Resource Allocation in Heterogeneous Computing Systems

First Semester Report
Fall Semester 2007

By
Christopher Klumph
Kody Willman

Prepared to partially fulfill the requirements for ECE401

Department of Electrical and Computer Engineering
Colorado State University
Fort Collins, Colorado 80523

Report Approved:

________________________________________
Project Advisor

________________________________________
Senior Design Coordinator
Abstract

The increasing popularity of Virtual World Environments (VWE) presents a growing need for utilizing heterogeneous distributed and parallel computing systems in order to meet the demand for scalable computational capacity. Massive multiplayer online games allow multiple users to simultaneously play interactive games by communicating to a main server. Adding additional computer architecture because of user-base increase is both expensive and non-feasible, as demand and success of such a game cannot be viably predicted. Thus, resource allocation in heterogeneous computing systems is studied in order to provide practical solutions to reducing the load on the main server by creating secondary servers from existing players.

The research done this semester by the Senior Design students focused on studying, designing, and implementing static resource allocation heuristics for such a heterogeneous system as described above. There are many algorithms and heuristics used to solve resource allocation problems, but two were chosen for this project. Specifically, a modified Genetic Algorithm and a Two-Stage Minimization heuristic was analyzed, programmed, and simulated. Design problems facing such a task include how many computers should be made secondary servers, which specific computers should be secondary servers, and which players should connect through those secondary servers to the main server.

The results thus far confirm that a viable solution can be found to this problem, allowing more users to connect to a given VWE via distributed server architecture, as compared to traditional single-server architecture. Since faster connection times can be attained, user experience increases satisfaction and wait time decreases. Moving forward, heuristics and preliminary results can be improved upon in order to achieve usable realworld applicability.
# Table of Contents

Resource Allocation in Heterogeneous Computing Systems ................................................. 1
Abstract .................................................................................................................................. 2
Table of Contents .................................................................................................................... 3
List of Figures .......................................................................................................................... 4
List of Tables ........................................................................................................................... 4
Chapter I - Introduction .......................................................................................................... 5
Chapter II – Previous Work .................................................................................................... 8
Chapter III – Current Work ................................................................................................... 9
  Genetic Algorithm .................................................................................................................. 9
    Details of a genetic algorithm ......................................................................................... 10
    Problems with original genetic algorithm .................................................................... 12
  Genitor ................................................................................................................................ 13
    Genitor overview ........................................................................................................... 13
    Genitor results ................................................................................................................. 17
  Min-Min RT ......................................................................................................................... 18
  RT Iterative Minimization1 .............................................................................................. 19
  RT Iterative Minimization2 .............................................................................................. 22
Chapter IV – Conclusion and Future Work ......................................................................... 23
References .............................................................................................................................. 26
Bibliography ............................................................................................................................ 26
Appendix A - Abbreviations ................................................................................................. 27
Appendix B - Budget .............................................................................................................. 27
Appendix C - Performance Metric ....................................................................................... 28
Appendix D - Lower Bound ................................................................................................. 28
Appendix E – RT Iterative Minimization ............................................................................ 29
Acknowledgements ............................................................................................................... 29
List of Figures
Figure 1 Client Server Architecture
Figure 2 Secondary Server Architecture
Figure 3 Results for heuristics with the computational parameters of the MS set to: b = 0.03 and c = 0.06.
Figure 4 Example of Original Genetic Algorithm Chromosome
Figure 5 Genetic Algorithm Pseudo Code
Figure 6 Example of New Genetic Algorithm Chromosome
Figure 7 Genitor Algorithm Pseudo Code
Figure 8 DIM Pseudo Code
Figure 9 Procedure for Min-Min RT

List of Tables
Table 1 Preliminary Genitor Results
Table 2 Incorrect RT Iterative Minimization Results
Table 3 Good RT Iterative Minimization Results
Chapter I - Introduction

The topic of this project is resource allocation in a Massive Multiplayer Online Game, (MMOG) in which multiple players connect to a virtual game world. The purpose of this research is to maintain an acceptable system performance of a single-server architecture without increasing the processing power of the main server (MS). The proposed solution is to convert users to secondary servers (SS) that assist the MS in computation. In the client/server solution shown in Figure 1, all users connect to the MS, leaving the MS as the only machine performing computation. In the SS solution shown in Figure 2, the MS and SSs perform computation and the MS resolves conflicts among users and SSs connected to it.

