Centralized Matchmaking for Minimal Agents

K. Bertels
Delft U. of Technology

N. Panchanathan
UpsilonResearch

S. Vassiliadis
Delft U. of Technology

B. Pour Ebrahimi
Delft U. of Technology

Abstract

In this paper we propose a simple though efficient mechanism to enable distributed processing among nodes to communicate with each other and to dynamically rebalance the workload among them. We discuss a centralized matching mechanism and investigate different matching functions, varying in the amount of information that is taken into account. The main contribution of this paper is that the simplest mechanism, called FirstMatch, results in the fastest matchmaking achieving matching rates of 99%. We also show that this time efficiency does not have a cost in terms of efficient task allocation and resource utilization.

Keywords: simulation, minimal agents, task allocation, market, match making.

1. Introduction and Related Research

If we want computers to solve large and complex problems, one can either write large and complicated programs or, as an alternative, use a multi-agent system approach [12]. The basic idea is to have a collection of agents where each agent has to solve a small subproblem. The agents have to figure out how to do this and what resources they need. We are interested in the simplest agent structure or mechanism that allows such distributed task processing. We call such agents **minimal agents**. The idea of a minimal agent is a first step towards using them as basic hardware blocks for large, distributed systems. Numerous issues are involved such as routing, synchronization, resource allocation and rebalancing.

Starting point for building large scale distributed systems is their ability to process large quantities of data. From a computer engineering perspective, we are looking for mechanisms that can be implemented at a very low, hardware level. This implies that sophisticated reasoning mechanisms cannot be considered as they consume too many processor cycles. What we are actually looking for is a mechanism which has the same order of complexity as a multiplexor which selects from several inputs what output will be generated. Such a mechanism should allow to process large quantities of data at very high speeds. As shown in Figure 1), there is a trade off between the degree of intelligence of a component and its ability to rapidly process large amounts of data. Data is viewed here merely as a series of bytes that need to be treated in some way and the more bytes that need to be processed in any way (intelligent or not), the more processor cycle time is required. When targeting large scale distributed systems, ranging from several thousands to several millions components, it is therefore important to develop mechanisms that scale up easily and still provide high performance processing.

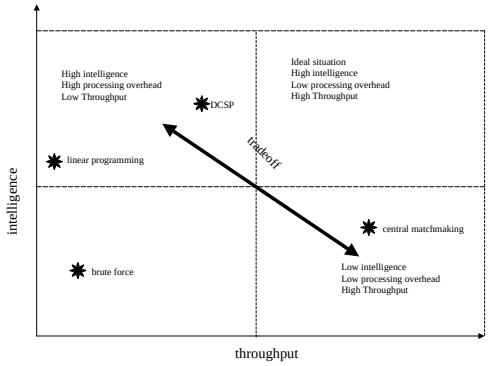

Figure 1. Tradeoff between intelligence and throughput

As far as resource allocation is concerned, several policies have been proposed that solve synchronization problems that can arise during resource allocation and scheduling. Examples of such policies are ”passing the baton” [3], the shortest job allocation [8] to name but some of the best known ones. These policies can be applied when two or

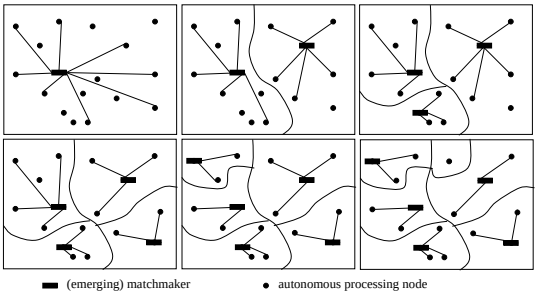

Figure 2. Robustness of the system

more processes compete for scarce resources. This arises when for instance several writes to a database need to occur or when one process needs to read information from a database while another is attempting to write to it. Other situations are competing for processor time, buffers, etc. Generalized descriptions of these problems are the Dining Philosophers problem, Readers and Writers and Consumers and Producers. They are all different instances of the same problem where competing processes are looking for resources.[4]

