In Proc. 31st Hawaiian International Conference on System Sciences, VOL VII - Software Technology Track, pages 4 - 13, IEEE Computer Society, 1998.
Winner of the best paper award of the Software Technology Track

Managing Large Scale Computational Markets

Arne Andersson
Department of Computer Science
Lund University
221 00 Lund, Sweden
Arne.Andersson@dna.lth.se
http://www.dna.lth.se/home/Arne_Andersson/

Fredrik Ygge
EnerSearch and
IDE at University of Ronneby
372 25 Ronneby, Sweden
Fredrik.Ygge@enersearch.se
http://www.enersearch.se/˜ygge

Abstract

General equilibrium theory has been proposed for resource allocation in computational markets. The basic procedure is that agents submit bids and that a resource (re)allocation is performed when a set of prices (one for each commodity) is found such that supply meets demand for each commodity. For successful implementation of large markets based on general equilibrium theory, efficient algorithms for finding the equilibrium are required.

We discuss some drawbacks of current algorithms for large scale equilibrium markets and present a novel distributed algorithm, CoTree, *which deals with the most important problems.* CoTree *is communication sparse, fast in adapting to preference changes of a few agents, have minimal requirements on local data, and is easy to implement.*

1 Introduction

In our human society, resource (re)allocations are in most cases performed through markets. This occurs on many different levels and in many different scales, from our daily grocery shopping to large trades between big companies and/or nations. The market approach to resource allocation in the human society has inspired the Multi-Agent Systems (MAS) community to construct similar concepts for MAS, where the trade is performed between computational agents on *computational markets.* Wellman [Wellman, 1993] refers to this as *market oriented programming.*

In computational markets, a common approach is to use a mechanism that obtains *general equilibrium*, as done by Cheng and Wellman [Cheng and Wellman, 1997]. This is also the market approach investigated here. General equilibrium is obtained when a set of prices (one price for each commodity) is found such that supply meets demand for each commodity and where the agents optimize their use of resource at the current price level [Mas-Colell *et al.*, 1995; Varian, 1996]. In this paper we only describe pure exchange markets, but all concepts here are also well suited for the incorporation of production.

For making market-oriented programming successful, it is of vital importance to incorporate available knowledge from economic theory. At the same time it is important to study efficient implementations. In previous work [Ygge and Akkermans, 1996] we showed how numerical analysis and mathematical optimization could be utilized to implement efficient markets with a reasonable number of agents (up to 1000), but scaling up to huge systems in highly distributed environments introduced some unanticipated difficulties. In this paper we discuss the difficulties of scaling up to large markets. We then introduce a novel algorithm, CoTree, which deals with the most important problems. Our research aims at implementing the application power load management [Ygge and Akkermans, 1996] as a computational market, but all results in this paper are general and application independent.

The paper is arranged as follows. In Section 2 we discuss some basic economic concepts and important characteristics of two general approaches to finding the equilibrium. The drawbacks of current methods in the context of large-scale distributed multi-agent systems are discussed in Section 3; in particular problems caused by too much communication. Then, in Section 4 we present our new method that aims at overcome these problems. We then demonstrate the dynamics of the method once an initial equilibrium has been found (Section 5). Some experimental data is given in Section 6. The multi-commodity case is discussed in Section 7, and, finally, Section 8 concludes.

2 Context and Problem

2.1 Basic Micro-Economic Concepts

We analyze a market with two commodities. (See Section 7 for a discussion on how our results can be extended for situations with more than two commodities.)

An agent's net demand describes how much it is willing to buy at a specific price in terms of the other commodity. We denote the net demand of agent i for commodity one by $z_i(p)$, where p is the price for commodity one in terms of commodity two. The *aggregate excess demand*, $z(p)$, is defined as the sum of the net demands of all agents, i.e.

$$z(p) = \sum_{i=1}^{n} z_i(p), \tag{1}$$

where n is the number of agents.

Then market equilibrium is given by

$$z(p) = 0. \tag{2}$$

Instead of asking an agent how much it is willing to buy or sell at a specific price, one might ask it how much it is willing to pay for an infinitesimal amount of the commodity at the current allocation (a price). That is, each agent, i, can be viewed as holding a price function $p_i(z_i)$, rather than a demand function $z_i(p)$. If the net demand is monotone and continuous, there is a bijective mapping between $p_i(z_i)$ and $z_i(p)$, i.e. $p_i(z_i)$ tells what price would lead to a certain demand. Then Eq. (2) corresponds to

$$\begin{cases} p_i(z_i) \leq p, & z_i = z_i^l \\ p_i(z_i) = p, & z_i^l < z_i < z_i^u \\ p_i(z_i) \geq p, & z_i = z_i^u \end{cases}, \tag{3}$$

where n is the number of agents and z_i^l and z_i^u are agent i's lower and upper limit of the net demand.