![Figure 1 Client Server architecture](image1)

![Figure 2 Secondary Server architecture](image2)

We were asked to design, implement, evaluate, and analyze the performance of different static mapping heuristics used to minimize an objective function. But first we need to define the problem we are going to be attempting to solve. For this system, both MS and SS’s are doing computation. The MS resolves conflicts between players connected to the MS and the SS conflicts, the SS resolves conflicts between players...
connected to it, or sends it to the MS for final verification of conflict resolution. To simplify this we have assumed that all activity between players is considered identical, and the communication time from any player A to any player B is the same as from player B to player A. The communication times do not change during a player’s session, and are independent of the number of players connected to the SS or MS. Letting Comm(A,B) represent the communication time between player A and player B, and Comp(C) representing the computation time at player C we can now solve for the communication and computation times at different servers.

At any SS a we find the computation time to be.

\[ \text{Comp}(SS_a) = \mu_a x (n_a)^2 \]

Where \( \mu_a \) is the computational constant at secondary server a.

At the MS we need to consider all the players connected directly to it as well as all the players connected indirectly to it. Computation time of the MS is calculated as.

\[ \text{Comp}(MS) = c x n_{\text{secondary}} + b x (n_{\text{main}} + n_{\text{nss}})^2 \]

Where \( b \) and \( c \) are computational constants of the MS, \( n_{\text{secondary}} \) is the number of players connected to SS, \( n_{\text{nss}} \) is the number of SS’s and \( n_{\text{main}} \) is the number of players connected directly to MS. Once we have these formulas in mind on how to solve the computation time we need to know how to solve the communication time which we will call round trip time (RT). Please see Appendix C on how we computed this. Since we want to have the smallest round trip time even for the player with the worst connection we need to provide a lower bound that we try to attain with our heuristics. Please see Appendix D on how we computed this. Well now that we know what we need to solve for and how we need to solve it, we need to know what we are given to be able to solve this.
Luis Briceño, a PhD student that we are working with gave us 100 data files that contain the inputs for our simulation. We are going to be working with 200 players and each data file is arranged in a matrix with the following numbers; first number being the player ID number (x), second the computational constant \((\mu_a)\) for that player, third the connection time to the main server from that player, then the connection time for player x to every other player (player 1 in column 4, player 2 in column 5, player 3 in column 6 etc.) in the game.

Once all relevant information concerning our project was received, we took off and decide to get to work. Chapter two describes the previous work that had been done on this project that we were given to help understand what we were to do. Chapter three discusses our current work in detail and the different heuristics that we have implemented in trying to solve this problem. There are several subsections in chapter three that discuss our individual heuristics. Chapter four is our conclusion and the plans for future work in simplifying and expanding on our current work.
Chapter II – Previous Work

Previous types of resource allocations that have been implemented include a Tabu search heuristic, a dual iterative minimization, a discrete particle swarm optimization, and an ant colony optimization. Tabu search enhances the performance of a local search method by storing the previously visited areas in the search space using a Tabu list. Dual iterative minimization represents a solution as a vector whose $i^{th}$ element indicates the way user $i$ is connected to the MS. Discrete particle swarm optimization uses an algorithm based on the flocking behavior of animals such as birds and fish. An ant colony optimization is a population based approach that mimics the path finding behavior of ants, wandering randomly looking for food.

There have also been a genetic algorithm (GA) and RT Iterative Minimization, which did not yield good results, so we were asked to re-implement these hoping to yield a better result which leads us to our current work.
Chapter III – Current Work

Genetic Algorithm

One of our tasks was creating a genetic algorithm heuristic to find viable solutions to the MMOG problem. Genetic algorithms are a family of algorithms that search a global search space using techniques from evolutionary biology such as chromosome representation, selection, crossover, mutation, and inheritance. Our task was to implement such an algorithm in order to find results that would be compared to other heuristics and the theoretical lower bound (See Appendix C).

The first phase of the project was intensive research. We had never even heard of genetic algorithms, and had a lot of learning to do. Professor Siegel and Luis Briceño were very instrumental in getting us caught up to speed on the various algorithm designs. At the heart of a genetic algorithm is the chromosome. A chromosome is a representation of a possible solution. It is a mapping of a specific combination of players and secondary servers, though it did not originally contain connectivity information. Figure 4 shows an example of a simple chromosome.

