note that there exists "a tight and subtle interdependence between the functions implemented at the Fabric level, on the one hand, and the sharing operations supported [by higher layers], on the other."[7] This is not to say that the connectivity layer should ever assume that resources are using a particular architectural configuration. Rather, only functions which are implemented can be used by higher layers. The mode of their implementation is not important; their existence is the only requirement. A mechanism to specify whether a stored file is striped across multiple disks, for example, can only be called by higher layers if the resource in question supports such operations. Foster et. al. observe that at minimum, computational resources must provide a means of starting, monitoring, and controlling process execution. Storage resources must similarly at least provide operations for writing and reading files [7].

2.4 Grid Architecture and Technology

While specific hardware and application architectures cannot be sufficiently generalized to grid systems as a whole, several middleware technologies are widely implemented and worthy of further study. The Globus Toolkit is a suite of software services and development tools devoted to all aspects of the collective, resource, and connectivity layers. In addition to the pre-made packages, libraries are included for purposes of independent development. Condor exists as a service running both above and below Globus. It is responsible for fabric creation and resource management in a number of grid implementations. Before delving into either of these, however, an overview of the parallel computing libraries that are used in both toolkits is presented.

2.4.1 Parallel Processing

Common to almost all forms of distributed computing is the concept of parallel processing. In brief, this is a computing approach that divides a job into several component subtasks. Each subtask is solved in parallel on its own processor, and the results are re-integrated into one solution. Consider, for example, the calculation of a factorial. This is at first glance a serial process; one number is multiplied by another until the product of all integers between one and n have been found. Indeed, this is how the calculation would proceed in a single-processor environment. In a multi-processor setting the process would be handled differently, however. One approach might assign each processor some portion of the problem. Multiplication then proceeds in parallel, and at the end each sub-problem can be recombined to arrive at the solution. The total time taken to reach this answer with multiple parallel processors should be proportionally faster than that of a single processor.

The potentially vast decrease in running time has made parallel processing critical in solving many computation-intensive problems. Among these are a variety of scientific research needs in fields ranging from climatology and biology to physics and economics. Furthermore, "such computing power is driving a new evolution in industries such as the biomedical field, financial modeling, oil exploration, motion picture animation, and many others.”[14] Admittedly, barriers exist in most problems which prevent perfect scalability and parallelization. Some tasks simply cannot be sub-divided, while others may rely on shared access to a database or storage element.

Currently two de facto standards exist for the implementation of parallel computing. Both of these at minimum define means by which processes may communicate with one another. The Parallel Virtual Machine (PVM) uses a virtual computing device to integrate heterogeneous resources, while the Message Passing Interface (MPI) uses message passing and static computing to optimize the performance of multiprocessing. In most cases one implementation will be more appropriate for a given problem than the other. In “cases where the problems do match but the solutions chosen by PVM and MPI are different…usually such differences can be traced to explicit differences in the goals of the two systems, their origins, or the relationship between their specifications and their implementations.”[12] Both PVM and MPI will now be considered in more detail, and their particular strengths and weaknesses noted.

2.4.1.1 PVM

The Parallel Virtual Machine is designed to provide both portability and interoperability in a heterogeneous distributed environment. The distinction between these two goals can be fine-grained, but is necessary in understanding the different directions taken by PVM and MPI. Portable applications that are “written for one architecture can be copied to a second architecture, compiled and executed without modification.”[11] Interoperable applications are portable applications whose “executables can also communicate with each other.” [11]

PVM provides both portability and interoperability through the use of a virtual machine: a set of heterogeneous hosts connected by a network that appears logically to the user as a single large parallel computer." [11] Since architectural differences are hidden from the running process, PVM distinguishes itself in heterogeneous environments that integrate a variety of computing architectures and platforms. An unfortunate outcome of this interoperability, however, is the inability of PVM to use hardware-specific implementations. If a programmer is building an application to run on clusters of identical machines, for example, PVM may not be the most efficient choice. In general, performance tends to be sacrificed for the ability to integrate a wide variety of differing systems into one logical environment.

As a virtual machine PVM also includes resource management features. This allows for a dynamic computation environment, in which "computing resources, or '\hosts,' can be added or deleted at will, either from a system '\console' or even from within the user's application." [11] Load balancing and task migration are therefore possible in a PVM environment. Furthermore, "applications can exhibit potentially changing computational needs over the course of their execution." [11] Consider, for example, a process that has an initial serial component, a parallel component, and final serial component. Resource management also allows for dynamic allocation of computing elements; the extra processors need not be assigned until the process' parallel stage.

Finally, the system supports fault tolerance through notification mechanisms. Individual "tasks can register with PVM to be '\notified' when the status of the virtual machine changes or when a task fails." [11] Each notification contains data relating to the event that triggered it. Furthermore, a “task can ‘\post’ a notify for any of the tasks from which it expects to receive a message.” [11] In this way failed processes may be restarted or allocated to a different machine. Load balancing and resource management can also take advantage of the notification system. Hosts joining or leaving the virtual machine can cause dynamic reallocation of jobs by notifying a resource broker of their presence. Lastly, consider the case in which a particular host has a sub-par record for job completion. This could be noted by recording the amount of task-fail notifications, and the resource could be avoided by future processes.

2.4.1.2 MPI

Unlike PVM, the Message-Passing Interface places performance before interoperability. While an “MPI application can run, as a whole, on any single architecture and is portable in that sense…nothing in the MPI standard describes cooperation across heterogeneous networks and architectures.” [11] Technically, this does not prevent any organization from developing such an implementation. To do so, however, would be in direct opposition to MPI’s performance orientation. By executing a single architecture, the system can take advantage of hardware-specific implementations and thereby achieve potentially large gains in terms of efficiency. This makes it ideal in homogenous environments such as multi-processor systems and some networked clusters.

Note, however, that there is no support for resource management in MPI. The “standard does not support any abstraction for computing resources,” and maintains a strictly static computing environment [11]. Once resources are assigned to a process (i.e. at execution-time) those resources are considered to be allocated until process termination. This remains true even if multiple-processors are assigned to a serial job, such as the example given above with one parallel component sandwiched between sequential components. MPI was "specifically designed to be static in nature to improve performance. There is clearly a trade-off in flexibility and efficiency for this extra margin of performance," in addition to the overhead of a custom resource management scheme [11].

As expected in a static environment, MPI lacks any real fault tolerance. If "a task or computing resource should fail, the entire MPI application must fail."[11] No native methods exist by which a process may be spawned after the initial execution phase, therefore no means to recover from failure are provided. The MPI-2 standard (a revision of the original MPI specification) does include a way to spawn processes and provides notify scheme comparable to PVM's, however [11].

2.4.2 Globus Toolkit

The evolution and proliferation of grid systems has created "a need for protocols (and interfaces and policies) that are not only open and general-purpose but also standard." [9] Indeed, "it is standards that allow us to establish resource-sharing arrangements dynamically with *any* interested party and thus to create something more than a plethora of balkanized, incompatible, non-interoperable distributed systems." [9] Without a common development standard individual grids may at best collaborate only after a great deal of work, and at worst risk becoming application-specific systems dependent on specialized or proprietary code. The Global Grid Forum is attempting to address this need for standards through the Open Grid Services Architecture (OGSA), which "modernizes and extends Globus Toolkit protocols to address emerging new requirements, while also embracing Web services." [9]

The Globus Toolkit provides a suite of software tools to allow rapid implementation and installation of grid middleware. Specific application services are avoided in favor of general infrastructural services, such as security, data management, execution management, and information management. Furthermore, run-time libraries for Java, Python, and C are provided to support the development of custom services. Each of these functions will now be considered in more detail.

2.4.2.1 Security

Globus uses the X.509 public-key infrastructure to "implement credential formats and protocols that address message protection, authentication, delegation, and authorization." [8] Each virtual organization has one certificate authority (CA), and each user has an authentication certificate signed and validated by that CA. Once any two entities are issued their certificates, protocols are implemented which allow them to "validate each other's credentials, to use those credentials to establish a secure channel…and to create and transport delegated credentials that allow a remote component to act on a user's behalf for a limited period of time."[8]

Authorization is handled through the authorization framework component. This provides a means to create secure interface modules for services, and for all access requests to pass through these interfaces. Various supporting tools, such as MyProxy, are supported in order to abstract the process of certificate management from end users [8].

2.4.2.2 Data Management

Information is moved, stored, and access in the Globus Toolkit through several key services. GridFTP "provides libraries and tools for…memory-to-memory and disk-to-disk data movement" in a grid environment [8]. The protocol is also interoperable with conventional FTP services, however. Hence, data stored on a grid resource could be accessed by computers not participating in the grid, or vice versa.

GridFTP transfers are managed by the Remote File Transfer (RFT) service. It is capable of reliably handling a large number of simultaneous network transfers; RFT “has been used, for example, to orchestrate the transfer of one million files between two astronomy archives.” [8]

A number of other data management services are also included in the Globus Toolkit. Data Access and Integration Tools (OGSA-DAI) allow processing of relational and XML data, and the Replica Location Service (RLS) maintains information on replicated files. A revision of RLS is currently in development, which combines its services with GridFTP to better manage file duplication [8].

2.4.2.3 Execution Management

Remote process control is provided through the Grid Resource and Management (GRAM) service. Specifically, it provides “a Web Services interface for initiating, monitoring, and managing the execution of arbitrary computations on remote computers.” [8] Precise resource requirements and data transfers can be specified, and the credentials to use on job execution may be included. In addition to this service, two other technologies are currently included in Globus Toolkit v4: the Workspace Management Service (WMS) and Grid TeleControl Protocol (GTCP). Both are in beta stages, and likely to undergo changes in future releases. WMS allows for “the creation of execution sandboxes, using Virtual Machines or Unix accounts.” [8] GTCP is used in “managing instrumentation; it has been used for earthquake engineering facilities and microscopes.” [8]

2.4.2.4 Information Services

The Globus Toolkit employs an XML-based monitoring and discovery service. Standardized procedures are provided for "associating XML-based resource properties with network entities." [8] Through querying, resource managers may monitor grid resources. Through subscription, new hosts may register with the necessary brokering and management systems. These procedures are built in each GT4 service. In addition, they are included in all provided containers (See Section 2.4.2.5). User services can thus "be configured to register with their container, and containers with other containers, thus enable the creation of hierarchical (and other) structures." [8]

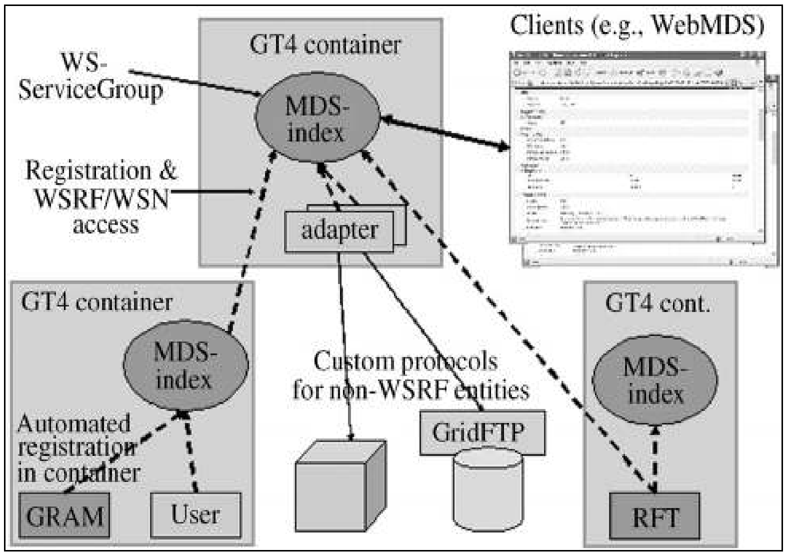


Figure 3 - GT4 Monitoring and Discovery [8]

2.4.2.5 Common Runtime Libraries

GT4 supports user-created services through Web Services container libraries. Individual containers are provided for Java, C, and Python. In general these provide "message handling and resource management, thus allowing the developer to focus their attention on application code." [8] As their name implies, containers can be thought of as a wrapper layer that shields user code from grid implementation details. Any software built on a container should run on any GT4 environment, assuming any dependencies on external libraries are met.

2.4.3 Condor

The purpose and application of Condor can at first seem confusing, as two systems share this moniker and serve very different roles. Each can be considered adjacent to one ‘face’ of the Globus layer, but still fall within category of middleware (See Figure 4). The lower component is more properly referred to as Condor High Throughput Computing, which serves in this regard as a “fabric management service (a grid ‘generator’) for one or more sites,” interfacing with the infrastructural layer [26]. The upper component, Condor-G, is a “submission and job management service,” which deals directly with the application layer [26]. The Globus Toolkit bridges the gap between both versions of Condor, providing the services discussed in Section 2.4.2. Each will now be considered in more detail.