2.2 The Computational Problem

In this paper we investigate the standard case were the net demand is continuous and monotonically decreasing with price. If we can solve a market problem with such a demand, we can solve a separable resource allocation problem with concave objective functions [Ygge and Akkermans, 1997].

The computational problem investigated is to solve Eq. (2) in a multi-agent setting, i.e. in a setting where the information about the net demand (and/or price) is computed from local information by local agents.

For a (re)allocation to be performed the system must 1) compute the equilibrium solution and 2) notify the agents about their new allocation. Depending on the setting, the difficulty of the second step will vary significantly. In this paper we deal only with step one, as step two is too application dependent to be discussed in general terms. In some settings, the task of reallocating commodities can be completely manual, i.e. no computerized communication will take place after the equilibrium has been found. In another setting, it might be enough to announce the equilibrium price and let every agent autonomously give and take resources as declared by their bids. In yet another situation the agents may exchange digital cash for an encrypted piece of information.

2.3 Two General Approaches for Finding Equilibrium

When implementing a market mechanism for achieving general equilibrium there are basically two alternatives: either the price is used as the free parameter, and p is updated until Eq. (2) holds, or the resource is used as the free parameter and different transfers of resource from one agent to another are evaluated until Eq. (3) holds.

This classification leads to the notion of *price-* vs. *resource-oriented* approaches. The conditions in Eq. (2) and Eq. (3), together with the corresponding market mechanism, are different ways of saying the same thing. In the first case the method itself guarantees that the price of each commodity is equal for every agent, and in the second case the allocation is always feasible, since resource is only transferred between agents.

2.4 Characteristics of the Different Approaches

Above, we argued that the equilibrium conditions for the price-oriented approach and the resource oriented approach, Eq. (2) and Eq. (3), are equivalent. This is true if the conditions are *exactly* fulfilled. However, in practice, for the algorithms to converge in reasonable time, termination conditions for the algorithms will be $|z_j(p)| \leq \epsilon$ and $|p_{ij}(x_i) - p_{mj}(x_m)| \leq \epsilon$ respectively, where ϵ is a small positive constant. In the resource-oriented case, this means that the allocation is *not perfectly fair*, i.e. some agents will pay slightly less than they would have done on a perfect market, while others will pay slightly more. In the price-oriented case, on the other hand, the allocation is *not perfectly feasible*. This is an important difference; in the latter case it can be impossible to allocate the computed amount to the involved agents, due to physical constraints. In practice this need not be a problem, but if ϵ is chosen to be relatively large, e.g. for performance reasons, and if many consecutive trades are performed, this should be considered.

Another important difference is the input to the two approaches. In the price-oriented case the input is the demand function and in the resource-oriented case the input is the price function. The price function is more closely related to the utility function (it is merely the first derivatives of the utility function with respect to the first commodity divided by the first derivatives of the utility function with respect to the second commodity) which is the primary property in micro-economics (e.g., [Varian, 1996]) while the demand has to be derived from function inversion.

3 Drawbacks of Current Algorithms

Even though some of the available algorithms scales nicely with the number of agents when run on a single host, managing a huge number of agents and running them in highly distributed environments introduces problems. We showed that managing 1000 agents on a single host causes no problems [Ygge and Akkermans, 1996], but scaling up to, say, a million distributed agents is not easily managed.

In our application area, power load management [Ygge and Akkermans, 1996], the local area networks normally have low communication costs, while the wide area networks have rather high communication costs. We believe that this situation with highly interconnected sub-groups, and somewhat looser connections between the sub-groups is very common for other application areas as well.

We recognize some important problems with traditional algorithms. These problems are relevant not only for our application but for distributed markets in general.

- As the number of agents grows, the number of messages received and sent out by the global auctioneer will be very large for each iteration. The work needed to process these messages at a single site will create a bottleneck.

  This suggests that the task of sending, receiving, and processing messages should be distributed among more than one auctioneer.

- Even if multiple auctioneers are used, if data is communicated at each step in an iterative algorithm, the system will have to wait for the *slowest* agent to communicate. This is likely to cause significant waiting time in practice, since

  - some agents might have rather slow communication, like in our application area where some of them are communicating via cellular telephones or over the electric power line.
  - even for devices that have fast communication according to today's standard, there is a substantial variation in response time, and when communicating with many agents it is likely that the maximum delay time will be large.

  This suggests that the number of iterations that requires communication to reach equilibrium should be kept small.

- Within a fairly large range, the cost for sending a message is more or less independent of the size of the message. Furthermore, the cost for processing the sent/received message is normally considerably smaller than the waiting time. Thus, instead of sending a message containing just one number – as suggested in, e.g., [Kurose and Simha, 1989] and [Ygge and Akkermans, 1996] – one can send a set of, say, 25 or even a few hundred numbers at essentially the same cost.