![Figure 4. Example of Original Genetic Algorithm Chromosome](image)

For this example containing eight players, a ‘1’ denotes that a player is a secondary server, whereas a ‘0’ denotes that it is a player that is either connected to a secondary server or is connected directly to the main server. Player 1 may be connected to either
players 2, 3, or 7, since all represent secondary servers. The specific mapping of the
players was determined using a greedy Min-Min algorithm which is a common and
simple resource allocation algorithm [1, 2]. The pseudo code for our original genetic
algorithm is as follows:

(1) A population of 100 chromosomes is generated randomly
(2) For each chromosome, the $R_{T_{\text{max}}}$ is calculated after mapping using Min-Min
    heuristic
(3) While 10 minutes have not elapsed or there are 10,000 consecutive iterations with
    no improvement in best known solution
    (a) Select 50 pairs of parents using roulette wheel selection
    (b) Generate 100 offspring using a two-point crossover
    (c) Randomly mutate (with 3% chance) every element in each new offspring
        chromosome
    (d) Replace old population with new, making sure that the best known
        chromosome so far is included in new population
(4) Output best solution (i.e. lowest $R_{T_{\text{max}}}$)

Figure 5. Genetic Algorithm Pseudo Code

Details of a genetic algorithm

The first thing a genetic algorithm must have in order to work is a population—a
collection of chromosomes. In our case we used a randomly-generated population of 100
chromosomes. A Min-Min greedy algorithm is then used to evaluate the $R_{T_{\text{max}}}$ of each
chromosome by first mapping players to servers using the method described in Figure 9.

Once all players are mapped, a straightforward evaluation is possible and the
chromosome’s $R_{T_{\text{max}}}$ can easily be calculated. A table of each chromosome’s $R_{T_{\text{max}}}$ is
maintained throughout the program’s execution. The next step in the algorithm is to
select 50 pairs of parents using Roulette Wheel Selection. The theory behind Roulette
Wheel Selection is modeled after a roulette wheel that is spun and ends with the selection
of a random “slice” of the wheel. Roulette Wheel Selection is a “fitness proportionate
selection” [3, 4] which gives better, or “fit,” chromosomes a higher chance of being randomly selected by apportioning a larger area of the wheel. Thus, desirable characteristics are more likely to be passed on to subsequent generations via inheritance.

Once all pairs of parents are selected, a two-point crossover is performed. Two points are chosen randomly within the range of chromosome indices (which equals the number of players). All elements between the points from the first chromosome are swapped with the elements between the same points in the second chromosome. Thus, information is traded between the chromosomes, which now represent entirely new solutions.

The next step is mutation, in which each element in every new offspring chromosome is randomly mutated with a 3% chance. Thus, if an element within the chromosome is chosen to be mutated, it becomes a ‘0’ if it was a ‘1’ and becomes a ‘1’ if it was a ‘0’. If a chromosome is mutated, it represents a new solution, since the set of SSs has changed.

There are now 50 pairs of new chromosomes. They must all be evaluated just as the initial population was evaluated using a Min-Min mapping algorithm. The $RT_{\text{max}}$ of each chromosome is found and recorded. The new chromosomes replace their parents, except in the case of the best overall chromosome. If the previous generation contains a chromosome whose $RT_{\text{max}}$ is smaller than any of the new ones, it will be kept and the slowest offspring chromosome will be discarded. There is now an entirely new population of chromosomes. The steps of selection, crossover, mutation, and evaluation continue until one of two stopping criteria is met: 10,000 iterations of no change in the best known solution or 10 minutes. Once a stopping criteria is met, the best known
solution is outputted, which represents which players should be made secondary servers, and a Min-Min algorithm will once again tell who should connect to each server (SS or MS).

Once we understood on a fundamental level how a genetic algorithm worked, we then started to actually code a working version. The original plan was to use the Java language, since we had more experience with it than with C++. However, about half-way through the coding process, we found that Java made it near impossible to pass arrays to functions using pass-by-reference. Thus, any methods dealing with multiple arrays would be tedious to code and would slow down the program immensely. Pass-by-reference calls are very simple in C++ and we decided to convert the code we had to C++ and continue the duration of project using C++. As is the case with most programming projects, debugging became the longest phase of the coding life-cycle. After about 4 weeks of coding, debugging, and recoding, we had a working genetic algorithm, albeit crude. Using this code, we achieved an average $RT_{\text{max}}$ of 427.76 time units after 100,000 iterations (stopping criteria was ignored for the time being for testing purposes). Though not completely unreasonable in a real-world environment, it was still far away from the theoretical lower bound of 71.73 time units.

**Problems with original genetic algorithm**

Even though we had a working genetic algorithm, it was plagued by multiple problems. For one, it was incredibly slow. Each population took a long time to generate via selection, crossover, mutation, and evaluation. If sped up, perhaps a better solution would be found, since at every iteration, an entirely new generation is generated instead
of just working with a few chromosomes at a time. Along with being slow, it was also not converging to a minimum very efficiently.