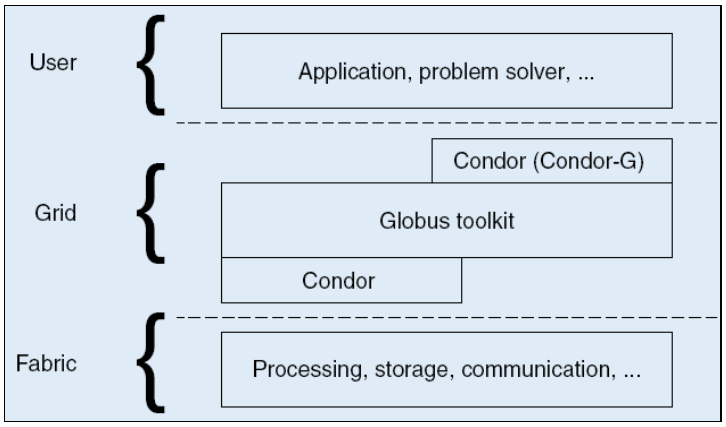

User
Application, problem solver, ...
Condor (Condor-G)
Grid
Globus toolkit
Condor
Fabric
Processing, storage, communication, ...

Figure 4 - Condor and Globus [26]

2.4.3.1 Condor High Throughput Computing

As a fabric management service, Condor HTC is tasked with making a grid from heterogeneous networks of computers. It is easiest to understand the structure of such grids (called Condor Pools) by going step by step through a normal job submission process. Figure 5 provides a graphical overview of the below procedure.

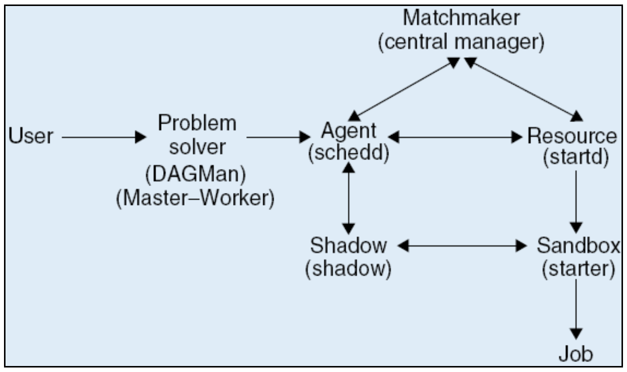


Figure 5 - Condor HTC Kernel [26]

First, a user creates the job and submits it to problem solver service. Here it is formatted and passed to the agent, which is "responsible for remembering jobs in persistent storage while finding resources willing to run them."[26] Both agents and computing resources report their existence to a matchmaker, whose function is to find optimal resources to fulfill agent requests. Once such a match is found the existence of each is reported to the other. Actually establishing the connection and verifying that resources are still available is left to the agent [26].

To run a job, both sides must spawn new processes. Agent-side, a shadow is responsible for providing job specifics. Resource-side, a sandbox is created in which the job will be computed [26]. The actual process is a good deal more complicated, and includes facilities for handling multiple matchmakers and geographical resource optimization. Such topics are covered in detail by Thain et. al. in *Condor and The Grid*.

2.4.3.2 Condor-G

Essentially a GRAM-based agent, Condor-G was created to fill the need for a "system that can remember what jobs have been submitted, where they are, and what they are doing. If jobs should fail, the system must analyze the failure and resubmit the job if necessary."[26] It may be run over any kind of batch system, and is not limited to working only with Condor HTC. This has the obvious advantage of hiding architectural differences and allowing for easier interaction between heterogeneous resources. Unfortunately, the resource allocation process and job execution processes are coupled in Condor-G. This "forces the agent to either oversubscribe itself by submitting jobs to multiple queues at once or undersubscribe itself by submitting jobs to potentially long queues." [26] Lastly, Condor-G suffers from the same tradeoff between performance and interoperability that PVM does. Features specific to certain batch systems cannot be utilized, therefore potentially sacrificing gains in efficiency implemented in a particular piece of batch software.

2.5 EMMIL

The Electronic Marketplace Model Integrated with Logistics (EMMIL) is a business to business marketplace engine developed by Dr. Livia Kacsukné Bruckner. In this system, a buyer submits a buy order containing a list of goods and their desired quantities. The model then selects an optimal combinations of sellers and logistics providers to satisfy this order the least possible cost. EMMIL makes a basic assumption that all goods are not digital, i.e. products are tangible and must be physically transported. This model can function regardless of the orientation of the marketplace, allowing for buyer, seller, and exchange oriented services. The EMMIL engine is designed to sit between front end processing programs for the buyer, seller, and logistical provider. The model requires that each of these entities is able to supply the engine with its required data very quickly [3].

Combinatorial auctions are used to evaluate potential combinations of product sellers and logistics providers. Specifically, the model generates every possible combination of sellers and logistics providers to be considered, then computes the most optimal of these combinations by minimizing the price of purchasing some number of goods. The number of datasets which are generated can be determined using combination functions; these are explained in Appendix III. In general, however, this can create a potentially tremendous number of datasets, and elevates EMMIL to the status of a grand challenge problem. These require significant amounts of computing power to solve. The independent nature of each combination (i.e. it may be evaluated in isolation from the others) indicates that parallel computing could be used to evaluate many of these simultaneously, however.

2.5.1 EMMIL and Grid Systems

An EMMIL system can require large amounts of computing power as the number of variables in the model increases. This computational requirement may be impossible for one computer to supply in the short amount of time required for an interactive bidding marketplace. One possible solution that has arisen is a grid based implementation. As already described in this paper, grid systems attempt to make available large amounts of distributed computing power to users who might otherwise lack the resources to carry out such massive computations on their own.

The way in which the EMMIL model functions by running computations on different combinations of input data allows it to be broken up into a number of parallel processes that can run independently of each other. Each run through the process of solving for the best sub-combination of sellers for items can be run on a separate computing element on a grid. If the solving process is fairly rapid, several can be run by a single computing element and minimize the overhead of allocating grid resources for the task. This parallel use of multiple computing elements would allow the computations to finish faster then they would on a single computer.

Note that a grid solution will not allow the EMMIL model to function as a real-time bidding system due to time delays between allocation of grid resources, computation, and the returned result. This solution can still allow many bid sessions over a time period, where bidders will enter values at a predetermined time and the job will be submitted to the grid. Upon returning from the grid results can be displayed to selected involved parties, who can then adjust bids based on returned values and submit them for another period of grid computation.

3 Methodology

Our project was divided into several developmental phases, with the overall intent of transforming the basic EMMIL system designed by Dr. Livia Bruckner into a useful grid application. An improved version of the model is already in development by students and researchers at MTA-SZTAKI. However, testing of this prototype will indicate whether further research should be put into the conversion of EMMIL to a grid-based B2B e-marketplace. Our specific objectives are as follows:

- Investigate the feasibility of EMMIL as a grid application
- Create a portlet on the P-GRADE portal to allow intuitive and complete data entry
- Enhance the basic EMMIL model to optimize performance

In brief, the first objective inserted benchmarking code into the model, but was much more focused on performance evaluation than code development. In this phase the P-GRADE Portal provided an interface to the SEE-GRID[1], on whose resources we performed all of our grid-based testing. The second phase of our project centered on code development, using Java Server Pages to create a new portlet on MTA-SZTAKI's P-GRADE Portal. The final phase of our project entailed modification of existing C code running on a Linux platform. Each enhancement was developed independently of the other, and once operational both were merged into a new version of the EMMIL system.

3.1 EMMIL Implementation

3.1.1 EMMIL Algorithm

The EMMIL algorithm used in the implemented version employs combinatorial auctions to arrive at an optimal combination of sellers and logistics providers. It is as follows:

Minimize over S:

$$\sum_{j=1}^{S}\sum_{t=1}^{C}\left(t * F_j * y_{j,t} + \sum_{i=1}^{N}(V_j + P_{j,i})Q_{j,t,i}\right) \text{where C is the upper limit for } 1+\left\lceil \frac{1}{Z}\sum_{i=1}^{N} Q_i \right\rceil$$

Constraints:

- $y_{j,t}$ must be either 0 or 1
- $Q_{j,t,i}$ will be set to 0 if $y_{j,t}$ is 0

[1] A grid system maintained by the South Eastern European Grid-Enabled eInfrastructure Development Virtual Organization

The variables used are as shown in Figure 6. Several of these require further explanation. The number of total sellers, M, does not appear in the above equation. This quantity is not involved in individual combinations to be evaluated, but rather assists in determining how many will be created. The number of filtered sellers, U, is a quantity representing some number of sellers filtered from M. The algorithm governing this is currently very crude, ranking sellers by the total price of buying all items from that seller and ignoring transportation costs. U replaces M as the set of sellers to select from. Specifically, the number of total combinations is equal to $\binom{U}{S}$. The binary decision variable, y, is used in order to keep the problem linear.

Variable	Description	Origin
M	Number of total sellers	Input
U	Number of filtered sellers	Input
S	Number of sellers in a combination	Input
N	Number of products	Input
Q	Desired product quantity	Input
P	Unit price	Input
F	Fixed transportation cost	Input
V	Variable transportation cost	Input
Z	Container size	Input
C	Maximum containers needed per seller	Derived
Y	Binary decision variable	Derived, see text

Figure 6 - Variables Used

This becomes clearer when one walks through the algorithm. First, the total price for all products purchased at a seller is calculated. This considers both unit price, P, and a variable transportation cost, V. To this is added the fixed transportation cost, F, which is a per-container expense (e.g. a flat fee per lorry). The result is compared against all sellers in that combination, and the cheapest solution is returned.

Note, however, that the number of containers cannot be a continuous value; in a real world situation one cannot use 1.5 containers. Two containers must be employed, with one filled to only half capacity. In order to allow discrete integer variables while remaining linear, the binary decision variable, y, is used. This value is true if and only if the currently selected number of containers, t, is the exact number required to transport the given selection of products.[2] A constraint is introduced to guarantee that when y is false, Q is set to zero. In other words, the current iteration only contributes to the ongoing summation if the currently selected number of containers is correct; otherwise the calculation is zeroed out.

3.1.2 Implemented Model

The initial model implemented by our group and colleagues at MTA-SZTAKI is split into three separate components, to be run in order. The Generator reads input data, creates combinations of sellers based on that input data, and outputs these to a file. The Core then reads this file, generates boundary constraints, and submits both these and the formatted generator output to LP_Solve (a linear problem solver, see Section 3.1.2.2.1). This must be performed for each combination output by the generator, which often numbers in the thousands. A file containing LP_Solve's results is written for each run. Once all combinations have been processed by the solver, the Collector evaluates the resulting output, selects the best combination based on this output, and displays the results in a human-readable format.

3.1.2.1 Generator

The original generator does not include any pre-processing functions. Upon execution it will open the data input file and initialize the basic variables (see Figure 6), as well as pricing data for products and shipping costs, with the values specified therein. Note that this file is not easily human-readable; it is merely an ordered list of numbers

[2] The maximum number of containers per seller, C, is based on maximum container size and total product quantity. See the above equation for details.

separated by spaces. Data structures are then generated to hold products' unit prices, which are assigned using a random number generator. A similar technique is used to create the fixed and variable transportation costs for each seller.

Next, the list of sellers is sorted in increasing order by the total price that would result from buying all products in a given buy order from one seller. The first U (the total number of filtered sellers, from which combinations will be drawn) sellers are then used to generate all possible combinations of sellers. Each combination has the number of elements specified in S, an input variable containing the number of sellers per combination. These combinations are stored in a double-dimensional array indexed by combination number and seller index. Lastly, all basic variables, generated data structures, and a number indicating how many combinations to process are written to output files.

3.1.2.2 Core

Upon execution, the system core processes one input file. This is appropriate for a grid environment, as the core is the parallel parameter study component and is run on multiple resources simultaneously. P-GRADE's workflow management system provides each instance of the core with one input file, and the results are copied to an output directory as they are generated.

After processing input the core proceeds to generate a series of constraints that define the problem space. These are as follows:

- All X-variables (which represent combinations of product quantity, seller, and container) must be positive, i.e. total prices must be greater than or equal to zero

- Total product quantities must be equal to those specified in the buy order
- Total product quantities per container must fit within the lower and upper bounds of that container's capacity
- All Y-variables must be binary

The constraints and input data are then submitted to LP_Solve, an open-source Mixed Integer Linear Programming (MILP) solver available at http://www.lpsolve.sourceforge.net.

3.1.2.2.1 LP_Solve

LP_Solve is not an application, but rather a mixed-integer linear problem solving library available from Sourceforge. The specific algorithms and mathematical techniques that this module employs are outside the scope of this paper, but are published online at the library's development page [20]. One important to feature to note, however is LP_Solve's treatment of integer variable constraints.