  This suggests that one should trade message size for iterations, *even if the total amount of communicated information grows*. Furthermore, the fact that the size of message headers and footers can be considerable compared to the net data when the messages are small, may cause the actual amount of data sent on the network to be *smaller* when few iterations are needed even if the amount of net data communicated increases. (See also [Cheng and Wellman, 1997].)

- If only the preferences of a few agents change, search for the new equilibrium from scratch should not be required.

  This suggests that information from previous computations should be stored and reused.

We would also like the algorithm to put minimum requirements on local data, i.e. not be dependent on properties such as the derivative of net demands and prices. Furthermore, it is important that the algorithm is numerically stable, i.e. be guaranteed to find the equilibrium as long as the requirements (e.g. that z is continuous and decreasing) are fulfilled, while at the same time being computationally efficient.

Such an algorithm is presented in the next section.

4 The CoTree Algorithm

The CoTree algorithm (COmbinatoric TREE) is a hierarchical approach to computing the equilibrium. Its principles and properties are described in this section.

4.1 Basic Principles

We let the hosts in the system form a logical tree.[1] An example of such a tree is shown in Figure 1. The basic idea is that at each level in the tree, a compound demand function (if a price-oriented approach is used) or price function (if a resource-oriented approach is used) is constructed. The compound function is a sampled function with k samples and it is used as the input to the computation on the next higher level in the tree.

[1]The hosts need not be physically coupled in a tree structure, though this is an advantage. For our application of power load management, they will physically form a tree.

Each host receives one message from each of its n children and produces a sample of k samples.[2] Thus, the required data is communicated in one signle round and the number of messages required is proportional to the number of hosts.

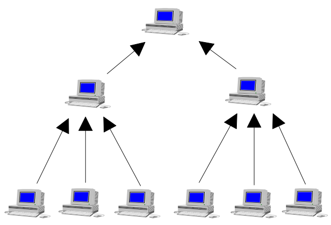


Figure 1: *A logical tree of hosts.*

The basic operation in CoTree for the computation of the compound function is the pair-wise aggregation of preferences, see Figure 2. Consequently, each host holds a binary tree where each leaf represent the preferences of each of its children in the host tree and the root node represents the compound function for all of its children. When the compound function of one host is send to the next level, it will be represented by a leaf at that level. Thus, the entire system can be viewed as a binary tree with all consumers and producers as leafs. We refer to the non-leaf nodes as *auctioneers*. The root auctioneer will hold the aggregate preferences for the entire system. Each auctioneer holds three vectors with information about its children: one compound demand or price function and two allocation vectors telling how much was assigned to each of the two children at each of the samples in the compound function.

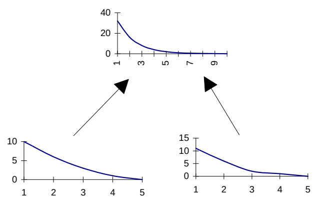


Figure 2: *The basic operation of* CoTree*: pair-wise aggregation.*

4.2 The Resource-Oriented Case

In the resource-oriented case the compound function is a price function which is computed as follows. Say that we are to compute the aggregated price, $p(3)$, that we in the previous iteration computed $p(2)$, and that the allocation to the left and right child at a total allocation of 2 was 1.5 and 0.5. Now since we know that the prices are monotonically decreasing with demand (since we had that the demand is continuous and decreasing with price as a precondition), we know that the only possible allocations for the left child is from 1.5 to 2.5 and, for the right child, 0.5 to 1.5. With CoTree we first try $\langle 1.5, 0.5\rangle$ and then $\langle 2.5, 1.5\rangle$. If none of these were the equilibrium solution we interpolate from these end points. This results in a constant number of operations for each slot of the compound vector. The computation has to be repeated for each auctioneer on the host. If all leafs are managed by one host the complexity is $\mathcal{O}(nk)$, where k is the number of slots and n is the number of leafs. If each host only manages two children, the complexity is $\mathcal{O}(k \log n)$. Further details are given in the Appendix.

[2]In this paper we (though this is not necessary) only discuss the use of one k everywhere in the system.

4.3 The Price-Oriented Case

CoTree, as presented above, is a resource-oriented scheme, but a similar price-oriented approach is also conceivable. This would mean that every auctioneer, instead of holding a price vector telling how much its children are willing to pay for an infenitecimal amount of resource at certain allocations, would hold a net demand vector, telling how much its children would change their allocation at different prices.

The advantage of such an approach is that the net demand vector of an auctioneer is simply the sum of the net demand vectors of its children. The complexity of this is $\mathcal{O}(nk)$, i.e. the same scaling with accuracy and number of children as with the resource-oriented approach. However, the computations here are simpler, and hereby the price-oriented approach would be faster. This assumes that the preferences of the consumers and producers are given as demand functions.