But beyond being slow and not providing optimal solutions, our algorithm faced a much more serious problem. Professor Tony Maciejewski pointed out in one of our preliminary presentations to the Heterogeneous Computing group meetings that our chromosomal representation was flawed. In short, it didn’t represent a full solution and therefore our algorithm was not searching the whole search space, but only a very limited scope of it. The reason is that our chromosome only gave us the list of who was a secondary server but no information on who was connected to those secondary servers. A Min-Min heuristic determined which players should be connected to each server at the time of the evaluation function and therefore the search space was dependent on the specifics of the Mini-Min algorithm. Since our Min-Min was a greedy algorithm by nature, we not only never considered every possible mapping, but we weren’t even guaranteed the possibility of considering every valid mapping.

Our algorithm needed some major changes. All of these short-comings were attacked at the same time via different fixes, as described in the next chapter.

**Genitor**

**Genitor overview**

The most crucial thing that had to be changed in our genetic algorithm in order to even guarantee a search of the full massive search space was to change how we represented solutions using chromosomes. Instead of simply containing information on who was a
secondary server, we had to represent a complete mapping of the players. The chromosome must not only contain information on who is a secondary server, but must also show who is connected to each server, whether a secondary server or main server.

An example of the new chromosomal representation can be seen in figure 6:

<table>
<thead>
<tr>
<th>Player #</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>3</td>
</tr>
<tr>
<td>4</td>
</tr>
<tr>
<td>5</td>
</tr>
<tr>
<td>6</td>
</tr>
<tr>
<td>7</td>
</tr>
<tr>
<td>8</td>
</tr>
</tbody>
</table>

| 2 | SS | 2 | M | M | SS | 6 | 6 |

Figure 6. Example of New Genetic Algorithm Chromosome

In this example of eight players, players 2 and 6 are secondary servers, denoted by “SS”. As such, other players can connect through them instead of directly to the MS. An ‘M’ denotes that player is connected directly to the main server, as seen with players 4 and 5. Players 1 and 3 are connected to the SS player 2, and players 7 and 8 are connected to the SS player 6. Thus, this chromosome represents a full mapping of players and servers.

Making this change in the chromosome, although likely to improve our results, also forced us to modify most of our code in order for all methods to be able to correctly handle the new representation. Despite this downside, we also realized that our evaluation function would be much simpler because we no longer need the Min-Min algorithm to map the unmapped players to secondary servers. Thus, the evaluation was not only more straightforward and easy to understand, but would be orders of magnitude faster. Player mapping would not need to be determined, and evaluation would consist of
only calculating what the \( R_{T_{\text{max}}} \) was for each chromosome based on who was connected to whom.

Now that we had a correct representation of solutions encoded as chromosomes, the next change was to the very way that the genetic algorithm worked. The traditional genetic algorithmic approach we were using was very slow because a new population was generated each iteration. Since our problem presented such a massive search space, it simply was not viable to continue using this approach. We therefore changed our algorithm to a Genitor heuristic. A Genitor, while still part of the family of genetic algorithms, works a little bit differently. It is a steady-state genetic algorithm and is still based on the same techniques from evolutionary biology, but generates new populations much faster, and therefore hopefully converges to an optimum solution in less time [5]. The pseudo code for the Genitor is as follows:

1. A population of 200 chromosomes is generated randomly
2. For each chromosome, the \( R_{T_{\text{max}}} \) is calculated and chromosome is sorted into ranked list
3. While 10 minutes have not elapsed or there are less than 10,000 consecutive iterations with no improvement in best known solution
   a. Select two parents using roulette wheel selection function
   b. Generate 2 offspring using a two-point crossover
   c. Randomly mutate (0.1% chance) every element in each new offspring chromosome
   d. Rank new offspring chromosomes based on \( R_{T_{\text{max}}} \) and sort into ranked list
4. Output best solution (i.e. lowest \( R_{T_{\text{max}}} \), which will be at top of ranked list)

Figure 7. Genitor Algorithm Pseudo Code

The next improvement added to the program was seeding it with a known solution that would hopefully help it converge faster. For this, we used a Dual Iterative Minimization (DIM) algorithm to quickly find a seed value for the Genitor. This way, the algorithm would have at least one solution in its ranked list that was likely to be
comparably better than those randomly generated at the initialization of the program. This would hopefully cause the program to converge to an optimum solution quicker than if left solely to random chance. The pseudo code for the DIM can be seen in Figure 8.