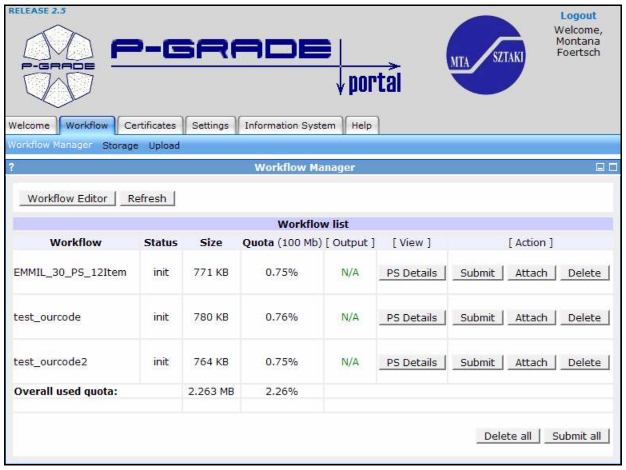
LP_Solve uses branch-and-bound to handle non-continuous (e.g. integer) variables. The problem is initially solved without any such constraints; i.e. all variables are treated as continuous. This produces a relaxed solution. Next, the system checks which variables must, in fact, be integers. For each such variable the model branches into two: "one with a minimum restriction on this variable that has the ceiling integer value and a second one with a maximum restriction on this variable that has the floor integer value."[20] Each model is again solved, and the one with a value closest to optimal (i.e. smallest if minimizing, largest if maximizing) is retained. This process repeats until all integer variables have been found and replaced with either the ceiling or floor values of the initial continuous value. Due to the nature of such branch-and-bound algorithms, models with integer variables "are harder to solve and solution time can increment exponentially." [20] Since EMMIL must make use of a potentially large number of integer variables, finding an optimal solution for certain sets of input data may take an infeasible amount of time on just one processor.

3.1.2.3 Collector

The collector reads a directory of output files generated by the core. Each file is opened and parsed in order to present a summary of that run listing its minimum price, how many of each item is to be bought from each seller, and how many containers will be needed per seller. Finally, after presenting each run the collector prints the overall minimum price, and hence the best combination of sellers and items to meet the buy order.

3.1.3 Gridification

In order to send EMMIL files to the grid and have results returned, it was necessary to set up a grid workflow in P-GRADE Portal. A workflow is a set of nodes and links which represents the path of a computing process. The workflow manager page displayed in Figure 7 shows all created workflows along with their current status. The workflow editor tab opens an application to construct a workflow.

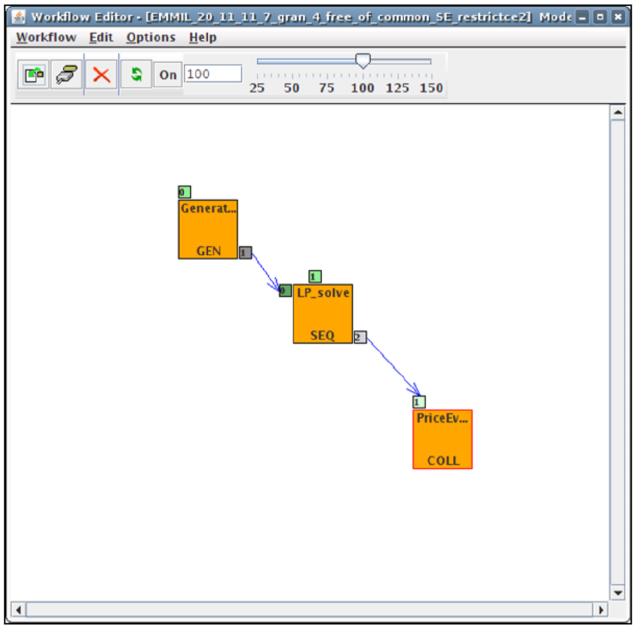


Workflow list

Workflow	Status	Size	Quota (100 Mb)	[Output]	[View]	[Action]		
EMMIL_30_PS_12Item	init	771 KB	0.75%	N/A	PS Details	Submit	Attach	Delete
test_ourcode	init	780 KB	0.76%	N/A	PS Details	Submit	Attach	Delete
test_ourcode2	init	764 KB	0.75%	N/A	PS Details	Submit	Attach	Delete
Overall used quota:		2.263 MB	2.26%					

Figure 7 - Workflow Manager

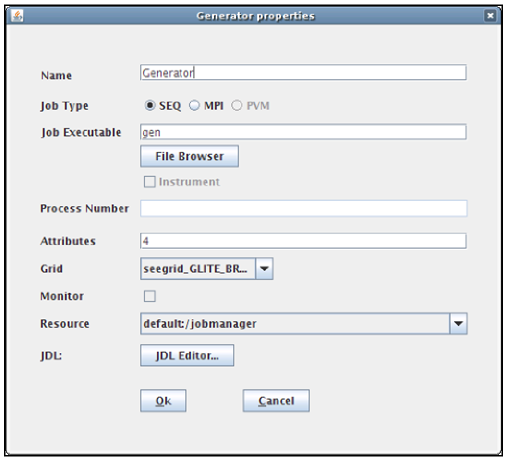
The workflow editor display in Figure 8 shows our created workflow. The workflow has been changed to a parameter study workflow, and the editor designated a generator node, a sequential node, and a collector node. The node marked GEN holds our generator file. The input port of this node is represented by a small box labeled 0. This port is the generator's input file. A second port is defined for output and is labeled 1. This output port represents all of the data files that will be created by the generator and passed on to the core. As this output port belongs to the GEN node of a parameter study workflow, the workflow editor understands that this process may produce multiple files and that each file should be passed on to a separate core process.


Workflow Editor - [EMMIL_20_11_11_7_gran_4_free_of_common_SE_restrictce2] Mode
Workflow Edit Options Help
On 100
25 50 75 100 125 150
Generat...
GEN
LP_solve
SEQ
PriceEv...
COLL
Figure 8 - Workflow Editor

The second node marked SEQ (sequential, i.e. without message-passing or other parallel communication capabilities) represents the parameter study component of our workflow and contains the core process. A copy of this process will automatically run on the grid for every data set created by the generator. The input port labeled 0 on this node and connected by a link to the output port on the generator takes in the data set file. A second input port is shown in the workflow editor on the core process node. This input port provides the core with access to the linear problem solver library. The output port of this node sends a computation result file to the final node in the workflow.

This final component, which is labeled as COLL, represents our collector. This node contains a single input port. As this is a parameter study workflow, the editor understands that the collector process will take in a variable number of files, one for each core process that executed.

Each node in the workflow can be opened to view specific properties. Displayed in Figure 9 is the generator properties tab with different attribute fields of this node which can be modified. This properties tab also allows access to the job description language, visible in Figure 9 as the button labeled JDL Editor.

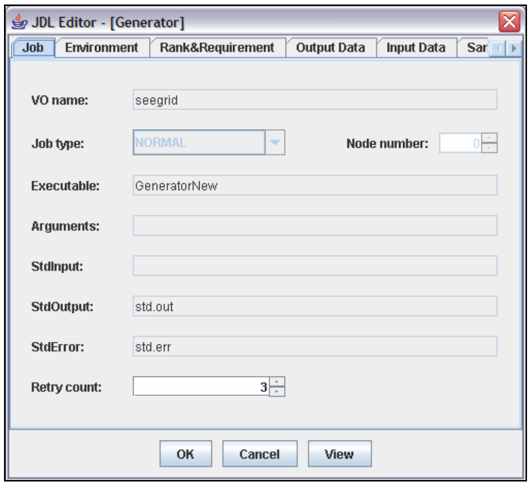

Generator properties

Name	Generator
Job Type	[✓ SEQ] MPI PVM
Job Executable	gen
Process Number	
Attributes	4
Grid	seegrid_GLITE_BR...
Monitor	☐
Resource	default:/jobmanager
JDL:	JDL Editor...

Figure 9 - Generator Properties

The Job Description Language Editor displayed in Figure 10 allows access to further customization of a node. Running environment specifics can be modified in this editor. Specific grid computing resources can be requested, or prevented from being allocated for this task. The definition of a set storage element can also be defined, which forces the workflow node to run on a computing element associated with that storage element.

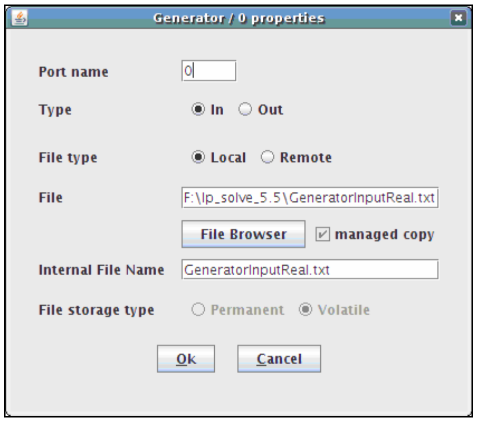
Displayed in Figure 11 is the Generator port 0 properties window. A port properties window allows changes to be made to the properties of files passed through the port. The type of the port defines whether this port will pass a file into a workflow node, or pass out a file produced by that node. The file type defines


JDL Editor - [Generator]

VO name:	seegrid
Job type:	NORMAL
Node number:	0
Executable:	GeneratorNew
Arguments:	
StdInput:	
StdOutput:	std.out
StdError:	std.err
Retry count:	3

Figure 10 - JDL Editor

whether or not the file is stored locally or on a grid storage element. The internal file name is the name that the executable in the workflow node associated with this port associates to with this file. Finally the file storage type tab defines whether this file should be permanently stored, or should be erased after it has been used. Since a parameter study can potentially produce a very large number of files, it is necessary to define files produced by parameter study processes as volatile.


Generator / 0 properties

Port name: 0

Type: [✓ In] Out

File type: [✓ Local] Remote

File: F:\lp_solve_5.5\GeneratorInputReal.txt

File Browser [✓ managed copy]

Internal File Name: GeneratorInputReal.txt

File storage type: Permanent [✓ Volatile]

Figure 11 - Generator Port Properties

3.2 EMMIL Grid Application Feasibility Testing

If EMMIL is to be used as a grid application, its performance in a distributed grid environment must be noticeably better than on a single machine. We created two sets of tests to determine if such was, in fact, the case. The first concerns single-processor execution measurements in a well-defined environment, aimed at discovering bottlenecks in dataset evaluation. The second set is a series of granularity and performance tests on SEE-GRID. We were unable to determine a feasible means by which computing element environments, time spent in broker queues, and time spent in local queues could be measured independently of execution time. Libraries for instrumenting code in such a way do exist, but are outside the scope of a seven-week project.

3.2.1 Single-Processor Testing

Before conducting any tests, we recorded the operating environment of the system that EMMIL runs would be executed on. No other significant processes were allowed to run during our tests, in order to ensure that random resource fluctuations would not confound any results. Our experiments were aimed at determining how the model responded to different sets of input data, and identifying crucial factors in computation time for creation of a rough granularity heuristic. Specifically, the effects of varying S, selected sellers; U, filtered sellers; Z, container size; N, number of different products; and Q, the desired product quantities, were investigated. For testing purposes all desired product quantities were set equal to one another.

Initially, our group intended to run each test a large number of times. The mean of all results for a given test would then be found, and random variance would be minimized. The amount of time that such experiments actually ran for, however, made this infeasible in practice given our limited scope of time. (See Section 4.1.1 for specific numbers and analysis.)

3.2.2 Grid-Based Testing

After establishing single-processor baseline values, our group measured the performance of EMMIL on SEE-GRID at several levels of granularity. These results were used to evaluate the efficacy of granularity controls, and to establish average performance measures for a SEE-GRID application. Unfortunately, there is no simple way to measure the running time of a partially parallel process such as a parameter study. Since each process is executed on a potentially different resource, and each process will have varying times of completion, one cannot simply record the time taken by each one and divide by the number of jobs spawned at once. Our solution was to design a testing framework such that generator and collector completion times are recorded in addition to the running times of individual jobs (note that this refers to one process, not one set of input data) in the parallel component.

One significant problem exists with this method, however: resource broker queues and local processor queues are conflated the perspective of our timing measurements.

Note also that any error requiring human intervention could significantly affect any recorded values.

Despite these flaws in the testing framework, our group was unable to employ more advanced techniques. P-GRADE Portal does support, through the grm library, a method of monitoring many aspects of a parallel process. All of the following conditions must first be met, however:

1. The source code of the respective processes has been extended by special instructions at proper places to send monitoring messages.
2. There is a special infrastructure (the Mercury_monitoring service) deployed in the remote resource where the submitted job runs to listen for and to gather these monitoring messages.
3. The user has enabled the monitoring by setting the Monitor flag [22]

The specialized knowledge of instrumentalization techniques and cooperation of remote resources was judged to not be a feasible solution in the three weeks of time allotted to our group for testing. All grid-based tests must therefore be considered as highly dependent upon grid loads at any given time.

One other factor had to be considered before jobs were submitted to the grid. Unless a storage element (SE) is defined for a given workflow, output files are not guaranteed to be written on storage elements accessible to all computational elements on the grid. The disadvantage of using a set SE, however, is that all jobs will be run on a nearby cluster. This can prove to be problematic, as a limit exists for how many threads per user may run on a given computing element. A concurrency limit of five processes is therefore normally imposed on any parallel workflow. This can be manually changed in the P-GRADE portal. However, if the thread limit is exceeded no other jobs associated with a given certificate will be run on that element until currently running threads terminate. This can interfere with experiments if granularity is set to allow more jobs than there are processors available, or if multiple workflows from the same user are running on a particular computing element.