An alternative approach is to use the Distributed Constraint Satisfaction Paradigm (DCSP) which has been proposed as a way to model and reason about the interactions between agents' local decisions [16]. In DCSP each agent controls the value of a variable and agents must coordinate their choice of values so that a global objective function is satisfied. The global objective function is modelled as a set of constraints where each agent is only assumed to have knowledge of the constraints in which its variable is involved. Determining appropriate values for the decision variables can be done through either refinement or local search. The idea of refinement search involves an iterative reduction in the problem space of each decision variable, resulting in one admissible value. At each refinement step, constraint propagation techniques are applied. Whenever refinement turns out to be inapplicable in a later stage of the search process, backtracking is used to explore other possibilities. This refinement process is guided by either domain dependent or domain independent heuristics. The major disadvantage of this approach is the lack of a global perspective. An alternative therefore is to introduce local search algorithms. Local search aims to find a value for the variables given the set of constraints by repeatedly revising the concrete value assignment. This revision is typically guided by what is called a measure of inconsistency. The more global perspective is obtained by identifying those constraints that cause an inconsistency. That information is represented by so called inconsistency measures and is used to generate the next value assignment. This value assignment can be done using linear programming [18] or by linking demons to the tasks that need to be executed [2]. The main disadvantage of these approaches is the computational overhead involved, and therefore their limited scalability, in both computing the inconsistency measures and the subsequent local search. For instance, as described in [2], for each pair of tasks there are two demons recording the number of iterations a constraint was inconsistent. When a recommendation is to be made,the highest demons' count is used to propose a new value for the next search round. It is found that this particular method may result in qualitative good solutions but the computation overhead is considered too heavy.

The problem we are looking at is similar albeit different : we do not have competing processes but rather processes that are looking for collaboration to execute some of the tasks in their local queue nor do we want to find optimal solutions given a set of constraints. The focus of this paper is rather on low level and simple mechanisms that can be scalable and result in as little overhead as possible. More specifically, we focus on resource allocation in the case where some resources are lying idle and could be linked with overloaded nodes in a network. The goal is to re-balance the workload. Such situations may be caused by local congestion or a node failure, thereby inducing the necessity to re-allocate tasks to other nodes. Task reallocation means that the snapshots of the current work load is available and can be used to perform efficient re-allocation. The question arises as to how this allocation should be done by the agents. Whenever an agent needs additional resources to perform its assigned task, the agent needs to locate the available resources and make use of it. This assumes some kind of matchmaking as a precursor to possibly collaboration. There are basically three approaches to accomplish this. First, one can use middle agents providing some kind of central directory [11] [6] [5]. An alternative approach is to use market bidding mechanisms like in [19] [7][10] where bids and offers are broadcasted to all agents in the market. A third possibility is to allow peer-to-peer communications inducing lower communication cost but using only local information [13] [17]. In addition, we want the mechanism to be robust. Whenever for some reason (see 2), some routers or bridges in the network go down, the same matchmaking capability needs to be restored as soon as possible for the different segments in the network. The easiest way to accomplish this is to redefine an existing node as the matchmaker. This mechanism would allow any number of network segments and still have some matchmaking capability.

This paper extends the work on matchmaking of min-

imal agents with and without a facilitator as described in [13],[14] and [15]. It investigated matchmaking with and without a central facilitator and indicates that even when only local information is available, but using some kind of auction, the matchmaking achieves a 93% success rate for populations of up to 32K agents. However, finding a good match might take up to 300 bidding rounds before actually finding a match. The main advantage of such approach is the reduced communication overhead because no broadcasting to all agents is required. However, the time required to find a match in the local neighborhood increases substantially. Another reason to look at centralized mechanisms is that in many situations of extreme distributed computing, such as spray computers [9] or dust computing [1], there is still a need at some point for a centralized mechanism. This can be for pure routing purposes (e.g. a gateway) or for match making as discussed in this paper. We assume the following about the nodes in the network :