(1) Mark all players as unassigned
(2) For each unassigned user ($u$) in a fixed arbitrary order
   (a) Define $\text{minRT}$ as the $RT$ if $u$ is connected directly to the MS
   (b) Among all potential host ($PH$), find the $PH$ that minimizes $RT$ of $u$
       connected to the MS through $PH$ ($RT_{u\rightarrow PH\rightarrow MS}$)
       (i) If $RT_{u\rightarrow PH\rightarrow MS}$ is less than $\text{minRT}$ then attach $u$ to $PH$, and convert $PH$ to an SS if it is not one yet
       (ii) Else, attach $u$ directly to the MS
   (c) Mark user $u$ as assigned
(3) Output final solution

Figure 8. DIM Pseudo Code

With the program seeded with a chromosome found using the DIM, it then proceeded through its iterations using the same evolutionary techniques based on desirable inheritance, as described in the previous section. One major difference between our original genetic algorithm and the Genitor is how the Genitor performs the crossover and mutation methods. A two-point crossover is still used to swap information between the two chromosomes, but since they both represent full solutions, the new offspring may be invalid mappings. For example, a player may have been mapped to a valid SS which is no longer an SS because of the crossover. Thus, the chromosomes now have to be checked and, most likely, fixed after each crossover. This was done by mapping each player that was not an SS and was not mapped to a proper SS to a randomly chosen pre-existing SS or to the MS. The same fixing method had to be done after the mutation, since the same problems could arise. So although the Genitor simplified the evaluation of
the chromosomes and had a faster convergence, the crossover and mutation methods were more complicated and therefore slower.

**Genitor results**

Now that we had completely redone our genetic algorithm into a Genitor (which, remember, is still part of the family of genetic algorithms), it was time to see if our improvements really made a difference. To our delight, the Genitor was faster than the genetic algorithm and also produced better results. The next step was tweaking input parameters in order to empirically determine the best simulation parameters. Inputs such as population size, mutation rate, and iteration count were varied in order to find the optimums. A population size of 200 chromosomes and mutation rate of 0.1% were found to produce the best results. Preliminary results can be seen in Table 1.

<table>
<thead>
<tr>
<th>Simulation file</th>
<th>Result (in time units)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data1</td>
<td>122.06</td>
</tr>
<tr>
<td>Data2</td>
<td>122.44</td>
</tr>
<tr>
<td>Data3</td>
<td>125.18</td>
</tr>
<tr>
<td>Data4</td>
<td>148.76</td>
</tr>
<tr>
<td>Data5</td>
<td>123.31</td>
</tr>
<tr>
<td>Data6</td>
<td>116.4</td>
</tr>
<tr>
<td>Data7</td>
<td>125.26</td>
</tr>
<tr>
<td>Data8</td>
<td>118.22</td>
</tr>
<tr>
<td>Data9</td>
<td>117.5</td>
</tr>
<tr>
<td>Data10</td>
<td>124.43</td>
</tr>
</tbody>
</table>

Table 1. Preliminary Genitor results

While these results are still not extremely close to the theoretical lower bound of 71.73 time units, they are far better than the original genetic algorithm average of 427.76 time units. We were therefore very pleased with the performance of the Genitor heuristic.
**Min-Min RT**

The Min-Min RT heuristic is based on the concept of Min-Min heuristic [2]. The Min-Min RT heuristic is widely used in the area of resource allocation [1, 2]. It takes a given set of secondary servers and assigns the remaining players to that set of secondary servers or to the main server. The procedure for Min-Min RT is shown in Figure 9. This heuristic is used by the RT Iterative Minimization.

1. Given a predetermined set of SSs, all users that are not in the set of SS are marked as unassigned.
2. For each unmapped user, the server (main or secondary) that gives the minimum RT is determined (first minimum).
3. The best paired user/server (i.e., with smallest RT) among all the pairs generated in (2) is selected (second minimum).
4. The user in the best pair selected in (3) is then assigned to its paired server.
5. Steps (2) through (4) are repeated until all tasks are assigned.

Figure 9 Procedure for Min-Min RT

To elaborate more on the Min-Min RT this code creates an assignment mapping to the unmapped players. It takes the minimum round trip time for each player to the main server, either directly or indirectly through a secondary server. From this list of minimum round trip times it identifies the minimum time to main server, either directly or indirectly through a secondary server. It then assigns the player with that minimum time to whatever route it takes to get to the main server. The Min-Min RT then has to update all connection and computation times and re-solve all the round trip times for the remaining unmapped players. The mapping has changed and now a faster connection could have become available. That is why we had to re-evaluate it after each assignment of a single player. This type of mapping allows for three different types of mapped players. First, there are players that are mapped directly to the main server with no one connected to them, because their best connection time indicated that trying to connect
through someone else was not as good or their computational constant for other players mapping through them was bad. Second, there are players that are mapped directly to the main server and have other players mapped to them. These are the secondary servers and have a good connection time to the main server and have an acceptable computational constant so the data received from other players gets computed, in most cases, efficiently. Third, there are players that connect through a secondary server, which indicates that their connection time to the main server is not good.