With the above testing framework our group evaluated EMMIL and SEE-GRID performance at several levels of granularity. In this context, the term refers to the amount of independent datasets allotted to a single process, i.e. a measure of how the quantity of datasets maps to the number of processes spawned on remote computing elements. Fine granularity approaches a one-to-one mapping, while coarse granularity allocates a high number of datasets to one process. Through these tests we hoped to find a rough approximation of the relationship between the number of jobs produced, time per job, the number of computing resources allocated, the effects of grid conditions, and the total completion time.

The importance of this relationship in a grid environment cannot be understated. If too few resources are allocated to the EMMIL model due to excessively coarse granularities, parallelization is not being fully utilized and performance may approach that of non-grid solutions. If too many resources are allocated due to excessively fine granularities, grid overhead such as queues and network load could substantially inhibit time efficiencies. Optimum granularity, then, is tied on the one hand to job quantity and execution time (investigated in single-processor tests) and on the other to grid load and environment.

Our group hoped to arrive at an empirical approximation of this relationship, and use it to justify a heuristic algorithm which would automatically adjust EMMIL granularity based on an initial scan of input data and some knowledge of grid conditions. We did not intend to implement such a procedure, but merely sketch a potential heuristic.

3.3 EMMIL Data Input Portlet

The EMMIL application model requires a number of parameters in order to perform its simulations. The model must know the number of items (N), the number of sellers (M), the number of suppliers to choose from (M), the combination size, the container size, the desired quantity of each product and information pertaining to each seller. The details required of the seller include: the unit price for each item, their capacity of each item, and their fixed and variable logistic costs. A typical run of this model may contain as many as thirty sellers and ten products. For a user to manually enter all the seller details, they would be required to specify the price and product capacities of 300 items. This is an unacceptable situation, as it would require a large amount of tedious work by the user before each run.

Instead we made the decision that the data should be generated randomly for the user, using a normal distribution[3]. As input, the user would need to enter only a mean value and a deviation value for the unit price and for each product, as well as the fixed and variable costs. These values could then be generated for each seller. For the product capacity the user enters an upper and lower bound percentage, to determine the product capacity of each seller. In this case the capacity is uniformly distributed throughout the specified range.

The process of generating random values originally occurred within the Generator, which took the constant parameters as input in addition to the means and standard deviation of seller-related parameters. This leaves the user with no control over the data that is processed in the model. An alternative was to change the input file of the

[3] A form of statistical distribution in which all values fall within some number of specified deviations of a specified mean

generator to allow the user to specify all of the seller data. To prevent the user from having to manually type the seller data into the input file, we created an input portlet. This portlet allowed the user to specify all the constant parameters, as well as the means and standard deviations of the seller related items. Using these values the portlet then generates the values for all the seller related data, and displays the results in an editable HTML form. This allows the user to create specific scenarios, in which they can specify all of the seller data, or just the values they wish to control. This increases the control the user has over the application, while leaving it flexible enough where the user does not need to manually choose every seller value.

3.4 EMMIL Enhancement

The basic version of EMMIL that our project built on lacked many of the advanced features described in Bruckner and Csekenyi [3]. Our group introduced two enhancements to this model, designed to better simulate actual situations and to minimize the number of linear problems that are generated. Product capacity filtering introduces limits to how many of each product a seller can actually provide. Hopeless job filtering discards combinations of sellers that cannot possibly be used to arrive at an optimal solution.

3.4.1 Product Capacity Filtering

In order to prevent sellers from placing bids that they cannot fill, a data structure was introduced to hold maximum quantities of each product for each seller. A double-dimensional array of width N and depth M was used to store and access these as needed. (See Figure 12) While potentially memory-intensive, this approach allowed rapid random-access modification and retrieval through memory pointers in C.

	Product		
Seller	1	2	3
0	9	16	31
1	11	19	11
2	12	20	19
3	20	16	43
4	16	17	8

Figure 12 - Example Product Capacities

The actual values of a product capacity table are tied to the desired quantities of each product. This is done for simulation purposes only; if employed in real-world situations this information would not be dynamically generated. In order to test the model's performance under varying conditions of product availability, mechanisms were included to specify lower and upper bounds in its input file.

During EMMIL's preprocessing phase an algorithm was implemented to discard any combinations of sellers that cannot collectively supply at least as many products as are specified in the buy order. However, the number of iterations within each category makes this computationally expensive:

```
For each dataset i:
   For each seller j:
      For each product t:
         IF SUM_j Capacities[Seller_indexes[i,j],t] <Quantity[t]
         THEN DISCARD DATASET i
```

In effect, this algorithm adds the total product capacity for each product individually over all selected sellers in a given combination. If any value is less than the desired quantity specified in the buy order, that combination is discarded before processor cycles are wasted trying to evaluate it. In addition to discarding combinations of sellers that cannot supply the desired quantity of goods, it was further necessary to ensure that no seller provided more goods than its specified maximum. Our enhanced model accomplished this through a constraint introduced into the linear solver. Specific implementation details are available in Section 4.

3.4.2 Hopeless Job Filtering

A preprocessing filter has been implemented to reduce the number of jobs sent to the EMMIL core. The filter removes hopeless jobs containing combinations of sellers that have no chance of being the ideal solution. To determine if a job is hopeless, we computed the lowest possible cost of the job; that is the sum of the lowest cost of each product, the lowest variable cost, and the lowest fixed cost from the selection of sellers. This lowest cost value of the selection is then compared with the lowest cost value of any individual seller. If the individual seller's cost is less, the job is classified as hopeless and is filtered out. Figure 13 to the right shows an example of a situation where the hopeless job filter would be effective. The column on the left indicates the size of the selection, and the column on the right displays the combinations with the filtered selections in red. In this scenario the seller's rank is ordered based on the cost to buy all items from that seller, so Seller 1 is the best and Seller 5 the worst. In the case where the selection size is one, all but the selection containing Seller 1 are filtered because no single seller can beat the price of

S=1	{1}*{2}****{3}{4}{5}***
S=2	{1,2}{1,3}{1,4}{1,5} {2,3}{2,4}***{2,5}{3,4} {3,5}{4,5}***
S=3	{1,2,3}{1,2,4}{1,2,5} {1,3,4}{1,3,5}{1,4,5} {2,3,4}{2,3,5}{2,4,5} ***{3,4,5}***
S=4	{1,2,3,4},{1,2,3,5} {2,3,4,5}
S=5	{1,2,3,4,5}

Figure 13- Hopeless Jobs

Seller 1. In the case where S =2, four sellers are filtered, all selections with Seller 1 remain, as well as the selections {2,3} and {2,4}, this would be the case when Seller 2 may offer some products cheaper than Seller 1, and Seller 3 and 4 offer other products cheaper than Seller 1, producing a lower total cost. When S=3 only one selection is filtered, as there is now a better chance that a selection will be better than that of the single best seller. The only filtered selection is the one containing the three worst sellers.

This filter is potentially computationally intensive, as it performs an increasing number of computations as the parameters for number of products, sellers, combinations or the size of seller selections increases. The increased computation time of the preprocess filter is justified by filtering out the more computationally intensive EMMIL core jobs, which require large amounts of processor time to calculate linear equations. The hopeless job filter first must iterate through each seller selection; within the selection each seller of the selection is then iterated through, and then the products are iterated through for each seller. Below the pseudocode of the prefilter algorithm is shown:

```
For jobindex=1 to U do
   'Leave out jobs that are hopeless
      'Calculate theoretical lowest limit of cost for this data
      'set
      Cost_lowest_limit :=0
      For j=1 to S
            For i=1 to N do
                  lowestProductCost= min(
                                    unitPrice[j,i]*Quantity[i],
                                    lowestProductCost)
            EndLoop
            Cost_lowest_limit+= lowestProductCost
            lowestVarCost= min(varCost[j,i]*contNeeded),
                              lowestVarCost)
            lowestFixedCost= min(fixedCost[j,i]*totalQty),
                                 lowestFixedCost)
      EndLoop
      costLowestLimt+=lowestVarCost+lowestFixedCost

      'Do not use this data set if the cheapest sellers
      'can offer lower cost then theoretical lowest limit
      'calculated here
      If Cost_lowest_limit> Sorted_Purchase_cost[0] Then
```

```
            DISCARD_DATASET_JOBINDEX
        Endif
    EndLoop
```

4 Implementation and Results

Our implemented objectives spanned two programming environments. EMMIL model enhancements were written in the C programming language and compiled on a UNIX platform using gcc version 3.2.3. The P-GRADE portlet was written using Java Server Pages, and implemented on version 2.5 of the portal. All raw data tables are available in the Appendices.

4.1 EMMIL Grid Application Feasibility Testing

4.1.1 Single-Processor Testing

All testing was done on n49.hpcc.sztaki.hu. This machine has the following relevant statistics:

- Intel Pentium 4 3.00GHz with a 1024kb cache
- 2Gb of RAM
- Running Red Hat Linux 3.2.3

A single process was allotted up to 50% of total processor usage per execution.

4.1.1.1 Selected Sellers

Our first series of tests investigated the effect of increasing the number of selected sellers (S) while holding all other values constant. As expected, the amount of time taken to process each job increased steadily, and indeed almost linearly (see Figure 15). The precise progression is impossible to determine given the dearth of data points available for statistical analysis, however. An important note is that as S approaches U/2, (U is the number of filtered sellers) the number of jobs generated will increase to a maximum of $\binom{U}{\frac{U}{2}}$. (See Appendix III) This increases the number of jobs as well as increasing the time taken to compute each job. Figure 14 provides an overall summary of the effects of S-values on total computation time. Again, a linear progression appears to relate various values of S. The lack of data points beyond S=6 makes this impossible to prove satisfactorily, however.

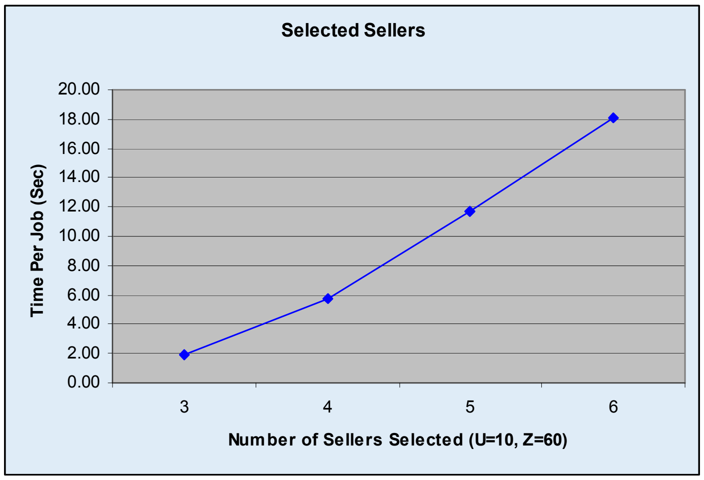


Figure 14 - Selected Sellers Job Time

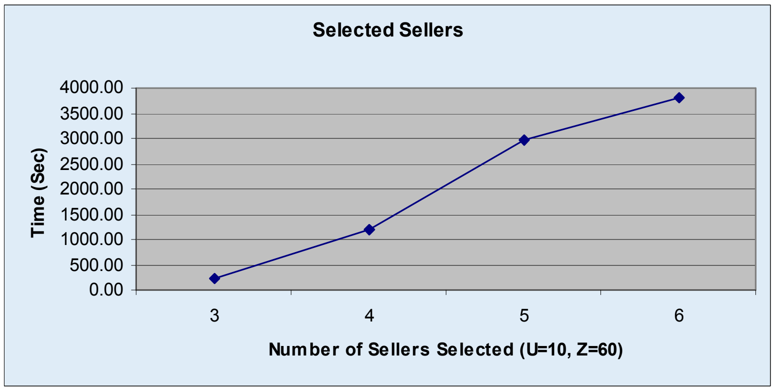


Figure 15 - Selected Sellers Run Time

4.1.1.2 Container Size

Our next series of experiments investigated the effects of container size on total computation time. Again, all variables were held constant unless otherwise stated. At high container sizes job time was quite rapid, and

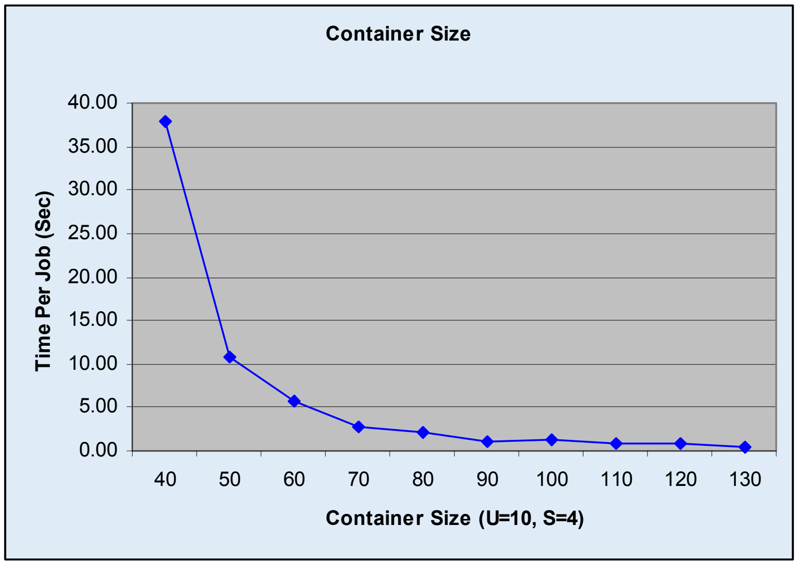


Figure 16 - Container Size

individual processes often finished in less than five seconds. At lower container sizes the time taken per job increases drastically, demonstrating exponential behavior. We attribute this to lp_solve's branch-and-bound handling of integer variables and the associated exponential increases in solve time caused by the introduction of more binary y variables[20]. At high container sizes all items bought from a given seller can fit into a single container, keeping the amount of integers in the problem to a minimum. At lower container sizes, multiple containers must be allocated to each seller in order to fit all goods purchased from that vendor. This could significantly increase the number of integer variables involved, especially if a large number of sellers are being purchased from.