- They are capable of determining whether they require collaboration with other nodes given the current state of their queue. This implies some scheduling and planning knowledge on their behalf in order to determine for what tasks they require a certain amount of additional processing power. The question can be raised whether the nodes can be termed autonomous agents as the matchmaking does not give any degrees of freedom. However, as stated, the autonomy lies in determining whether or not they need collaboration.
- Once the agent determines to submit a request, that particular request is handled by the centralized matchmaker. This is similar to the communication channel manager in the JADE environment where all the communications are centrally channelled.
- There is no need for our processing nodes to hide their preferences. Each node simply has tasks to process and may request assistance in that. There is no benefit in lying

The main contributions of this research are :

- The simplest matchmaking function is the most efficient one both in terms of resource usage, task execution and matching time.
- more computationally expensive functions only yield a marginal improvement in the order of 1% as far as the resource usage and task execution are concerned.
- Centralized matchmaking is scalable in the range of (studied) 5 to 50K agents interacting on the market. [1]
- Multiple matchmakers can be introduced to reduce substantially the communication overhead. From our findings we can prudently advance the suggestion that a population size should preferably not be lower than 10K or larger than 20K. Beyond a size of 10K, performance drops considerably and there is no substantial increase beyond a size of 20K.

The paper is structured as follows. After this section, we present the match maker model and illustrate how the nodes exchange information with the match maker and propose 3 different matchmaking functions. We then discuss the simulation model that was built to assess the efficiency of the different functions and discuss the underlying assumptions. We then present the empirical results and conclude by discussing further research directions.

2. The Central MatchMaker

In this section,we present the implemented model and justify the choices we made in order to obtain the simplest possible algorithm.

2.1. Justification

Agent Properties In [14], agents are actively bidding on an auction. This implies that they react to bids and offers made by other agents. On the basis of past experiences, the agents learn to optimize their bidding strategies and requires a learning algorithm. In the context of minimal agents, we do not want to include such bidding or learning mechanisms as it creates too much overhead. In [13], peer to peer local interaction was induced by allowing agents to interact locally with neighboring agents. These interaction patterns can be changed whenever no match is occurring. This induces quite some overhead and multiple searches before a good match occurs. Our agent is simpler as its only communication means (as far as matchmaking is concerned) is directly with the matchmaker.

Bidding Rounds and Messages Transmitted An important aspect is the number of messages that need to be transmitted over the network in order to reach a particular allocation state. Our approach requires that we need only a minimum of 2 messages per agent where tasks are requested and resources are allocated. Similar to [14], this implies that the time complexity of our approach is O(N). By introducing multiple matchmakers or auctions, this message distribution is reduced to O(log(N)). In [17], a related albeit different problem of agent collaboration is investigated. Agents have to produce an optimal solution given some global objective. The general distributed constraint optimization assumes that each agent is sending information to all of its linked descendants. Such an approach is known to have exponential time($O(2^N)$) and is only feasible

1 It should be noted that all agent population sizes mentioned in this paper should be multiplied by 2 as it involves pairs consumers and producers.

for a very low number of agents. In conclusion, we wanted to avoid bidding or any other form of message passing other than submitting a request (bid or offer) to the matchmaker. This avoids additional message broadcasting that need to be sent to all other agents informing them of the bid made by any of the other agents.

Match Making A third aspect of our approach involves the actual match making process. In [13][15], localized random search is used to find a match. In [14], sorting is required for matchmaking as the highest bid is matched to the lowest offer, etc. Even though algorithms such as Quicksort are efficient, they nevertheless represent an additional computational step in the matchmaking process. We avoid the use of such expensive mechanisms given the large population sizes we are interested in.

2.2. The Model

The goal of this experiment is to evaluate different ways of redistributing tasks to different processing nodes, given some constraints. We consider a grid like environment where some nodes might fall idle whereas others are still overloaded. The latter category wants to delegate some of the tasks in their local queue to other nodes in order to decrease the overall computing time. We assume 2 categories of agents, Producers and Consumers. Producers have processing power to sell and Consumers are looking for additional processing power to execute the tasks in their queue. The goal is to facilitate the match making process in such a way that the maximum number of tasks is executed, given the available resources. This implies that a Consumer has to find a Producer that can provide sufficient processing time to execute the task. The matchmaking mechanism involves a centralized mechanism (similar to but different from a market) that will receive from producers the resources offered and from the consumers the requested resources. No subsequent bidding or negotiation is then required. We make the following assumptions :