There are two variations of this Min-Min RT used in the RT Iterative Minimization. Version one, Min-Min RT1, is the above-mentioned version, where unmapped players can be mapped as secondary servers, mapped directly to the main server, or mapped through a secondary server. Version two, Min-Min RT2, is used for the initial mapping of \( k \) secondary servers in RT Iterative Minimization. This second version does not allow for the creation of new secondary servers. That is so we can keep the initial \( k \) secondary server list to test how efficient that random mapping works. Version two can create players mapped directly to the main server, and players mapped to the main server through one of the randomly generated \( k \) secondary servers.

Another function of the Min-Min RT that is used in both versions is the ability to find the maximum round trip time. This is used multiple times in RT Iterative Minimization to find the player with the maximum round trip time to try and reassign them a better time to generate a better overall mapping.

**RT Iterative Minimization**

This greedy heuristic consists of two phases. In phase 1, we iteratively adjust the number of secondary servers (by adding or removing) to minimize \( RT_{\text{max}} \). In phase 2, we
swap the secondary servers with users to minimize $RT_{\max}$. This heuristic also uses the Min-Min RT heuristic described for the RT Iterative Minimization to convert a set of secondary servers to a mapping. Because RT Iterative Minimization uses the Min-Min RT heuristic, some users marked as a secondary server may not have users attached to it. The procedure for RT Iterative Minimization is shown in Appendix E.

This was the first implementation that we tried for the RT Iterative Minimization. In Phase1, we would randomly pick a number $k$ between 1-200 that would become the number of secondary servers. We then randomly assign these $k$ players as secondary servers and finish the mapping with the first version of the Min-Min RT. Next, we find the player that has the largest $RT_{\max}$ and then, depending on what it was originally, either assign it as a player connected through a secondary server or as a player connected to the main server with the possibility of someone connecting to it later. We repeat this for 1000 iterations or until there is no improvement. Phase2 would then find the player with the largest $RT_{\max}$ and again, depending on how they connected to the main server, either swap them with a secondary server, or swap them with a player connected through a secondary server. We repeat this 1000 times or until there is no improvement and then output the resulting optimized round trip time.

After many trials and errors in coding we had a working version that gave us results. Many hours of compiling and running 20 iterations of two or three data file runs to the Phase1 results were in the neighborhood of around 100-90 time units, and the Phase2 results were about 105-95 time units, so going from Phase1 to Phase2 caused the results to become worse. The results were never intended to become worse. This caused us to go back and look for errors in the code. This error was corrected by making sure we
always kept the best solution even if it remained the previous solution. After finding this error and fixing it we again ran a few test runs to see if we were getting bad results.

Our next batch of results running 20 iterations of two or three data files, now showed that Phase1 was getting around 75-65 and Phase2 was getting around 70-60. We were really happy about these results. The previous implementation was yielding “ok” results around 92. These results were below 70, and were the best yet. Presenting these results to Luis, he immediately had suspicions that we had done something wrong. The theoretical lower bound (see Appendix C) was 71.7, meaning that it was the best possible solution that could be found by these types of heuristics. We then had to go back and print out one of the mappings that was in the 60 range and calculate all 200 players connection times and computation times by hand to see if was indeed in the 60 range. That is how we found our next problem. In only keeping a best solution, we were dropping the players that did not have the best solution. These results were being generated by only 120-140 players instead of all 200 players. We then corrected this problem so we no longer dropped players.

<table>
<thead>
<tr>
<th></th>
<th>Original large and increasing incorrect numbers</th>
<th>Corrected but wrong small incorrect number</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>20 iterations  Phase1</td>
<td>Phase2</td>
</tr>
<tr>
<td>100 Sums</td>
<td>9367.27</td>
<td>10307.11</td>
</tr>
<tr>
<td>Average over 100</td>
<td>93.67</td>
<td>103.07</td>
</tr>
<tr>
<td>Minimum out of 100</td>
<td>90.03</td>
<td>92</td>
</tr>
</tbody>
</table>

Table 2 Incorrect RT Iterative Minimization Results
Luis then decided that the current Phase2 was not actually helping much for the improvement of the heuristic and asked us to scrap it and come up with a new Phase2 that would improve the results of the heuristic. It was sad to have to delete several hours worth of work, but as the code wasn’t helping, it was a necessary thing to do. That brings us to our new implementation of the RT Iterative Minimization.