4.1.1.3 Selected Sellers and Container Size

After investigating the role of selected seller quantities and container sizes individually, we ran a series of experiments to study their interaction at 3-6 sellers per combination and maximum container size of 20-130 units. The expected spike arising from multiple container allocation does indeed occur, at a container size of 50 units (see Figure 17).

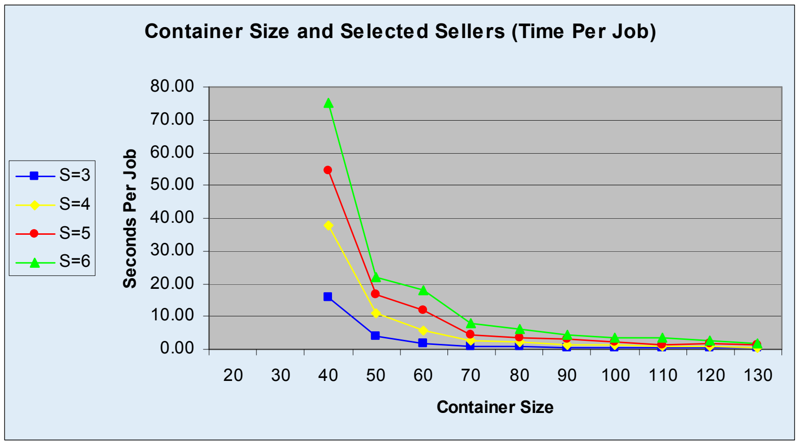


Figure 17 - Container Size and Selected Sellers (Job Time)

At high container sizes relative to product quantity, these data indicate that the selected number of sellers may be the determinant factor in processing time. As container size decreases, we theorize that a greater number of integer variables are introduced to the solver. This becomes a more important factor in computation time than the number of selected sellers. The two are also more intricately related, however. Increasing numbers of selected sellers presents a wider variety of sellers from which products may be purchased. As is shown in Section 3.1.1, each new seller will cause new y-variables to be created equal to the maximum number of containers per seller.

Furthermore, Figure 18, when compared with Figure 17, indicates that the number of jobs generated has, on its own, little effect on total execution time if all jobs are executed by the same processor. This is unsurprising, as in this case the number of jobs acts as a scalar value multiplied by the time taken per job. Granularity is, effectively, at the most coarse level possible.

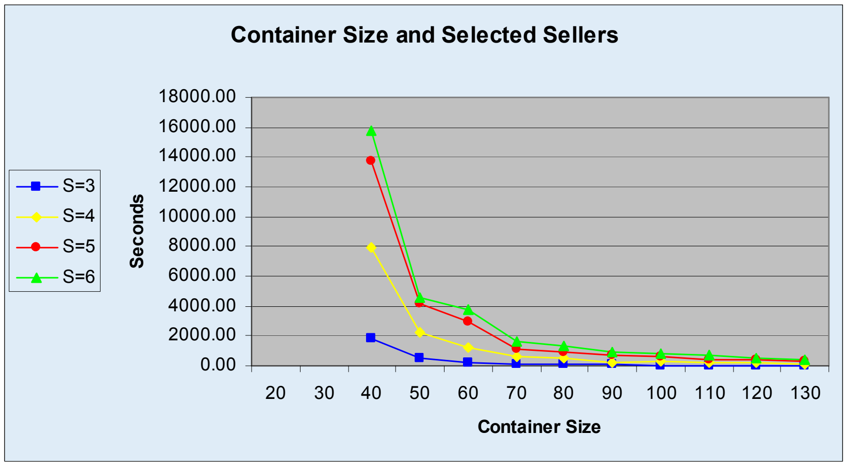


Figure 18 - Container Size and Selected Sellers (Total Time)

4.1.1.4 Product Number and Quantity

Our last series of experiments on the implemented EMMIL model investigated the effects of product number (N) and desired product quantities on execution time (Q). For testing purposes all desired quantities were set equal to one another. In Figure 19 above, Q=25 ensures the desired amount of each product will be less than the container size. Q=50 ensures it will equal the container size, and Q=75 will result in each product order exceeding the container size. The significant increase in processing time between

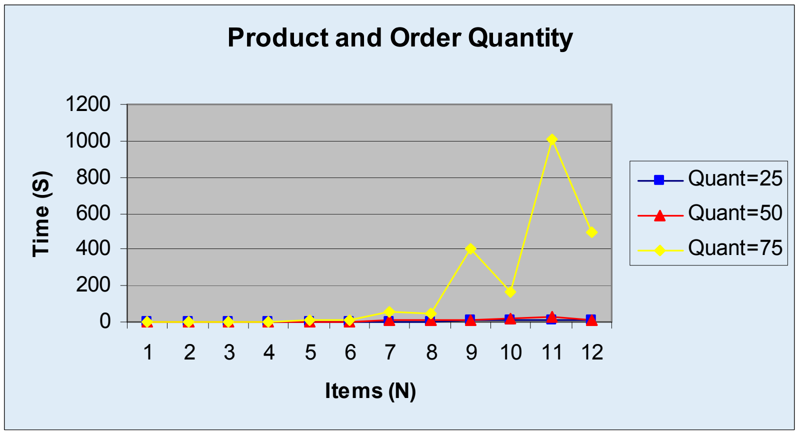


Figure 19 - Product and Order Quantity

Quantity = 50 and Quantity = 75 is the most interesting feature of these experiments. Note that if each product order exceeds the container size, the number of y-variables will increase steeply. Each seller must be allotted a number of containers equal to the ceiling of total product quantity divided by container size. These data support our theory that increased numbers of integer variables have a significant impact on computation time.

4.1.2 Grid-Based Testing

The results of our granularity tests are presented in Figure 21. Pre-solver idle time represents the time a job spent waiting for a resource to become available. Solve time represents the duration a process spent executing on a resource. Finally, post-solver idle time represents the amount of time taken waiting for other processes to finish and for the collector to run. The most desirable outcomes have the least total time from the first process beginning to the final process finishing. It is important to note that for these tests there were ten processes on a cluster reserved for our use. However these processors were not idling waiting for our tests to be submitted. If these resources were working on a previously assigned job they would first finish and then begin running our processes. The granularity of the different tests was chosen to force a workflow with more, the same number, and fewer processes then the number of computing resources available.

In the first graph in Figure 21 displays the time taken by the test running five core processes. This test was assumed to provide a suboptimal time increase over running all the tests on a single machine, as it did not make use of all of our available resources. Visible is the effect of different computation times for different data sets. While each process computed the results for sixty six different data sets, the overall computation time of each process varied widely.

The second graph represents the results of our ten process test each with thirty three different sets of data and produced the greatest increase in overall time required for computations. Two processes of this test were scheduled for a longer time then the rest, indicating that of our ten processors, only eight were immediately available. However, this scheduling delay did not greatly prolong the total time consumed by this test.

The final graph represents the results of our thirty process test each handling eleven different data sets. For this test ten processes began at approximately the same time, indicating that all of our resources were available from the start of the test. The purpose of this test was to determine weather or not it was better to further subdivide data sets. It was possible that a workflow with more processes each with fewer datasets would save time overall, by removing additional tasks form computationally intensive processes, and scheduling them on processors who had completed computationally light data set calculations.

Figure 20 is a graph displaying the total time required for the computation processes to finish. This means that each bar is the time from the first process scheduled beginning to run

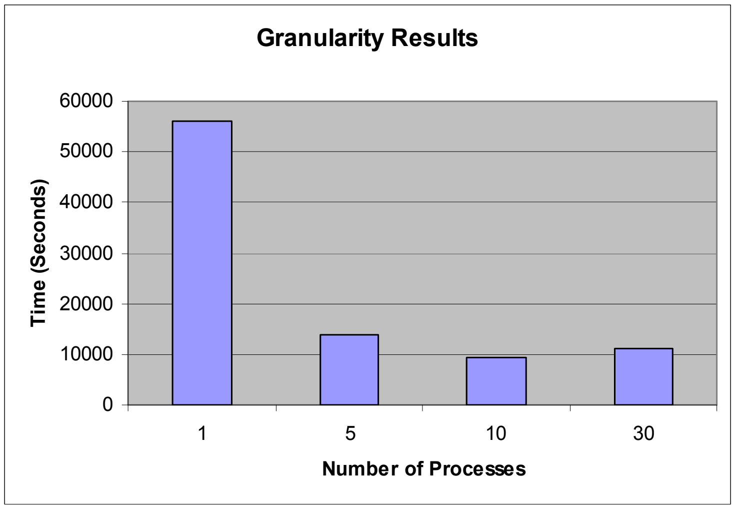
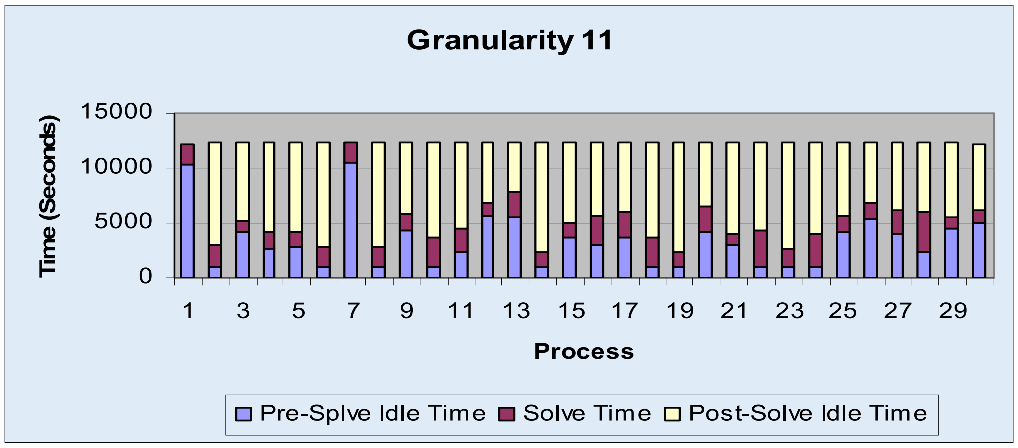


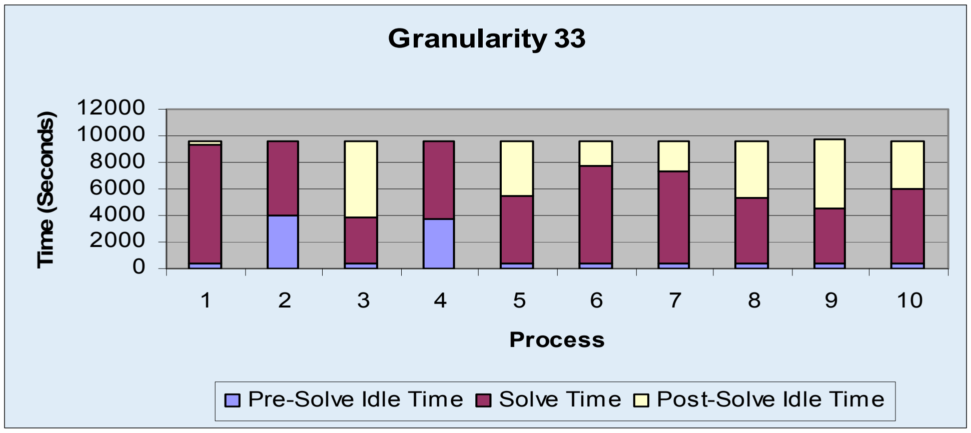
Figure 20 - Granularity Results

until the final process finished. It does not include initial process schedule times or the final scheduling delay before the collector process began. Our five run process test completed approximately four times faster then a single process running all computations on data sets. Our ten run test completed six and a quarter times faster then the single process run. While the thirty process test completed five times faster then a single process, and completed faster then our five process test, it failed to complete faster then the ten process test.

One observation from the thirty process test in Figure 21 is that two processes waited in the scheduled state after all other processes had completed which may have caused this test to take a noticeably longer time period to complete. Two processes were also delayed in our ten process test. These interferences with our test indicate the importance of not relying on resources being available beyond the start point of a workflow.