The disadvantage is that it results in an allocation which is not perfectly feasible. For cases where this is not acceptable, one could think of ways to assure that exactly the resource that is allocated to one auctioneer is allocated to its children. Then some computations would be required when computing the allocations rather than when computing the equilibrium (as was done in the resource-oriented case). For our application area, the computation time required for computing the equilibrium (as will be described in Section 6) is totally neglectable compared to the communication time. We believe this to be true for most other applications areas as well. Thus, whether a resource-oriented or price-oriented approach is chosen is of minor importance.

5 The Dynamic Situation

As discussed in the previous section, CoTree has the advantage of being communication sparse during the initial equilibrium computations. However, the major advantages of CoTree occur in the dynamic situations where an equilibrium has been computed once and then some agents change their preferences. This section discuss those issues. The discussion is relevant both for the resource- and price-oriented approaches.

5.1 The Change of an Agent's Preferences

If the preferences of only one agent change and everything else remains unchanged, the computations only have to be performed along the path from this agent to the root to compute the new equilibrium allocation.

Let d and n denote the number of children per host, and the number of leaves respectively. The tree of hosts has height $\mathcal{O}\left(\frac{\log n}{\log d}\right)$. As an example, assume that $d = 100$ and $n = 1000000$. Then the height of the three is 3. If the preferences of one agent changes, only 3 messages are required to compute the new equilibrium. This is in bright contrast to methods in which the aggregate functions are not cashed where one million messages would have to be sent if the agents submit entire functions and one million times the number of iterations if the agents only submit single information items. (Specifically, if single information items are used, such as the demand and its derivative at a certain price, it is impossible to cash data so that only three messages are required.)

We also note that if we have a large consumer or producer whose preferences change frequently, it is wise to put this agent close to the root, in order to minimize communication. In particular, if the agent representing such a consumer or producer resides on the same host as the root auctioneer, *no* inter-host communication is required for obtaining the new equilibrium. We note that here the computation of the new equilibrium is close to instant, while the traditional methods typically require recomputation from scratch.

5.2 When to Perform a Recomputation

For some applications it might be the case that the recomputations are performed periodically regardless of whether or not really needed. In other applications, recomputations might be performed when needed. We investigate the latter case here, and it turns out to be a delicate issue.

If the preferences change only for one agent, it is possible that the reallocation resulting from a total recomputation of the equilibrium is so small that it is not worth the efforts, even if the preference change seems quite large from the local perspective. At the same time, it is possible that there is a quite small change of *every* agent's preferences, which would lead to significant reallocations for the group as a whole. Thus, it is not possible to tell from the local perspective whether or not a recomputation is needed. The same holds at the global level; we can not tell if a recomputation is required without asking many local agents.

In an approximation scheme, we expect the equilibrium to be close to the true equilibrium without necessarily being exactly correct. Hence, after a small change of a few agent's prefernces it may not be necessary to change the (global) price/allocation at all. In this case, we would like the recomputation of price/allocation to "fade out" after very little communication as it is propagating upwards in the tree.

One way to implement this is to assign a permitted deviation at each level in the tree. Assume that we have a resource-oriented CoTree where we allow the values in the global price vector to deviate from the correct values with a maximum value of δ. We can then spread this tolerance out on the auctioneer in the tree. Each auctioneer i is assigned a tolerance δ_i, such that the total tolerance sums to δ. Exactly how the tolerances should be spread is an intriguing topic that requires a number of assumptions on the sizes and update frequencies in the individual agents' preferences. Once the tolerances have been assigned, each auctioneer keeps track of the difference between its current price vector and the price vector that was last sent to the parent. As long as this difference is less than (or equal to) δ_i, no message is sent to the parent. In this way, small local changes will not propagate up to the root.

It should be noted that if the changes in preferences can be seen as random, the changes are likely to even out as they propagate upwards in the tree.

In all, this is a non-trivial question and more research is needed here in order to obtain a good method. This problem is general and not unique to CoTree. We do however think, as indicated above, that one can benefit from using a tree structure when attacking this problem.

6 An Experimental Case Study

Above we argued that the CoTree algorithm is in most cases superior to available algorithms from a communication point of view. We also argued that the algorithm has good asymptotic behavior with respect to internal computation. However, in practice, constant factors may be as important as asymptotic behavior. Therefore, we give some performance measures of CoTree and compare them to corresponding figures for a Newton algorithm.