**RT Iterative Minimization**

In this version of the RT Iterative Minimization we are keeping Phase1 the same as before, but are implementing a new Phase2. In this new Phase2 we look at the connection time of individual players. We find the player with the largest connection time ($P_{CT}$), no matter if it is a player connected directly to the main server, is a secondary server with at least one other player connected through it, or is a player connected to the main server through a secondary server. If that $P_{CT}$ is a player connected directly to main server, we unmap him and use Min-Min RT version one to remap it. If that $P_{CT}$ is a secondary server, we unmap all players connected through it, and unmap the $P_{CT}$, and remap them using version one of Min-Min RT. If that $P_{CT}$ is a player connected to the main server through a secondary server, we unmapp it and use version one of Min-Min RT to remap it. In all of these cases we always keep the best solution, whether it be the original mapping or the new mapping. We repeat this for 1000 iterations or until no improvement. With this type of heuristic we are getting fairly decent results. Increasing the number of iterations we run the heuristic for gets even better results typically. Results for the best solution in 10, 20, and 30 iterations are listed in Table 3 along with the summation of all 100 data files, and the average over all the data files.
With these results we are also thinking about adding a Phase3 that will find $\alpha$, with $\alpha$ being defined as the secondary server with the largest computation plus communication time. Once we find this $\alpha$ we then try and swap all players mapped through a secondary server into the location of the $\alpha$ secondary server and see if one of them will give a better $RT_{\text{max}}$ time. Again, we only keep the best solution, and repeat this $\alpha$ selection for 1000 iterations or until no improvement.

**Chapter IV – Conclusion and Future Work**

The research done for this project has resulted in both participants achieving an in-depth understanding of Heterogeneous Computer and resource allocation. Neither of us had any prior exposure to Parallel and Distributed Computing and all material was new to us. Apart from being an incredibly complicated learning experience, this project also proved to be exciting and rewarding in the end. Both the Greedy Two-Phase Algorithm and the Genitor heuristics were successfully studied, coded, and implemented in a way that yielded usable results to our initial problem concerning MMOGs and VWEs.
Many obstacles arose during the course of this first semester and each had to be met with detailed analysis, creativity, and correct implementation of fixes. Both heuristics were plagued by problems during their creation and testing and had to be either redone completely or modified extensively. Debugging the C++ programs took up the majority of the time and constant testing was always being performed. However, both heuristics are working and providing satisfactory results.

Although the heuristics are working, there is still much work to be done. Each project can be tweaked and modified to improve it. For example, the Genitor has many input parameters that can still be altered to provide better results. Within the code of both programs, there are many improvements that can be done. These heuristic implementations are only recently successfully working and still in the first stages of their optimization.

Apart from these two heuristics, there are many other heuristics that can be used to solve this resource allocation problem. These two were chosen because they were not only a good starting point into the world of Heterogeneous Computing, but also historically yielded good solutions to similar problems. There are many other heuristics that can be implemented and tested, such as Tabu Search, RT Iterative Minimization, A*, and Ant Colony Optimization. We will probably test some of these algorithms next semester and compare them to results from the Genitor and Min-Min.

Another possibility for future work involves changing the problem into a dynamic environment where players can join, play, and leave whenever they want. The current programs, since they are static resource allocation heuristics, do not account for dynamic changes during game play. This would greatly increase the complexity of the system but
would perhaps more accurately simulate real-world applications. No matter where we choose to go with improvements or new heuristics, there are many possibilities for how to continue this project and we look forward to the challenges they will bring.
References


Bibliography


Appendix A - Abbreviations

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MMOG</td>
<td>Massive Multiplayer Online Game</td>
</tr>
<tr>
<td>DIM</td>
<td>Dual Iterative Minimization</td>
</tr>
<tr>
<td>GA</td>
<td>Genetic Algorithm</td>
</tr>
<tr>
<td>MS</td>
<td>Main Server</td>
</tr>
<tr>
<td>SS</td>
<td>Secondary Server</td>
</tr>
<tr>
<td>$RT_{\text{max}}$</td>
<td>Round Trip Time Max</td>
</tr>
<tr>
<td>RT</td>
<td>Round Trip Time</td>
</tr>
</tbody>
</table>