Secondly, the best completion time was achieved when the granularity used created a number of processes equal to our available resources. Despite the greatly differing computation times for each of these processes this test still performed better then our thirty run test. As the thirty run test was designed to shorten run time by further dividing data sets and distributing the resource load more evenly among resources its failure to complete faster indicates that too much time is wasted with scheduling additional processes. We observed that there does not appear to be a convincing reason to schedule more tasks then the number of available computing resources.





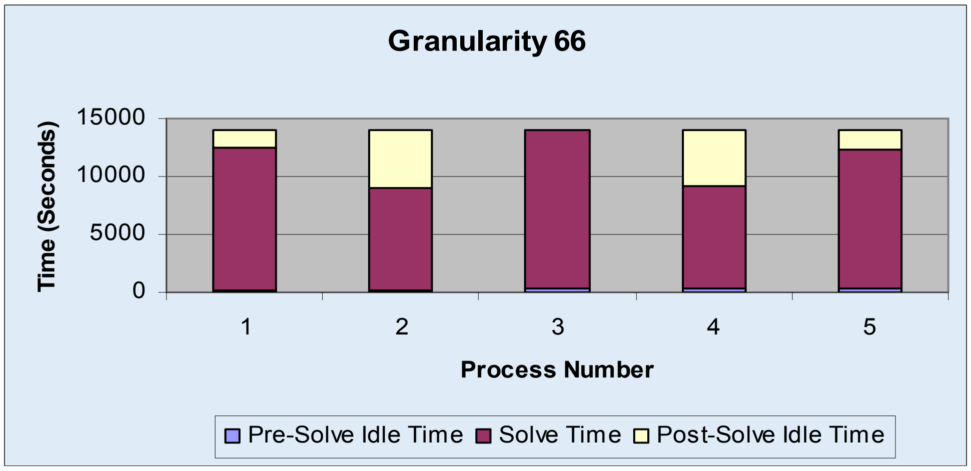
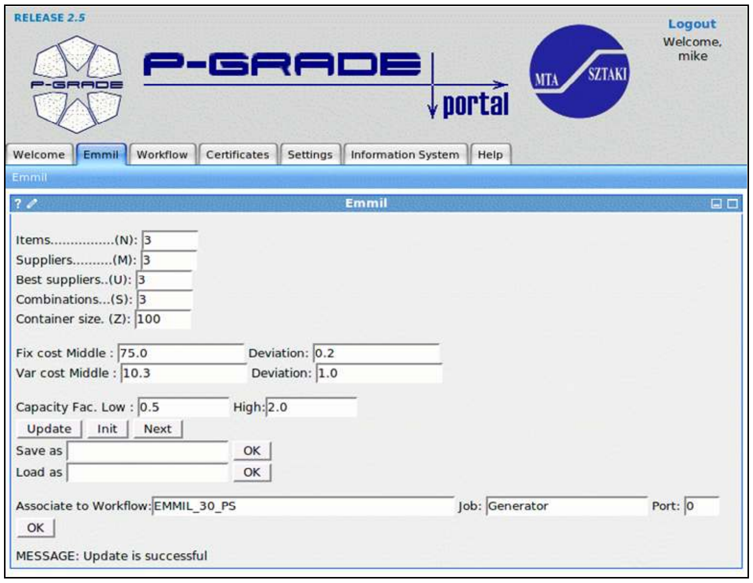


Figure 21 - Granularity Tests

4.2 EMMIL Data Input Portlet



Items................(N): 3
Suppliers..........(M): 3
Best suppliers..(U): 3
Combinations...(S): 3
Container size. (Z): 100

Fix cost Middle : 75.0 Deviation: 0.2
Var cost Middle : 10.3 Deviation: 1.0

Capacity Fac. Low : 0.5 High: 2.0

Save as
Load as

Associate to Workflow: EMMIL_30_PS Job: Generator Port: 0

MESSAGE: Update is successful

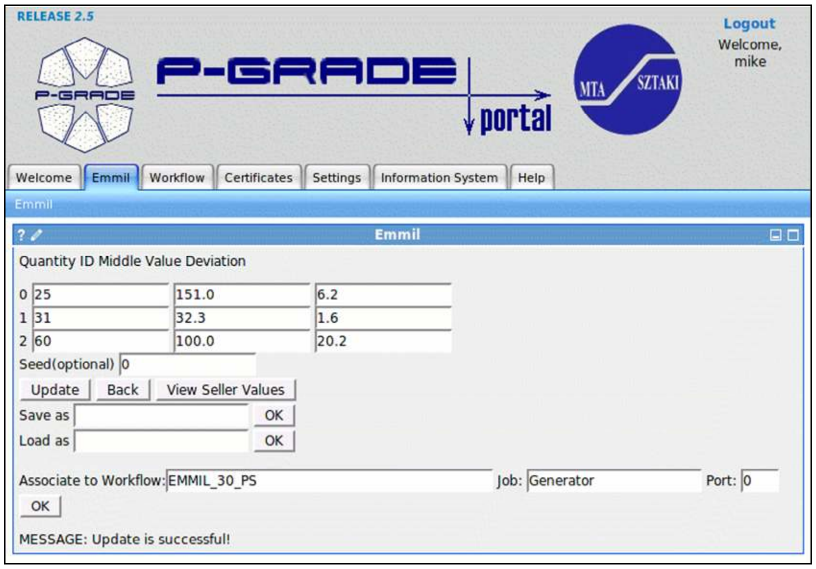
Figure 22 - EMMIL Data Entry

The EMMIL Data Input Portlet was implemented as described in Section 3.3. Preliminary work on the portlet was done with the assistance of colleagues at MTA-SZTAKI. This prototype allowed the user to input the basic application parameters and the means and standard deviations for the seller related items. We made the decision to extend the functionality of this existing model to support the generation of seller data. The result consists of four Java classes and two JSP pages.

The Java classes involved in this application were the EmmilHandler, EmmilDB, SessionUserData and Quantity. EmmilHandler extends ActionPortlet, a class from the Gridsphere Portlet library, which is responsible for handling all web-based events from the JSP pages. The EMMILDB class is responsible for remote storage of the application data, both to allow the user to save and load files within the portlet, and to associate the generator input file to a workflow. The SessionUserData class holds all the data the application needs while running. The Quantity class is used to hold the details about the product capacities.

The two JSP pages in the original portlet were EMMIL.jsp and EMMIL_2.jsp, these JSP pages provide the end user with an interface to create the generator input file.

EMMIL.jsp provides the user with an interface as can be seen in Figure 22 to set the basic application parameters, the mean and standard deviations for fixed and variable costs, and the high and low capacity limits. The EMMIL_2.jsp interface provides the user with the ability to specify the quantity, mean and standard deviation for each product (see). Each page also allows the user to save the current portlet data, for reuse in the portlet, or to associate it with a workflow.



Quantity ID Middle Value Deviation

0	25	151.0	6.2
1	31	32.3	1.6
2	60	100.0	20.2

Seed(optional) 0

Save as

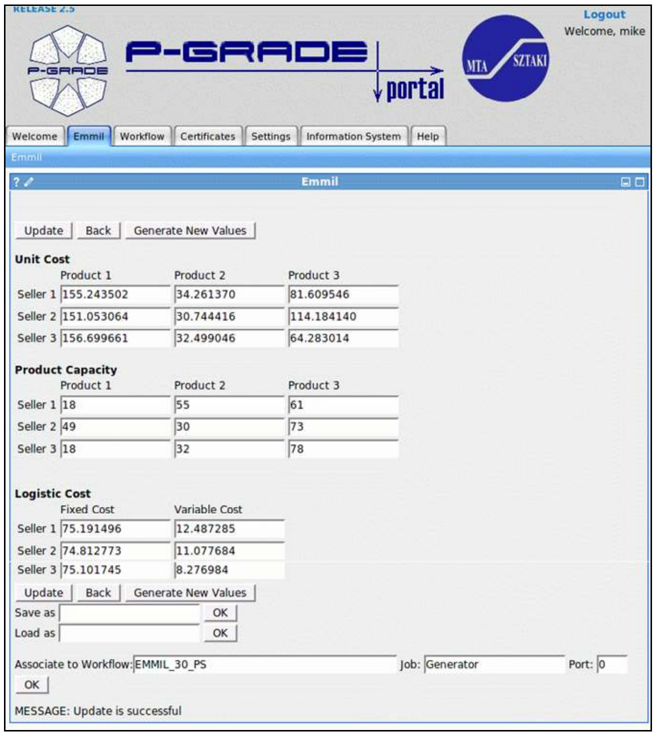
Load as

Associate to Workflow: EMMIL_30_PS Job: Generator Port: 0

MESSAGE: Update is successful!

Figure 23 - Mean and Standard Deviation Entry

In the modified version of this application two new Java classes and a new JSP page were added. Additionally, modifications were made to the existing files. The first class added was SellerData. This class stores all data related to the sellers, the unit costs and product capacities for each item, the fixed cost and the variable cost. The other class added was the DataGenerator class. This class uses the data stored within the SessionUserData class to generate the seller data using a standard deviation. To generate a standard deviation we made use of the central limit theorem approach, due to its ease of implementation as well as the fact that its performance was comparable to more complex methods. The new JSP page, EMMIL_3.jsp, as can be seen in Figure 23, displays the data generated by the Data Generator class to the user. Additionally the page allows the user to generate a new set of seller data which will erase any previous seller data values. The page also gives the user the ability to load and save the files as the previous pages did.



Unit Cost

	Product 1	Product 2	Product 3
Seller 1	155.243502	34.261370	81.609546
Seller 2	151.053064	30.744416	114.184140
Seller 3	156.699661	32.499046	64.283014

Product Capacity

	Product 1	Product 2	Product 3
Seller 1	18	55	61
Seller 2	49	30	73
Seller 3	18	32	78

Logistic Cost

	Fixed Cost	Variable Cost
Seller 1	75.191496	12.487285
Seller 2	74.812773	11.077684
Seller 3	75.101745	8.276984

Associate to Workflow: EMMIL_30_PS Job: Generator Port: 0

MESSAGE: Update is successful

Figure 24 - Costs and Capacity

In addition to the creation of these new files much of the work came in the modification of the existing portlet. Within the EmmilHandler class extensive modifications were made as a method needed to be written to handle the events created from any button clicks in the JSP pages. The EmmilDB and SessionUserData classes were both modified to store the generated seller data, stored in SellerData objects. Additionally the EmmilDB class was modified to write the new generator input file, which holds all seller data, rather than just the means and standard deviations. The final changes were made to Emmil_2.jsp which was modified to allow the user to input a seed for data generation, as well as by adding a button which allows the user to view the seller data.

4.3 EMMIL Enhancement

4.3.1 Product Capacity Filtering

The product capacity table was implemented as a double-dimensional array of primitive ints. This was populated with a uniform distribution whose possible values are defined by capFactorLow and capFactorHigh. These represent the lower and upper bounds of the distribution, respectively. Each variable stores a percentage of the desired product quantity. Values of 0.25 and 1.0, for example, would allow all capacities between 25% and 100% of the desired amount (inclusive) to be generated. This was done for testing purposes only; real-world situations would not use randomly assigned values.

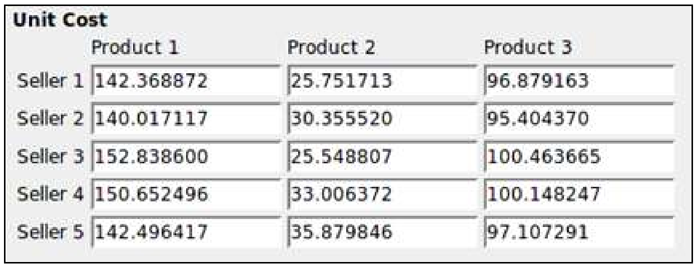
The preprocessing algorithm was implemented according to the pseudocode in Section 3.4.1. Language-specific considerations are described by in-line comments within the source code, and for the sake of brevity and readability will not be repeated here. Product capacity constraints needed by lp_solve are implemented in the EMMIL core. Since a single constraint can only govern one variable, actual constraint generation must take place in nested loops. Recalling that the X-variable used by lp_solve in this case represents combinations of seller, product and container, the following code is necessary:

```
for (j = 0; j<S; j++){  //For each seller
      gSj = getS(j,jID); //Get the seller index
      for (i = 0; i<N; i++) { //For each product
            for (t = 0; t<C; t++){ //Check over all containers
                  fprintf(out, "+1 %s - %d <= 0 ;\n",
                    varXgen(gSj,i,t), //Formats syntax
                    capacities[gSj][i]); //capacity table
            }
      }
}
```

4.3.2 Hopeless Job Filtering

The hopeless job filter was implemented in C according to the psuedocode presented in section 3.4.2. The implementation functions as originally intended and filters out the hopeless jobs which the algorithm was intended to remove. The impact of the filter is difficult to measure, as the length of time to process the filtered core jobs can vary anywhere from milliseconds to hours. The filter processes a job containing thirty sellers and ten products in under 200ms. As this is a realistic upper bound scenario for the model these findings demonstrate the effectiveness of the filter, as the algorithm completes in significantly less time than an average core job.