- We assume that each agent is directly and with the same average latency connected to the matchmaker and that the cost for this connection is the same for each agent. This simplifies the problem and even allows us to exclude it from the analysis.
- Tasks are atomic by nature and cannot be divided. Each task has a particular complexity which is represented by an integer value. This value indicates the amount of resources required in order to be executed. Resources are represented in a similar way. Whether or not a particular task can be executed by an available resource is then simply determined by comparing these integer values.
- One consumer can match with only one producer who has sufficient resources to execute the task. This means that we currently do not look at collaborative issues.
- We generate randomly the initial task and resource allocation from a uniform distribution;
- We are currently using a simulation model in which the agent requests are treated in a pure sequential way. In a realistic setting, the match making would occur in an asynchronous way where the different offers and bids are treated as they are submitted.
- We assume that there are an equal number of Consumers and Producers. This is not restrictive as the random generation of tasks and resources can result in a zero value which is similar to taking the agent out of the population.
- When referring to 'population', we mean the number of agents that use the matchmaker to buy or sell processing time. They represent only that fraction of the actual population that is looking for a match.
- The terms buying and selling may be a bit misleading as no actual transaction requiring some kind of payment is involved. This implies the following : consumers and producers do not need to propose a price and no price formation mechanism, such as a general equilibrium price, need be computed. Either requests match and that match is communicated to either party involved, or there is no match.

In our environment, we have N agents: $A = \{a_1, \ldots, a_n\}$. Some of these agents, called Consumers, have tasks to perform $T_a = \{t_1, \ldots, t_k\}$ for which they are looking for additional resources and others, called Producers, have resources to sell, $R_a = \{r_1, \ldots, r_k\}$. There is a matchmaker to which both consumers and producers announce their requests. Consumers will send to the matchmaker the number of tasks they want to delegate and the producer will announce in the same way how much processing time it has available. The matchmaker then uses that information to match consumer requests to producer bids. The basic matching goes as follows : $f : C \times P \rightarrow [0, 1]$ with

$$f(c_i, p_j) = \begin{cases} 1, & \text{if } (c_i, p_j) \text{ is a matching pair} \\ 0, & \text{otherwise.} \end{cases}$$

We define 3 matching functions that vary in complexity and information used for matching. Evidently, one can define any number of matching functions taking for instance the Quality of Service into account or any other relevant factor. The choice of our functions is justified in the sense that we want to compare an extremely simple one (FirstMatch) with functions that are more complicated as they take more information into account to compute a match. The chosen functions are defined as follows :

Producers	P1	P2	P3	P4	P5
	5	2	3	2	1
Consumers	C1	C2	C3	C4	C5
	1	2	2	1	1

Table 1. FirstMatch Illustration

- **FirstMatch** : for each consumer request, the matchmaker matches the consumer to the first producer that has enough resources to execute the consumer's request;
- **MinDifference** : the consumer is matched with that producer that has enough resources but also that yields the lowest difference between the requested task and available resources. This approach attempts to minimize the unused resources. We emphasize that we do not sort the producers or consumers data;
- **MinDistance** : the consumer is matched not only when enough resources are available but tries to minimize the distance between the two paired nodes. This is to minimize as much as possible the distance that has to be travelled between the two paired nodes. As each node has an (x,y)-coordinate, we compute the euclidean distance between two nodes. Similar to MinDifference, the producers or consumers are not sorted using their (x,y)-coordinates.

To illustrate the approach, consider the following example, as given in Table 1, for the FirstMatch function. Each of the 5 producers in our population is given a number of resources and the same is done for the 5 consumers. When adopting the simplest matching function, Consumer 1 will be matched to Producer 1, Consumer 2 to Producer 2, Consumer 3 to Producer 3 and Consumers 4 and 5 to Producers 4 and 5. When adopting the MinDifference matching function, the pairs are (C1,P5),(C2,P2),(C3,P4),(C4,P3) and (C5,P5).