Appendix B - Budget

As of this report we have not used any of the allotted budget. We still have the full $200 dollar amount for the year. We also have no companies sponsoring our project, so we do not have additional funding that way.
Appendix C - Performance Metric

Performance Metric

Let $RT_x$ represent the Round Trip time (RT) of a packet sent by the computer of user $x$ ($U_x$) to the MS (possibly through an SS) and the response time returning to $U_x$. Let $Comm(A, B)$ be the communication time between node $A$ and node $B$. The equation used to calculate $RT_x$ if $U_x$ is connected directly to the MS is:

$$RT_x = Comm(U_x, MS) + Comp_{MS} + Comm(MS, U_x);$$
$$RT_x = 2 \cdot Comm(U_x, MS) + Comp_{MS}.$$

If a user is connected to an SS, then the equation is:

$$RT_x = Comm(U_x, SS_\alpha) + Comp_{\alpha} + Comm(SS_\alpha, MS) + Comp_{MS} + Comm(MS, SS_\alpha) + Comm(SS_\alpha, U_x);$$
$$RT_x = 2 \cdot Comm(U_x, SS_\alpha) + 2 \cdot Comm(SS_\alpha, MS) + Comp_{\alpha} + Comp_{MS}.$$

If $U_x$ is $SS_\alpha$, used with $Comm(U_x, SS_\alpha) = Comm(SS_\alpha, U_x) = 0$.

The performance metric of a resource allocation in this environment is:

$$RT_{max} = \max_{U_x} (RT_x).$$

The objective for the heuristics is to minimize $RT_{max}$. The value of $RT_{max}$ will be influenced by three elements: the number of SSs, which users’ computers are converted to SSs, and the assignment of non-SS users to an SS or to the MS.

Appendix D - Lower Bound

Lower Bound

The primary purpose of deriving a mathematical lower bound was to evaluate the experimental results of our proposed heuristics. The bound has two components. The first component finds the minimum possible computation time of the MS and SSs (by performing an exhaustive search of all possible computation times). The second component is the minimum time a user can take to communicate to the MS (by finding the maximum among all of the minimum communication times each player can have to reach the MS). The variables used in this equation are $n$, number of players that are not connected to the MS ($n_{non}$ plus $n_{main}$); and $\mu_{min}$, minimum $\mu$ over all users. The lower bound (LB) is given as:

$$LB = \min_{1 \leq n \leq 200} \left( \min_{0 \leq n_{\alpha} \leq n} \left( \mu_{min} \cdot \left[ 200 - n \cdot n_{\alpha} \right] + c \cdot (200 - n) + b \cdot n^2 \right) \right)$$

$$+ \max_{U_x \in \text{all users}} \left( \min_{U_y \in \text{all users}} \left( 2 \cdot Comm(U_x, U_y) + 2 \cdot Comm(U_y, MS) \right) \right)$$

28
Appendix E – RT Iterative Minimization

**Phase 1** (Adjust the number of secondary servers)

1) Randomly pick a number $k$ between 0 to 200 as the initial number of SSs.
2) Randomly assign $k$ players as SSs.
3) Find $RT_{\text{max}}$ using Min-Min RT and determine the player ($P_{\text{max}}$) that has the $RT_{\text{max}}$.
4) If $P_{\text{max}}$ is:
   a) A SS (or player connected directly to the MS), then unassign it and reassign it to the SS that gives it the minimum RT.
   b) A player connected to a SS, then unassign it and reconsider reassigning it to the SS or MS that gives it the minimum RT.
5) Repeat 3) and 4) until there is no improvement, number of iterations is 1000

**Phase 2** (Determine which players are the secondary servers)

6) Using the list of SS from Phase 1 we find the $P_{\text{max}}$.
7) If $P_{\text{max}}$ is:
   a) A SS (or player connected directly to the MS), then consider which player ($P_{SS}$) could give it a smaller RT. If $P_{SS}$ is already a SS then connect $P_{\text{max}}$ to $P_{SS}$; otherwise, consider swapping $P_{SS}$ with all other SSs (using Min-Min RT to determine $RT_{\text{max}}$).
   b) A player connected to a SS ($SS_{s}$), then consider swapping all other non-SS players with $SS_{s}$, keeping the mapping with the smallest $RT_{\text{max}}$.
8) Repeat 7) until there is no improvement or number of iterations is greater than 1000

**Acknowledgements**

We would like to acknowledge the help received from Professor Siegel, Luis Briceño, and Samee Khan on this project. Kody would also like to acknowledge help received from Timothy Renner in coding her heuristic.