Depending on the input given to the application the filter can be very effective at removing jobs. Figure 25 - Unit Costs displays the unit prices of five sellers, in this scenario the fixed cost and variable cost were held constant. This scenario resulted in the filtering of a number of jobs; the results are displayed in Figure 26**.** The results in this table show that the filter is effectively removing jobs. The jobs filtered when S is two are all the jobs which do not contain the best seller, this shows that the scenario set up has a single powerful seller which can not be beat by the combination of any other two sellers. When S is three, however, two jobs are filtered out. This shows that some combinations which do not include the best seller are able to deliver a better value. Again, as in the hypothetical filter scenario that was produced, no jobs are removed when the selection size is higher


Unit Cost

	Product 1	Product 2	Product 3
Seller 1	142.368872	25.751713	96.879163
Seller 2	140.017117	30.355520	95.404370
Seller 3	152.838600	25.548807	100.463665
Seller 4	150.652496	33.006372	100.148247
Seller 5	142.496417	35.879846	97.107291

Figure 25 - Unit Costs

S=1	{1}***{2}{3}{4}{5}***
S=2	{1,2}{1,3}{1,4}{1,5} ***{2,3}{2,4}{2,5}{3,4} {3,5}{4,5}***
S=3	{1,2,3}{1,2,4}{1,2,5} {1,3,4}{1,3,5}{1,4,5} ***{2,3,4}{2,3,5}***{2,4,5} {3,4,5}
S=4	{1,2,3,4},{1,2,3,5} {2,3,4,5}
S=5	{1,2,3,4,5}

Figure 26 - Hopeless Discards

than four. The filter clearly shows its effectiveness in filtering jobs in the instances when S is low compared to the total number of sellers.

5 Conclusions and Future Work

This section details the conclusions that we've drawn from the data presented in Part 4, obtained using the methods described in Part 3. We have also pointed out avenues of future work in areas of theory and implementation. Section 5.1 presents our conclusions concerning the EMMIL model and its feasibility as a grid application, taking into account that all tests were run on SEE-GRID. The role of granularity in the parallelization process is considered, and a rough relationship between granularity, grid conditions, and input data is hypothesized. Section 5.2 suggests extensions to our implemented data entry portlet and discusses potential data output utilities. Finally, Section 5.3 contains many suggestions for improving the currently implemented EMMIL model. Many of these are based on the features described by Bruckner and Csekenyi [3]. Other arose from our own experiences with the code and its design.

5.1 EMMIL Grid Application Feasibility Testing

5.1.1 Single-Processor Testing

Analysis of our single-processor data has led to several conclusions concerning EMMIL and its implementation. First, the data represented in Figure 17 and Figure 18 suggest that the number of jobs generated by a given input set should not in and of itself be used to estimate overall completion times. The time per job is also a crucial factor, and the relationship between these is important in determining optimal granularity.

Second, the number of integer variables introduced into lp_solve has a very significant effect on the time taken to complete each job. This is not considered in the theoretical model, but represents a significant loss of efficiency in the implemented system. The number of integer variables in an EMMIL problem can be determined from its input data, using:

$$\mathrm{S}\left(1+\left\lceil \frac{1}{\mathrm{Z}}\sum_{i=1}^{N} Q_i \right\rceil\right)$$

Where S is the number of sellers per combination, Z is the container size, N is the number of products, and Q is the desired quantity of a product. Further empirical testing is needed to discover a more precise relationship between this number and the amount of processor cycles consumed by lp_solve.

The first and second conclusions lead naturally to our final conclusion based on single-processor testing. In a parallel environment, granularity heuristics should consider the number of integer variable constraints in an EMMIL problem. If time per job is likely high, a finer granularity may be beneficial in order to maximize parallelization. If time per job is low, a coarser granularity would help minimize grid overhead costs. The precise number of jobs assigned to each machine given fine or coarse granularity is a factor of the number of jobs that are created; this can be determined by $\binom{U}{S}$.

5.1.2 Grid-Based Testing

We have drawn several conclusions about the feasibility of EMMIL as a grid application and the role of granularity in overall efficiency. Note that in theory, EMMIL would complete faster on a grid than a single processor in most cases. The core algorithm is a parameter study, in which one process (in this case a mixed-integer linear problem solver) executes over a variety of independent datasets. Furthermore, a significant number of real-world situations can easily generate millions of jobs. For example, selecting six sellers to buy from out of fifty creates 15,890,700 independent processes. Only problems dealing with small parameter sets and low numbers of jobs would make a single processor desirable.

In practice, a number of external factors limit EMMIL's efficiency on SEE-GRID. The VO does not have dedicated resources, and what processors are available are often heavily utilized. Broker and resource waiting times can vary widely throughout the course of a day. Depending on the length and scope of a combinatorial auction this factor alone may make SEE-GRID an undesirable environment for EMMIL execution.

Our results in both single-processor testing (Sections 4.1.1 and 5.1.1) and grid-based testing (Section 4.1.2) have provided some indication of a granularity heuristic that could mitigate some of these problems. From a purely local perspective granularity should be based on an estimate of two factors: the number of jobs produced and the time taken to run per job. An upper bound on the former can be established by a combination of U and S; capacity and hopeless job filtering could potentially result in a lower amount of jobs actually passed to the solver. An algorithm to estimate the latter might approximate the number of integer variables generated by a set of input data, as explained above.

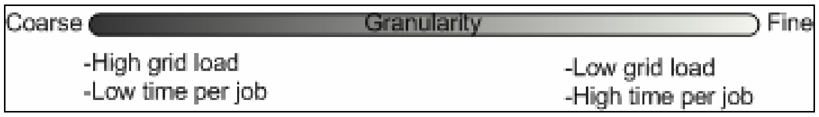


Figure 27 - Granularity

From the standpoint of a grid application, any granularity heuristic must account for current resource load and wait times. These are, after all, the essential point of a granularity metric. If scheduling time on the grid is high, representing a high grid load, then the algorithm should favor a coarse granularity, in order to minimize the time jobs spend waiting in queues. Lower grid loads should favor fine granularities, in order to capitalize on parallelization of the parameter study. These parameters are summarized in Figure 27. Finally, one of the greatest factors in the determination of optimal granularity is the number of processors available to that workflow. Our results indicate that running at the maximum concurrency limit generally achieves the best result. Little reason exists to set a granularity higher than the processor limit, since any threads beyond this threshold will be forced to wait until a resource becomes available. Although extra computing elements may become available over the course of a workflow, resources previously available may also have other jobs scheduled on them, resulting in a decrease in available computing elements. Note that a special case does exist, however, in which a finer granularity would be beneficial. This only occurs when one dataset takes an inordinately long amount of time to execute. However, by attempting to compensate for such a condition to much scheduling interference may be introduced, as occurred in our testing.

In conclusion, EMMIL is a viable grid application. Our test results on SEE-GRID indicate that jobs with a long solve time complete faster in a parallel environment despite overhead time costs. Process granularity is a critical factor in optimizing execution time and minimizing the effect of these overhead costs.

5.2 EMMIL Data Input Portlet

The EMMIL Data Input portlet facilitates the creation of advanced input files, which allows the user to have increased control over the parameters of the EMMIL model. This increased control allows the user to create specific scenarios by specifying the seller data according to their own desired criteria. The portlet also allows the user to easily associate the input file to a workflow, with no need to upload the file to the portlet. Unfortunately, the workflow cannot be run directly from the portlet as the user must still switch to the Workflow tab to execute the workflow. Additionally, there is no corresponding EMMIL output portlet. Currently the user must view the output through the Workflow Manager. Ideally one would be able to specify the input parameters, begin the execution of the model, and view the results a single application. Unfortunately at this time such an application does not exist, but it is theoretically possible to create such a program, given enough time, by extending existing Gridsphere and P-GRADE technology.

5.3 EMMIL Enhancement

The current EMMIL implementation is missing several features included in Dr. Bruckner's original model. Volume-based discounts, in which a price reduction is applied for buying some quantity of a product from a particular seller, are one of two discounting strategies discussed in Brucker and Csekenyi [3]. The second, total-spending discounts, provides price reductions if a buyer spent a specified amount of funds purchasing products from a particular buyer. Neither of these is supported by the current EMMIL implementation.

Furthermore, the original model included an expanded set of variables describing a buyer and each seller. Warehouse locations and shipping time are particularly notable omissions in the current implementation, and represent a level of geographical awareness that present several optimization strategies not yet considered. Sellers that are on the same transportation route could, for example, be favored in order to minimize logistic costs.

Arbitrary numbers of logistics providers are also currently unimplemented. The original model calls for each seller to have a list of valid logistic providers, while the implemented model has only one provider. This significantly impacts the algorithms used to select filtered sellers (U) and simplifies the computations performed by the solver. Inclusion of multiple logistics providers per seller may significantly affect computation times, and increase the feasibility of EMMIL as a grid application.

Several opportunities for future work also exist in the fields of algorithm analysis and creation. The mechanism by which hopeless sellers are filtered (see Section 3.4.2) may be amenable to improvement through use of a tree structure. This strategy has not been heavily analyzed, however. As is true with many algorithms which are currently implemented, support is not provided for arbitrary numbers of logistics providers.

In addition, the method by which sellers are filtered is currently very simplistic, does not allow discounting, and can not handle an arbitrary number of logistics providers. A variety of more sophisticated algorithms could be employed in its stead. Determining which technique is best is likely a matter of balancing increased pre-processing computation costs with an increased probability of arriving at an optimal solution.

File sizes could be minimized by writing only those combinations that will be solved to a given instance of the EMMIL core. Currently all possible combinations are stored in each file, even though only a fraction are used per process.

Finally, an increase in performance might be possible through increased interaction with the solver. Currently a problem is submitted and its solution is read; no record of previous solutions is kept. Cases have arisen, however, in which the relaxed solution to a given job (an upper bound on optimal price, obtained by relaxing constraints) is worse than the optimal solutions of previous jobs. In such instances there is no need to continue calculation, and the job could be discarded.

6 Bibliography

[1] Abdelnur, Alejandro, and Stefan Hepper. Java Portlet Specification. SUN Microsystems. 2003. 15 Apr. 2007 <http://jcp.org/aboutJava/communityprocess/final/jsr168/>.

[2] Abbas, Ahmarr. An Overview Grid Computing: a Practical Guide to Technology and Applications. 1st ed. Boston: Charles River Media, 2003.

[3] Bruckner, Livia K., and Jozsef Csekenyi. "Principles and Algorithms of EMMIL Marketplaces." Proceedings of the Seventh IEEE International Conference on E-Commerce Technology (2005).

[4] Clauson, Jens. "Branch and Bound Algorithms - Principles and Examples." University of Copenhagen (1999).

[5] Czajkowski, K., S. Fitzgerald, I. Foster, and C. Kesselman. "Grid Information Services for Distributed Resource Sharing." Proceedings of the Tenth IEEE International Symposium on High-Performance Distributed Computing (HPDC-10), IEEE Press (2001).

[6] Feller, M., I. Foster, and S. Martin. "GT4 GRAM: a Functionality and Performance Study."

[7] Foster, Ian, Carl Kesselman, and Steve Tuecke. "The Anatomy of the Grid." International J. Supercomputer Applications, 15 (2001).

[8] Foster, Ian. "Globus Toolkit Version 4: Software for Service-Oriented Systems." Journal of Computer Science and Technology 21 (2006).

[9] Foster, Ian. "What is the Grid? A Three Point Checklist." GRIDToday (2002).

[10] G. Singh et al., The Pegasus portal: Web based Grid computing, in Proc. of 20th Annual ACM Symposium on Applied Computing, Santa Fe, New Mexico, 2005.

[11] Geist, G.a., J.a. Kohl, and P.m. Papadopoulos. "PVM and MPI: a Comparison of Features." Calculateurs Paralleles (1996).

[12] Gropp, William, and Ewing Lusk. "PVM and MPI are Completely Different." Fourth European PVM - MPI Users\' Group Meeting.

[13] Harrington, Ramon, Martin, Danielle, and Winsnes, Carsten, Grid Portal Testing. WPI. 2006.

[14] Jacob, Bart, Michael Brown, Kentaro Fukui, and Nihar Trivedi. Introduction to Grid Computing. Vervante, 2005.

[15] Jamin, Amanda, and Domenic K. Giancola. GRID Portal Application Visualization. WPI. Worcester: WPI, 2005.

[16] J. Cao, S.A. Jarvis, S. Saini, and G.R. Nudd, BGridFlow: Workflow management for Grid computing, in Proc. of the 3rd IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGRID'03), pp. 198Y205, 2003.

[17] Kacsuk, Peter, and Gergely Sipos. "Multi-Grid, Multi-User Workflows in the P-GRADE Grid." Journal of Grid Computing (2006).

[18] Kacsuk, Peter, Gergely Sipos, Adrian Toth, Zoltan Farkas, Gabor Kecskemeti, and Gabor Hermann. "Defining and Running Parametric Study Workflow." MTA SZTAKI (2007).