In our application area, one host can have at most

Method	Improvement	Initial computation		Update	
		Time (s)	# Messages	Time (s)	# Messages
Newton, eps = 0.001	32.7004	0.22	9838	0.11	4935
Newton, eps = 0.01	32.7004	0.19	8851	0.085	3948
Newton, eps = 0.1	32.7001	0.17	6885	0.065	2961
Newton, eps = 0.2	32.6975	0.13	5906	0.065	2961
Newton, eps = 0.3	32.6034	0.11	4919	0.044	1974
CoTree, 200 int	32.7004	2.58	1000	0.050	1
CoTree, 100 int	32.6992	1.54	1000	0.030	1
CoTree, 50 int	32.6963	0.99	1000	0.019	1
CoTree, 25 int	32.6647	0.77	1000	0.015	1
CoTree, 10 int	31.7043	0.66	1000	0.013	1

Table 1: *Simulation results with 1000 children.* CoTree *requires significantly less communication than the Newton algorithm, in particular for updates.*

something like 1000 children to manage, and we investigate a computation of this size. (Thus, for our application we get an upper limit of the total computation time from multiplying the computation time of one host with 1000 children by the height of the host tree.) For the experiments we let the agents hold simple exponential utility functions and act competitive[3]. We compare the performance of the resource-oriented CoTree algorithm as described in Section 4.2 with the performance of a resource-oriented Newton scheme using a Relax approach [Ibaraki and Katoh, 1988] for managing boundaries as described in [Ygge and Akkermans, 1996]. In the Newton scheme the agents submit bids consisting of their current price and its derivative (for details refer to [Ygge and Akkermans, 1996]). The result of the comparison is presented in Table 1. The improvement column is a measure of how good the result is. Since we know, for this case, that the market equilibrium maximizes the total utility of the system [Ygge and Akkermans, 1997], we can measure how close to the equilibrium the solution is by observing the total utility. The improvement is simply the difference between the total utility after the first reallocation and the initial total utility. The next column tells how long the execution time[4] was for computing the equilibrium from the initial endowments, and the column after that describes how many messages were required. Note that the number of messages is not necessarily a multiple of 1000 because of how the Relax algorithm works. If an agent is outside its boundary after one iteration it will not be a part of the next iteration.

Then we also illustrate the discussion given in Section 5.1; the rightmost columns show the computation time and the number of messages required when the preferences of one agent changes. For this situation we have assumed that the change does appear in one of the leaves of CoTree. As discussed in Section 5.1, if we have prior knowledge about the frequency with which the preferences change for different agents and how much this affects the market, we can design the topology of CoTree to take advantage of this. The results presented in this experiment thus represent the upper limit of the computation time of an update.

From Table 1 we see that when using 25 intervals with CoTree the ratio of execution time between CoTree and a Newton scheme with a corresponding accuracy is somewhere around six. This is a positive result since it means that CoTree is not *that* much slower than the Newton scheme. The result of a corresponding comparison for the case where only one agent changes its preferences, once the initial equilibrium has been computed, is that CoTree is now approximately four times *faster*.

Our experiments strongly indicate that CoTree performs well compared to a resource-oriented Newton scheme. At the same time it should be emphasized that the computation time when being of the magnitude described above is normally *completely neglectable*. For example, in our application area, power load management, the time for computing the allocation with 1000 children and 25 intervals (0.77s) is in the same order of magnitude as the time required for sending *one* message on the electrical power line with today's communication systems. Therefore, we argue that when the computation time is of this order of magnitude, it is really the

[3]The utility functions are $u(\mathbf{x}) = ae^{-bx_1} + x_2$, with randomly generated endowments and parameters a, and b. Hence, the price function for each agent will be $p_1(z_1) = -abe^{b(z_1+e_1)}$, where e_1 is the endowment (i.e. initial allocation) of commodity 1. Further, we let $0 \leq x_1 \leq 3$ and x_2 be without boundaries.

[4]The algorithms were implemented in C++ and run on a Pentium Pro 200MHz PC.

number of messages that is the interesting performance measure, and from this perspective CoTree (and any other method that saves entire preference functions from previous rounds) has considerable advantages.

Another important remark is that CoTree is not dependent of the derivative of the price and hereby useful for a wider class of problems. Furthermore, the execution time of CoTree is virtually independent of the shape of demand functions, as long as z is monotonically decreasing and continuous, while the convergence speed of the Newton scheme is heavily dependent on those shapes. Therefore it is likely that CoTree is preferable to the Newton scheme in many situations, even when every agent is run on a single host, and no inter-host communication is present.

7 Managing the Multi-Commodity Case

So far we have only demonstrated the search for equilibrium in a market with two commodities. Even though this setting is useful for some realistic problems, e.g. [Kurose and Simha, 1989; Ygge and Akkermans, 1996; 1997], it is of interest to investigate how CoTree can be used for the multi-commodity case.

The two-commodity case can be extended to the multi-commodity case in a variety of ways. One way to perform such an extension is to compute the equilibrium for one market (one market per commodity) at a time. This procedure is repeated until every market is in equilibrium. Note that during this process the equilibrium for one market may have to be computed several times, since there normally are interdependencies between the markets and a change of the price in one market will effect the price on another. For a detailed discussion on the convergence of this scheme refer to [Cheng and Wellman, 1997]. We note that if the different markets are loosely coupled, in the sense that just a few agents act on several markets, CoTree will perform very good compared to the alternatives. In this case only the preferences of a few agents change in each market, and we will benefit from the advantages of CoTree as described in Section 5.1.