3. Empirical Results

For each of the matching functions, we compute the following statistics :

- **Task Execution Efficiency :** the percentage of tasks that could be allocated to available resources;
- **Resource usage Efficiency :** the percentage of the available resources that are used by the allocated tasks;
- **Matching Time :** we compute the relative matching time for a consumer to find an appropriate resource. In order to account for the varying population sizes, we divide the matching time by the number of agents in order to find the matching time for one agent. In this way, we can isolate the influence of the sequential nature of the match making time. We will also look at the variance of the matching time. It can be considered a measure of uncertainty where higher variance means a higher variability in finding a match time. This is an important issue in later hardware implementation as it may introduce a number of latencies in the communication processes.

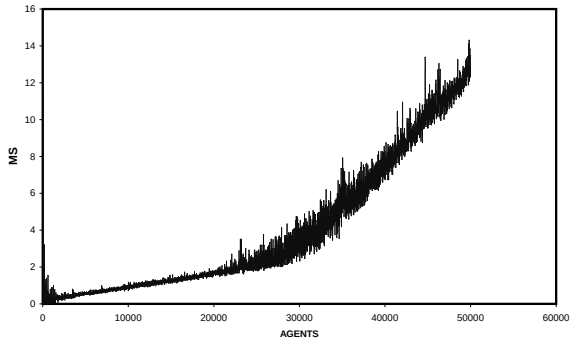

Figure 3. FirstMatch : Matching Time

3.1. FirstMatch experiment

We first look at the FirstMatch function and compute for a population ranging from 5 to 50.000 agents how many matches occurred. From Figure 3, we can see that the matching time is proportional with the number of agents. However, the increase is very slow as it ranges from 0.4 milliseconds to 1.453 for agent populations up to 20K. When going beyond this boundary, the matching time increases more or less proportional. As far as the allocation efficiency is concerned, we can also observe that for relatively small populations (less than 5K), we do not get very high allocation rates. The average, given in Table 2, is around 90%. Even though the allocation never reaches 100%, as the population size increases, we obtain percentages of around 97-98%. The upper bound is 99% for population sizes larger than 30K. However, the variance, which could be seen as a measure of uncertainty, increases also. This can be seen from Table 2 and from Figure 5). The plot of the first difference of the matching time (see Figure 5) graphically shows the increase in variance of the matching time for population sizes larger than 20K. As far as the resource utilisation is concerned, we see that there is a similar evolution as for the task allocation. It increases with the population size but then saturates close to 100% for sizes larger than 20K.

Pop.Size	FirstMatch				MinDifference				MinDistance			
	Tasks	Res.	Time	σ	Tasks	Res.	Time	σ	Tasks	Res.	Time	σ
5K	90%	98%	0.403 ms	0.3131	91%	97%	0.795 ms	0.2931	85%	98%	1.1818 ms	0.5587
5-10K	94%	99%	0.728 ms	0.0137	94%	98%	2.679 ms	0.3361	90%	98%	3.9100 ms	0.6797
10-15K	97%	99%	1.092 ms	0.0159	95%	98%	4.725 ms	0.4694	92%	99%	6.7801 ms	0.7467
15-20K	97%	99%	1.453 ms	0.0197	96%	98%	7.088 ms	0.7512	93%	99%	9.7213 ms	0.9663
20-25K	98%	99%	1.922 ms	0.0812								
25-30K	98%	99%	2.678 ms	0.3262								
30-35K	99%	99%	4.241 ms	0.6853								
35-40K	99%	99%	6.374 ms	0.6248								
40-45K	99%	99%	8.875 ms	0.9153								
45-50K	99%	99%	11.427 ms	0.7063								