[19] Keahey, K., I. Foster, T. Freeman, and X. Zhang. "Virtual Workspaces: Achieving Quality of Service and Quality of Life in the Grid." Scientific Programming Journal (2006).

[20] "Lp_solve Reference Guide." Sourceforge. 13 Apr. 2007 <http://lpsolve.sourceforge.net/5.5/>.

[21] Morgan, Steven S. A Comparison of Simplex Method Algorithms. 1997. University of Florida. 13 Apr. 2007 <http://www.cise.ufl.edu/~davis/Morgan/index.htm>.

[22] "P-GRADE Grid Portal Manual." LPDS. 15 Feb. 2007. SZTAKI. 12 Apr. 2007 <http://www.lpds.sztaki.hu/pgportal/manual/user/v25a/UsersManualRelease2_5.html>.

[23] Sun Microsystems. 05 Apr. 2007 <http://www.sun.com/>.

[24] Smarr, Larry, and Charles E. Catlett. "Metacomputing." Communications of the ACM 35 (1992): 44-52.

[25] Stevens, Rick, Paul Woodward, Tom Defanti, and Charlie Catlett. "From the I-WAY to the National Technology Grid." Communications of the ACM 40 (1997): 50-60.

[26] Thain, Douglas, Todd Tannenbaum, and Miron Livny. "Condor and the Grid." Grid Computing – Making the Global Infrastructure a Reality. Ed. F. Berman, A. Hey, and G. Fox. John Wiley and Sons, 2003.

[27] Welch, V., F. Seibenlist, I. Foster, J. Bresnahan, K. Czajkowski, J. Gawor, C. Kesselman, S. Meder, L. Pearlman, and S. Tuecke. "Security for Grid Services." Twelfth International Symposium on High Performance Distributed Computing (HPDC-12) (2003).

Appendix I: Single-Processor Test Data

Variable	Description
M	Number of total sellers
U	Number of filtered sellers
S	Number of sellers in a combination
N	Number of products
Q	Desired product quantity
P	Unit price
F	Fixed transportation cost
V	Variable transportation cost
Z	Container size

All timing measurements are in seconds.

M=20, N=10, U=10

Container Size Processor Time using times(&struct_tms)											
S	20	30	40	50	60	70	80	90	100	110	120	130
3			1801.31	466.94	219.39	95.27	77.79	58.30	38.44	39.94	41.33	26.77
4			7976.05	2274.61	1209.25	562.59	463.62	232.27	254.88	160.50	179.63	95.96
5			13769.00	4190.74	2957.68	1141.81	899.86	760.92	592.59	381.37	393.60	289.55
6			15781.71	4614.90	3795.87	1634.64	1276.68	922.68	763.29	712.57	509.72	370.76

M=20, N=10, U=10

Container Size (Job Time)												
S	20	30	40	50	60	70	80	90	100	110	120	130
3			15.94	4.13	1.94	0.84	0.69	0.52	0.34	0.35	0.37	0.24
4			37.98	10.83	5.76	2.68	2.21	1.11	1.21	0.76	0.86	0.46
5			54.64	16.63	11.74	4.53	3.57	3.02	2.35	1.51	1.56	1.15
6			75.15	21.98	18.08	7.78	6.08	4.39	3.63	3.39	2.43	1.77

M=20, U=10, S=3, Z=50

	Items (Execution Time)											
Quantity	1	2	3	4	5	6	7	8	9	10	11	12
25	0.11	0.16	0.32	0.29	0.89	0.91	2.02	2.06	5.77	5.89	9.48	7.07
50	0.12	0.29	0.44	0.79	1.63	2.73	4.69	8.21	13.29	18.66	26.25	6.9
75	0.21	0.35	1.19	1.62	10.3	10.01	53.45	47.58	398.71	161.31	1008.16	495.7

Appendix II: Grid-Based Test Data

Variable	Description
M	Number of total sellers
U	Number of filtered sellers
S	Number of sellers in a combination
N	Number of products
Q	Desired product quantity
P	Unit price
F	Fixed transportation cost
V	Variable transportation cost
Z	Container size

All measured times are in seconds

Granularity Graph Dataset 1.1

N	M	U	S	G	Jobs	Pre-Solve Idle Time	Solve Time	Post-Solve Idle time	Total Idle Time
15	10	10	6	42	5	303	609	337	640
15	10	10	6	42	5	356	420	473	829
15	10	10	6	42	5	303	588	358	661
15	10	10	6	42	5	360	574	315	675
15	10	10	6	42	5	360	613	276	636

Total Elapsed Time	1249

Granularity Graph Dataset 1.2

N	M	U	S	G	Jobs	Pre-Solve Idle Time	Solve Time	Post-Solve Idle time	Total Idle Time
15	10	10	6	21	10	287	336	1166	1453
15	10	10	6	21	10	1054	233	502	1556
15	10	10	6	21	10	261	303	1225	1486
15	10	10	6	21	10	312	370	1107	1419
15	10	10	6	21	10	312	193	1284	1596
15	10	10	6	21	10	329	182	1278	1607
15	10	10	6	21	10	818	262	709	1527
15	10	10	6	21	10	826	247	716	1542
15	10	10	6	21	10	838	255	696	1534
15	10	10	6	21	10	886	258	645	1531

Total Elapsed Time	1789

Granularity Graph Dataset 1.3

N	M	U	S	G	Jobs	Pre-Solve Idle Time	Solve Time	Post-Solve Idle time	Total Idle Time
15	10	10	6	10	21	544	96	1810	2354
15	10	10	6	10	21	998	80	1372	2370
15	10	10	6	10	21	1300	154	996	2296
15	10	10	6	10	21	1324	156	970	2294
15	10	10	6	10	21	1314	165	971	2285
15	10	10	6	10	21	1335	162	953	2288
15	10	10	6	10	21	1396	83	971	2367
15	10	10	6	10	21	1696	76	678	2374
15	10	10	6	10	21	1749	141	560	2309
15	10	10	6	10	21	1844	146	460	2304
15	10	10	6	10	21	1833	101	516	2349
15	10	10	6	10	21	561	102	1787	2348
15	10	10	6	10	21	1844	158	448	2292
15	10	10	6	10	21	2012	96	342	2354
15	10	10	6	10	21	687	68	1695	2382
15	10	10	6	10	21	600	150	1700	2300
15	10	10	6	10	21	658	111	1681	2339
15	10	10	6	10	21	871	139	1440	2311
15	10	10	6	10	21	906	135	1409	2315
15	10	10	6	10	21	971	106	1373	2344
15	10	10	6	10	21	983	124	1343	2326

Total Elapsed Time	2450

Granularity Dataset 2.1

N	M	U	S	G	Jobs	Pre-Solve Idle Time	Solve Time	Post-Solve Idle time
20	11	11	7	11	30	10293	1946	302
20	11	11	7	11	30	1026	1983	9550
20	11	11	7	11	30	4191	1046	7322
20	11	11	7	11	30	2693	1408	8458
20	11	11	7	11	30	2805	1365	8389
20	11	11	7	11	30	1024	1882	9653
20	11	11	7	11	30	10448	1809	303
20	11	11	7	11	30	1036	1750	9774
20	11	11	7	11	30	4293	1539	6728
20	11	11	7	11	30	1036	2609	8915
20	11	11	7	11	30	2400	2053	8107
20	11	11	7	11	30	5610	1272	5678
20	11	11	7	11	30	5564	2190	4806
20	11	11	7	11	30	1025	1308	10227
20	11	11	7	11	30	3662	1319	7579
20	11	11	7	11	30	2923	2666	6971
20	11	11	7	11	30	3701	2261	6598
20	11	11	7	11	30	1013	2668	8879
20	11	11	7	11	30	1036	1256	10268
20	11	11	7	11	30	4084	2354	6122
20	11	11	7	11	30	3024	1039	8497
20	11	11	7	11	30	1024	3246	8290
20	11	11	7	11	30	1036	1637	9887
20	11	11	7	11	30	1024	2997	8539
20	11	11	7	11	30	4120	1500	6940
20	11	11	7	11	30	5261	1506	5793
20	11	11	7	11	30	4044	2153	6363
20	11	11	7	11	30	2315	3639	6606
20	11	11	7	11	30	4473	1067	7020
20	11	11	7	11	30	4997	1159	6374
						Total Compute time	56627	
						Total Run time	11226	
						Speed up	5.0442722	

Total Elapsed Time: 12560

Granularity Dataset 2.2

N	M	U	S	G	Jobs	Pre-Solve Idle Time	Solve Time	Post-Solve Idle time
20	11	11	7	11	10	359	8932	11149
20	11	11	7	11	10	3936	5682	10822
20	11	11	7	11	10	359	3560	16521
20	11	11	7	11	10	3796	5808	10837
20	11	11	7	11	10	386	5139	14934
20	11	11	7	11	10	405	7352	12684
20	11	11	7	11	10	368	6965	13108
20	11	11	7	11	10	407	4862	15172
20	11	11	7	11	10	420	4116	15960
20	11	11	7	11	10	419	5586	14437
						Total Compute time	58002	
						Total Run time	9259	
						Speed up	6.2643914	

Granularity Dataset 2.3

N	M	U	S	G	Jobs	Pre-Solve Idle Time	Solve Time	Post-Solve Idle time
20	11	11	7	66	5	220	12285	2082
20	11	11	7	66	5	220	8743	5625
20	11	11	7	66	5	279	13673	636
20	11	11	7	66	5	292	8815	5481
20	11	11	7	66	5	292	12018	2278
						Total Compute time	55534	
						Total Run time	13732	
						Speed up	4.0441305	

Wallclock Time: 14588

Granularity Dataset 2.4

N	M	U	S	G	Jobs	Pre-Solve Idle Time	Solve Time	Post-Solve Idle time
20	11	11	7	17	20	428	3567	12934
20	11	11	7	17	20	498	2433	13998
20	11	11	7	17	20	506	2353	14070
20	11	11	7	17	20	504	404	16021
20	11	11	7	17	20	505	1108	15316
20	11	11	7	17	20	516	1345	15068
20	11	11	7	17	20	517	2740	13672
20	11	11	7	17	20	518	2916	13495
20	11	11	7	17	20	530	2369	14030
20	11	11	7	17	20	531	2204	14194
20	11	11	7	17	20	541	4551	11837
20	11	11	7	17	20	473	4036	12420
20	11	11	7	17	20	542	4172	12215
20	11	11	7	17	20	446	3084	13399
20	11	11	7	17	20	475	2796	13659
20	11	11	7	17	20	473	2328	14129
20	11	11	7	17	20	485	2179	14266
20	11	11	7	17	20	497	1987	14446
20	11	11	7	17	20	498	2888	13544
20	11	11	7	17	20	500	2881	13549
						Total Compute time	52341	
						Total Run time	4664	
						Speed up	11.222341	

Wallclock Time: 16930

Appendix III: Combinations

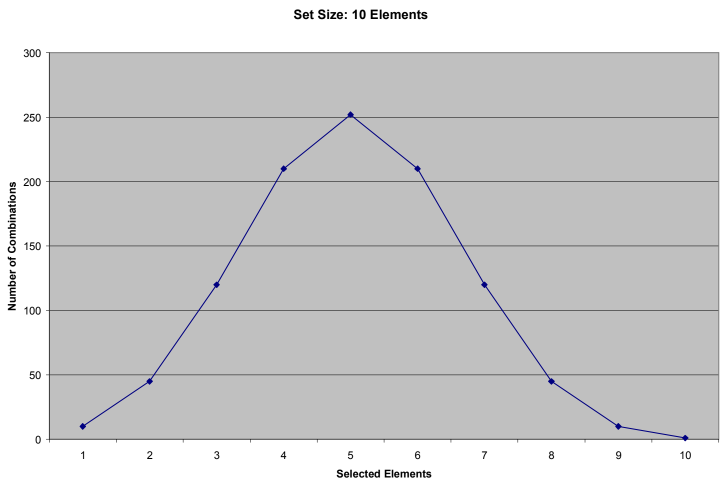
The combination function finds the number of ways to combine some number of unique elements in a set. Combinations are defined as unordered collections of elements from a specified set, and are calculated by:

$$\binom{n}{k} = \frac{n!}{k!(n-k)!}$$

Where n is the cardinality a set containing at least k elements, and k is the number of elements selected from n. For example, if one has a set of 10 elements and wishes to know how many combinations of six elements from this set is possible, the problem would be represented as:

$$\binom{10}{6} = \frac{10!}{6!(10-6)!} = \frac{3628800}{17280} = 210$$

A graph of all combinations of size 1 to 10, drawn from a set of ten, would be as follows:



N	K	Combinations
10	1	10
10	2	45
10	3	120
10	4	210
10	5	252
10	6	210
10	7	120
10	8	45
10	9	10
10	10	1

As you can see in the above graph and table, the highest number of combinations occurs at a selection size of five. This is due to the way combinations are calculated; *k!(n-k)!* is always smallest when *n-k* = *k*.