On the other hand, if the markets are not loosely coupled, the approach of treating every market separately will not lead to very high performance (neither with CoTree nor with any other algorithm for finding the equilibrium of each market). Even though the separation in terms of markets allows for some distribution of the search for equilibrium, the auctioneers of the separate markets must come to consensus regarding whether or not the general equilibrium has been reached. Then it is not clear that the gain of decentralizing the computation is that large after all. It might be the case that the markets for the different commodities are inherently distributed, but then it does seem more reasonable that the resource is reallocated every time a partial equilibrium is reached, i.e. we would have repeated tatonnement processes in each market, but a non-tatonnement search for the general equilibrium. Furthermore, Flecther [Fletcher, 1987, p. 18-19] argues that decomposing this kind of search into separate searches for each variable is "usually very ineffective" and that these "... early *ad hoc* approaches are gradually falling out of use". Examples of alternative algorithms are multi-variable Newton algorithms with variable step size, were the search for the equilibrium prices is performed in parallel [Press *et al.*, 1994; Fletcher, 1987]. However, the drawbacks of current algorithms, as were described in Section 3, hold for the multi-commodity case as well.

As part of current work we are expanding CoTree in new ways for the multi-commodity case. It is a delicate task and it seems hard to find a general approach such as the one described here for the two-commodity case. Rather we believe that for certain applications one can find efficient algorithms if heuristics about, e.g., the relations between the demands of the different commodities, are utilized. (For example, in our own area of power load management, if the consumption for the current and future time slots are treated as the different commodities, the coupling between adjacent slots is much stronger than the coupling between slots far apart.)

8 Conclusions

In this paper we reported difficulties with using available algorithms in distributed computational markets. We introduced a novel algorithm CoTree which:

- requires minimum information for computing the equilibrium (i.e. certain derivatives as used with Newton methods are not required). It performs well regardless of if input is demand or price functions.
- is communication effective.
- performs excellent when responding to changes of e.g. single agent's preferences.
- scales up nicely, also to huge markets. When run in a totally distributed environment the execution time is $\mathcal{O}(k \log n)$, where k is the number of samples in the price or demand function and n is the number of consumers/producers. Even more important, the longest communication chain is $\mathcal{O}(\log n)$, the number of messages is for a total recomputation is $\mathcal{O}(n)$, and the size of each message is $\Theta(k)$. If only

the preferences of one agent changes, the required number of messages is $\mathcal{O}(\log n)$.

- has a number of advantages over Newton algorithms even when run on a single host.

As shown in this paper CoTree can be seen as an algorithmic framework, rather than a specific algorithm. There are still issues to be further investigated. In our opinion, one of the most interesting is how one should choose the number of samples (denoted k throughout the paper) on different levels in the system, depending on the computational power at those levels as well as heuristics of the price/demand functions.

We have also implemented a variant of the CoTree algorithm which is well suited for standard resource allocation, even with non-concave objective functions. A publication describing this will soon be made available.

References

[Cheng and Wellman, 1997] J. Cheng and M. Wellman. The WALRAS algorithm – a convergent distributed implementation of general equilibrium outcomes. In *Computational Economics*, 1997. To appear. (Available from http://ai.eecs.umich.edu/people/wellman).

[Fletcher, 1987] R. Fletcher. *Practical Methods of Optimization.* John Wiley & Sons, 1987. Second Edition.

[Ibaraki and Katoh, 1988] T. Ibaraki and N. Katoh. *Resource Allocation Problems – Algorithmic Approaches.* The MIT Press, 1988.

[Kurose and Simha, 1989] J. F. Kurose and R. Simha. A microeconomic approach to optimal resource allocation in distributed computer systems. *IEEE Transactions on Computers*, 38(5):705–717, 1989.

[Mas-Colell *et al.*, 1995] Andreu Mas-Colell, Michael Whinston, and Jerry R. Green. *Microeconomic Theory.* Oxford University Press, 1995.

[Press *et al.*, 1994] W. Press, S. Teukolsky, W. Vetterling, and B. Flannery. *Numerical Recipies in C.* Cambridge University Press, 1994. Second Edition.

[Varian, 1996] H. Varian. *Intermediate Microeconomics – A Modern Approach.* W.W. Norton and Company, New York, 1996. Fourth Edition.

[Wellman, 1993] M. P. Wellman. A market-oriented programming
environment and its application to distributed multicommodity flow problems. *Journal of Artificial Intelligence Research (http://www.jair.org/)*, 1(1):1–23, 1993.

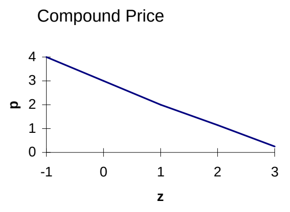
[Ygge and Akkermans, 1996] F. Ygge and J.M. Akkermans. Power load management as a computational market. In M. Tokoro, editor, *Proceedings of the Second International Conference on Multi-Agent Systems ICMAS'96*, pages 393–400. AAAI Press, Menlo Park, CA, December 9–14 1996. (Available from http://www.enersearch.se/ ygge).