Table 2. Average Matching time and Task Allocation for varying Population Sizes

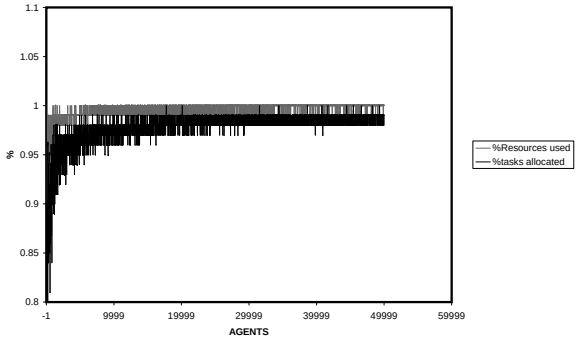

Figure 4. FirstMatch : Task Allocation and Used Resources

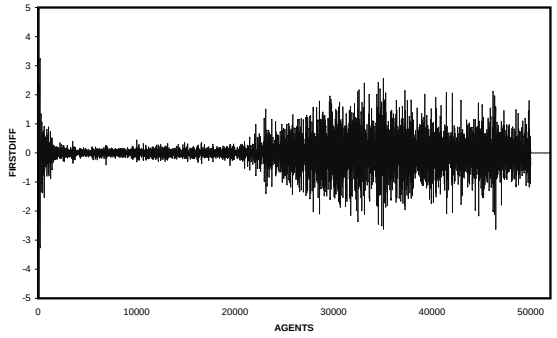

Figure 5. FirstMatch : First difference of Matching Time

3.2. MinDifference experiment

As can be seen from Figure 6, the first thing to notice about this matching function is the substantial difference in time required to produce a match. This is evidently explained by the fact that, for each consumer, the function iterates over all producers and picks the one that generates the smallest unused resources. The matching time is proportional to the population size and evolves in a relatively stable way as can be seen from the graph of the first difference even though there are population sizes where a higher variance is observed. As far as the task allocation efficiency is concerned and on the basis of Figure 7 and Table 2, we can make a similar observation as for FirstMatch. For populations smaller than 5K, the matching is not very efficient with allocation percentages ranging around 90%. This allocation percentage rises to reach 96% for a 20K population size. Increasing the population from 15K to 20K, only increases the efficiency by 1% but requiring 50% more time to produce the match. Because of the very low improvement for either task or resources, we do not simulate beyond the 20K boundary. Shown in the same figure, we also plotted its first difference. From that graph and also from Table 2, we can again observe that as the population size increases, the variance of the matching time goes up, introducing more uncertainty in the matching process. As far as the resource usage is concerned, and also shown in Figure 7, the same conclusion as for FirstMatch holds : it improves as population size increases and then saturates at 99%.

3.3. MinDistance experiment

The last matching function that was used is the minimal distance between two agents. The minimum distance

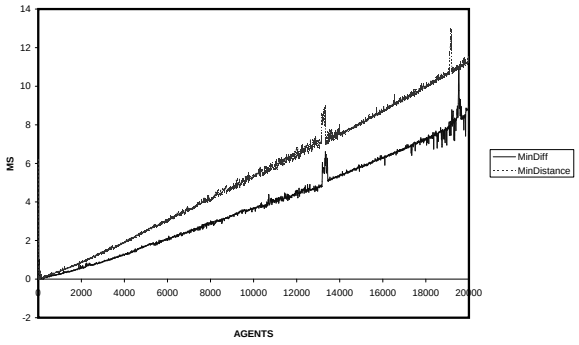

Figure 6. MinDifference : Matching Time

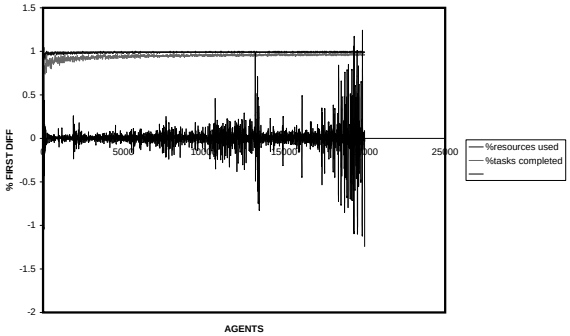

Figure 7. MinDifference : Task Allocation, Used Resources and first difference of Matching Time

function computes the euclidian distance between each consumer and a given set of producers which is more expensive in terms of computing cycles than MinDifference and Firstmatch. We can clearly observe from figure 6 that the matching time of MinDifference is less than MinDistance. This is a direct consequence of the structure of the MinDistance function. Looking at the MinDistance efficiency plotted in Figure 8, a similar observation as for the other 2 functions can be made : the task execution efficiency does not increase substantially as we approach the 20k boundary. The first difference plot shows as similar behavior as the MinDifference and the variance has a similar value. However,as can be seen from Table 2, MinDistance systematically has a higher variance than MinDifference. As far as the resource utilisation