[Ygge and Akkermans, 1997] F. Ygge and J.M. Akkermans. Making a case for multi-agent systems. In M. Boman and W. Van de Velde, editors, *Proceedings of MAAMAW '97*, pages 156–176. Springer Verlag, Berlin, May 13–16 1997. ISBN-3-540-63077-5, (Available from http://www.enersearch.se/ ygge).

Appendix: Details of the CoTree Algorithm for the Resource-Oriented Case

Structure and Communication

Each auctioneer keeps an approximation of the compound price function of its descendant nodes, as well as information on how to distribute resources between its children. This is visualized in Figure 3.



price	4	3	2	1.143...	0.25
leftAlloc	-1	-1	-0.5	0.0714...	1
rightAlloc	0	1	1.5	1.928...	2
Corresponding resource:	-1	0	1	2	3

Figure 3: *The structure of an auctioneer.*

The *price* vector holds k values, each value being the marginal price that the auctioneers children have at that change in resource, z. Thus, *price* gives the price at sampled points from the situation where the auctioneers children sells as much of the resource as possible to the situation where they buy as much as possible[5]. As an example we see that for $z = 1$ (the third slot), we have that $p = 2$. The $leftAlloc$ and $rightAlloc$ vectors are also holding k values corresponding to the respective equilibrium demands. For example, if the auctioneer is assigned $z = 1$ (i.e. will buy 1 unit of commodity one)

[5]For applications where there is no limit to how much an agent can buy, good guesses of a possible interval can be used. If the allocation should end up at the border of this interval, new limits can be set and only a small part of the equilibrium computations will have to be redone.

after the equilibrium computation, it gets directly from the *leftAlloc* and *rightAlloc* vectors that 0.5 will be sold by the left child and that 1.5 will be bought by the right child. If the auctioneer is assigned $z = 0.5$ it interpolates between adjacent values, and in this case it would mean that the left child would sell 0.75 and the right child would buy 1.25.

CoTree is implemented on a set of hosts interconnected in a tree structure. The height of this tree will vary from zero (when all auctioneers, consumer and producers reside on the same host) to the height of a binary tree when every auctioneer, consumer and producer resides on a separate host. We introduce a degree, d_i, defined as the number of children of host i. The choice of d_i for each host will depend on the communication and computation capacity of the employed hardware. If we assume that all hosts have the same degree d (which is not necessary) and the tree is well balanced, the height of the tree is $\mathcal{O}\left(\frac{\log n}{\log d}\right)$. Another important design parameter is k, the number of elements in the *price*, *leftAlloc* and *rightAlloc* vectors. The choice of k will depend on the tolerance on the quality of the result (*cf.* the discussion in Section 2.4) as well as the computer system's communication and computation capacity. Throughout the paper we assume k to be equal for every consumer, producer and auctioneer, although this is not necessary. We now take a look at two extreme cases of d.

As one extreme, we chose $d = n$, where n, as above, is the number of leaves in the tree, i.e. the number of consumer and producer agents on the market. This results in a host tree of height 1 where there are no internal hosts, only a set of n leaves and the root. We let each agent deliver all its k sample values to the root. The global auctioneer then computes an aggregated function. This requires that the global auctioneer receives n messages, each of size $\Omega(k)$.

A host tree of height 1 works well as long as the number of agents is not too large. For a large number of agents, the communication to the root may become a bottleneck and the computational burden of the host holding the root node may become too high (see further the Appendix).

As the other extreme case we choose $d = 2$ which gives a binary host tree, and let each host hold only one consumer, producer or auctioneer. The computation is now distributed among the hosts so that the root—as well as each internal host—only need to process two messages of size $\Omega(k)$. This minimizes the cost for communication (and computation) at each single host at the expense of more messages ($2n - 2$ compared to n)[6] and slightly longer communication chains; the height of the tree will be $\mathcal{O}(\log n)$.

[6]The number of messages is equal to the number of edges in the host tree. If we assume a balanced host tree with no unary nodes, the number of edges in a binary tree with n leaves is $2n - 2$.

The Computations of an Auctioneer

The basic structure of the algorithm is given below. The algorithm is explained in some detail after the pseudocode.

```
if the node only has one child then
   copy the vectors of that child
else
assign the highest p of the two children
   to slot 0 in price and assign the
   minimum values of the respective children
   to the first slots in leftAlloc and rightAlloc
for i:= 1 to k-1 {
//This loop is run k-1 times
   assign as much as possible to
          the right child
   if the price of the right child is larger
      or equal to that of the left child then {
      if the right child is at its boundary then
         assign the value of the left child to slot i
                   in price
      else
         assign the value of the right child to slot i
                   in price
      assign the current allocations to leftAlloc
           and rightAlloc
      continue with the next iteration
   } //end if
   assign as much as possible to the
      left child
   if the price of the left child is larger
      or equal to that of the right child then {
      if the left child is at its boundary then
         assign the value of the right child to slot i
                  in price
      else
         assign the value of the left child to slot i
                  in price
      assign the current allocations to leftAlloc
                  and rightAlloc
      continue with the next iteration
   } //end if
   interpolate to the point where the
      prices are expected to be equal
   assign price, leftAlloc and
      rightAlloc as described below
} //end for
```

If the node only has one child, the obvious solution is simply to copy its vectors. Otherwise we have to calculate the k values for the three vectors as described below.

The algorithm is most easily explained through an example. Assume that the *price* vectors sent by the children to the auctioneer are as in Figure 4. (The numbers in Figure 3 are from the computation with these price samples.) Note that $-1 \leq z_{left} \leq 1$ and $0 \leq z_{right} \leq 2$, and hence $-1 \leq z \leq 3$ for the auctioneer.

left price	3	2	1	0.5	0.25
Corresponding resource:	-1	-0.5	0	0.5	1
right price	4	3.5	3	2	1
Corresponding resource:	0	0.5	1	1.5	2

Figure 4: *The prices of two children.*

We start by assigning the highest value of slot 0 of the price vectors of the two children to slot 0 of *price* of the auctioneer. This is because the child with the highest price will start to buy at the auctioneers minimal z. Hence, 4 is filled in at slot 0, since $p = 3$ for the left child and $p = 4$ for the right child. The minimum values of the left and right children are filled in at slot 0 of *leftAlloc* and *rightAlloc* respectively.

Then we enter the `for` loop. For each $i = 0 \ldots\ k-1$, we test the allocation $z = min + \frac{i}{k-1}(max - min)$. First assign as much of z as possible to the right child. Assigning as much as possible to the right node means that the maximum value for that child can not be exceeded and that the value assigned to the left child can not be smaller than the value assigned to it in the previous iteration. In the first iteration this means that we start by assigning 1 to the right child and -1 to the left child. In this case $p = 3$ for both children and hence the condition (that the price of the right child is higher than or equal to the price of the left child) is true and 3 will be assigned to slot 1 in *price*, -1 will be assigned to slot 1 of *leftAlloc*, and 1 will be assigned to slot 1 of *rightAlloc*. Then we continue with the next iteration.

In the next iteration we start by assigning 2 to the right child and -1 to the left child. This time the condition (that the price of the right child is higher than or equal to the price of the left child) is false and we continue with assigning 1 to the right child and 0 to the left. As the price of the left child is smaller than the one of the right child, we continue to the interpolation. By now we have computed that $p_{right}(2) = 1$, $p_{right}(1) = 3$, $p_{left}(-1) = 3$, and $p_{left}(0) = 1$. We now do a linear interpolation to estimate where the prices are equal. Thus, we set up the equation

$$p_{right}(1) + \Delta z_{right} * \frac{p_{right}(1) - p_{right}(2)}{1-2} = \\ p_{left}(-1) + (1 - \Delta z_{right}) * \frac{p_{left}(-1) - p_{right}(0)}{-1-0}.$$

Solving the equation gives $\Delta z_{right} = \frac{1}{2}$ and the resulting price ($p_{right}(1) + \frac{1}{2} * \frac{p_{right}(1) - p_{right}(2)}{1-2} = 2$) is assigned to slot 2 in *price*, -0.5 is assigned to slot 2 in *leftAlloc* and 1.5 is assigned to slot 2 of *rightAlloc*. Correspondingly is done for $i = 3$.

For the last iteration ($i = 4$) we have that the first if statement is true and that the right child is at its boundary, thus we assign 0.25 to *price*, 1 to *leftAlloc* and 2 to *rightAlloc*. The rationale behind selecting the price of the other node when one of them is at its boundary is that once the boundary has been reached, this child can not buy anything more, and it will not sell before the price is above the price of the other child.

In this simple example the *price* exactly matched prices the resources requested for. In general this is not the case. Rather the price must be interpolated from adjacent values.

As seen from the pseudo-code above, for each of the possible allocations, the worst case is that both the if-statements return false and we have to perform the interpolation which is done in constant time. Hence, the computation time grows with the number of samples as $\mathcal{O}(k)$.

Looking at the total computation time of CoTree we again investigate the cases of $d = n$ and $d = 2$. When $d = n$ (all auctioneers reside on one host) the computation time is the computation time of each auctioneer times the number of auctioneers, i.e. $\mathcal{O}(nk)$. When $d = 2$ and we let each host hold only one auctioneer, the computation time is the computation time of each auctioneer times the height of the tree, i.e. $\mathcal{O}(k \log n)$. It is important to remember though that the communication, rather than the computation is the important performance measure here. (See further Sections 6.)