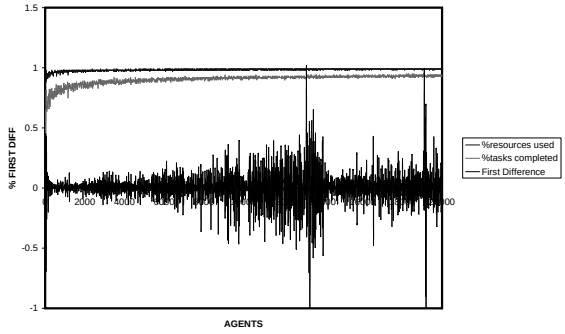

Figure 8. MinDistance : Task Allocation, Used Resources and first difference of Matching Time

is concerned, the same observations as above hold.

Putting the results together and comparing the task allocation efficiency for the 3 functions, as plotted in Figure 9, we can observe that MinDifference is the most efficient approach.[2] However, it is only marginally better than FirstMatch. As we can see, these differences center around 0 indicating that they are equally efficient. Combined with the fact that FirstMatch has much lower matching times, we can conclude that the simplest function is the most powerful one.

4. Size does matter

One possible extension which was mentioned before is to introduce multiple matchmakers. This would allow us to bring down the number of messages to O(log N). The question arises then how many of such matchmakers we need for a given population size. An answer to this question can be found in Figure 4. [3] We see that the matching efficiency rapidly goes up as the population size increases. This implies that for low population sizes i.e less than 6000 we do not obtain very high allocation rates(less than 80%). As the population size increases up to 20000 it reaches 98%. We then see that for larger population sizes there is no substantial increase in allocation efficiency. It remains constant for values larger than 30000. As mentioned before, the al-

2 As it is difficult to distinguish between the allocation curves of the three functions, we plotted only the FirstMatch task allocation efficiency and computed then the difference between their analogs for the two other functions.

3 As the FirstMatch algorithm has the best performance, we restrict our discussion to it.

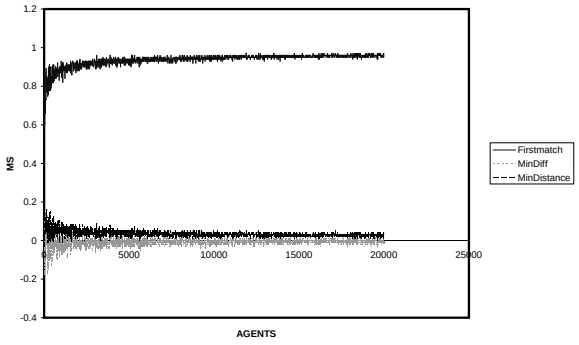

Figure 9. Task Allocation Efficiency for the 3 functions

location never reaches 100% and we did not simulate beyond the 50000 boundary because no appreciable changes are observed. In conclusion, population sizes of 15K to 20K paired agents guarantees sufficiently high matching rates. Going below the 10K boundary will generate lower matching rates and the going beyond the 20K boundary will increase the matching time but not the allocation efficiency.

5. Conclusion and Further Research

We have studied the resource allocation and match making in single clusters systems by giving a simple approach for matchmaking. We introduced a centralized matching mechanism that requires very little information to produce a good matching. In addition, the number of messages required to be broadcasted over the network is limited and in the order of O(N). When introducing multiple matchmakers, this can even be reduced to O(log N). We furthermore evaluated more complicated matching functions, as they incorporated more information for the actual matchmaking, by looking at task allocation efficiency, resource usage efficiency and matching time. The main findings are (i) FirstMatch is the simplest and the most efficient among all the three, (ii) the MinDifference function is better than MinDistance as far as the matching time is concerned taking into consideration resource allocation and finally, (iii) the centralized matching mechanism easily scales up to 20K. For FirstMatch this even goes to 50K. (iv) It seems that there exists some kind of population size beyond or below which either no improvement can be generated or the matching efficiency goes down respectively. This allows the introduction of multiple matchmakers, reducing the number of messages to O(log N).

Issues which remain unsolved are how efficient this approach is when resources and tasks are unevenly distributed. It might be that more information intensive approaches will outperform the FirstMatch approach. It also remains to be seen if this approach remains feasible when introducing asynchronous, rather than batch-like, sequential, matchmaking.

References

[1] K. Gabriel A. Berlin. Distributed mems: new challenges for computation. *IEEE Computing in Science and Engineering*, 4(1):12–16, Jan-March 1997.

[2] S. Smith A. Nareyek and C. Ohler. Integrating local-search advice into refinement search (or not). In *Proceedings of the CP 2003 Third International Workshop on Cooperative Solvers in Constraint Programming*, pages 29–43, 2003.

[3] G. Andrews. A method for solving synchronization problems. *Science of Computer Programming*, 13(4):1–21, Dec. 1989.

[4] G. Andrews. *Foundations of Multithreaded, Parallel and Distributed Programming*. Addison-Wesley, Reading Massachusetts, 2000.

[5] Epema D. Bucher A. Local versus global queues with processor co-allocation in multicluster systems. In *Eighth Workshop on Job Scheduling Strategies for Parallel Processing (in conjunction with HPDC-11)*, pages 184–204. 2002.

[6] Karonis N. Kesselman C Martin S. Smith W. Tuecke S. Czajkowski K., Foster I. A resource management architecture for metacomputing systems. In *The 4th workshop on Job Scheduling Strategies for Parallel Processing*, pages 62–82. 1998.

[7] Ch. Weinhardt D. Veit, J.P. Muller. Multidimensional matchmaking for electronic markets. *Journal of Applied Artificial Intelligence*, 16(9-10):853–869, 2002.

[8] E. Dijkstra. A tutorial on the split binary semaphore. In *EWD 703*, pages 1–10. Technische Universiteit Eindhoven, 1979.

[9] M. Mamei R. Tolksdorf F. Zambonelli, M.P. Gleizes. Spray computers: Frontiers of self-organization. In *1st International Conference on Autonomic Computing*, page poster, 2004.

[10] Klusch M. Widoff S. K. Sycara, Lu J. Matchmaking among heterogeneous agents on the internet. In *Proceedings. AAAI Spring Symposium on Intelligent Agents in Cyberspace*. 1999.

[11] Harada L. Kuokka, D. Matchmaking for information agents. In *Proceedings of the 14th International Joint Conference on Artificial Intelligence*, pages 672–678, 1995.

[12] Jennings N. On agent-based software engineering. *Artificial Intelligence*, 117:277–296, 2000.

[13] E. Ogston and S. Vassiliadis. Matchmaking among minimal agents without a facilitator. In *Proceedings. 5th International Conference on Autonomous Agents*, pages 608–615. May 2001.

[14] E. Ogston and S. Vassiliadis. A peer-to-peer agent auction. In *Proceedings of the first international joint conference on Autonomous agents and multiagent systems Part I*, pages 151–159, July 2002.

[15] E. Ogston and S. Vassiliadis. Unstructured agent matchmaking: experiments in timing and fuzzy matching. In *Proceedings of the 2002 ACM symposium on applied computing*, pages 300–306, March 2002.

[16] M. Tambe M.Yokoo P.Modi, W. Shen. An asynchronous complete method for distributed constraint optimization. In *Proc. International Conference on Autonomous Agents*.

[17] Milind Tambe Makoto Yokoo Pragnesh Jay Modi, Wei-Min Shen. Asynchronous complete method for general distributed constraint optimization. In *Proceedings of the first international joint conference on Autonomous agents and multiagent systems Part I*, July 2002.

[18] M. Sakkout H., Wallace. Probe backtrack search for minimal perturbation in dynamic scheduling. *Constraints*, 5(4):359–388, 2000.

[19] Jennings N.R. Vulkan, N. Efficient mechanisms for the supply of services in multi-agent environments. *Journal of Decision Support Systems*, 28(1-2):5–19, 2000.