Distributed Environments for Evolutionary Algorithms by means of Multi-Agent Applications

Herbert Kopfer, Thomas Utecht, Christian Bierwirth

Dept. of Economics, University of Bremen *– Germany

Abstract

Advanced modeling of control and optimization in management science often leads to a computational complexity which cannot be handled by traditional algorithms and computer systems. On this background the paper develops a general approach to combine the power of distribution and parallelism in natural systems and modern distributed and parallel computing systems. The link between these topics is reached by the concept of multi-agent systems. We show how to build a genetic agent system upon a base model of distribution. Computational performance of this system is presented by a sample application to production scheduling and a runtime analysis of distributed and parallel processing.

1 Introduction

In recent years many decision problems from the domain of management science were approached and solved by some classes of evolutionary algorithms. Within such fields as production, inventory, distribution and organization we find a broad range of new applications for computational paradigms that were already developed during the last two decades [11]. Nowadays these paradigms lead to most promising new approaches which fit the complex demands arising from optimization, configuration and forecasting. Probably the most important feature of evolutionary algorithms is the explicit introduction of distribution. In order to combine the power of distributed problem solving and strategies from natural systems we outline a model for handling evolutionary algorithms by means of multi-agent systems. First steps of our research in this topic were reported in [2].

Section 2 gives a brief review on multi-agent systems and its extension to distributed agents. Complex attributes of distribution lead to a Base Model of interaction between agents in a distributed environment. Section 3 sketches the formulation of evolutionary algorithms in terms of agent systems, which enables this kind of algorithms to make use of the functionality of distributing agents as provided by the Base Model. Section 4 reports on the application of a distributed Genetic Algorithm to a class of production scheduling problems. Scheduling performance and distributed runtime analysis are discussed for a number of benchmark problems.

*Chair of Logistics, D-28334 Bremen, Email: kopfer@logistik.uni-bremen.de

2 Distributed Computing Environment

The actual state in Distributed Computing Environments (DCE) [12] is represented by a combination of Remote Procedure Calls (RPC) and concurrently running light-weighted processes (multi-threaded processes). How to use this environment in order to realize distributed multi-agent applications is covered in this section.

2.1 Multi-Agent Systems

An agent is the basic component in a set of independently acting agents constituting a *multi-agent system*. In order to solve common application problems, agents cooperate with other concurrent agents in the set.

Cooperation requires communication which is based on shared data or it takes place by exchanging messages. The power of multi-agent systems comes from agent action and interaction. The mechanisms of interaction are cooperation and concurrency. Each action of an agent is either cooperation via communication or it occurs independently and it is therefore concurrent to other agents.

Many approaches have been developed to describe the way of acting and interacting in multi-agent systems. Some authors propose a knowledge based approach to model the behavior of agents. Weigelt and Mertens [16] introduce *Partly Intelligent Agents* (PIA) where each agent follows its internal goal with respect to the current information about the environment. So internal goals of several agents in combination have to solve the application problem. Other authors focus on the ability of agents to react flexibly on events of a changing environment. Internal goals are not presupposed. Therefore the application problem (external goal from an agent's view) is solved only as a result of teamwork. Risking a more difficult explanation of how and why an application problem is solved both approaches may be combined.

2.2 Distributed Agents

A multi-agent system may be implemented on operating systems which support concurrency for agents. Newer systems like Solaris 2.3, Mach 2.6, OS/2, Windows NT and OSF/1 support this kind of concurrency by using a *multi-threaded* process. An agent is therefore typically implemented in a thread. Communication is carried out via shared memory.

To distribute a multi-agent system in a network of processing elements, e.g. a workstation cluster, communication has to be done by exchanging messages. A global state (time, data) cannot be determined because of the lack of shared memory. An agent still should be implemented by a thread. Next it should communicate with other agents in a way which is independent from locations. Therefore agents communicate in a transparent way. Beside *transparency* and according to the currently available knowledge of understanding distributed systems there are some more attributes we have to comply with. These attributes are *autonomy*, *separation*, *flexibility* and *heterogeneity*. Each attribute can be

examined from different viewpoints in a distributed system. So beneath *transparency of location* we may have *transparency of service* meaning that we do not have to care how a service is provided. The service provider itself may have to react *flexibly* on a possible event (i.e. power down) affecting its service quality. With these attributes in mind we build a model of distributing agents in a workstation cluster depicted as the *Base Model* .

According to Enslow's model of decentralization as described briefly by Sloman and Kramer [15] our model can be classified as follows: From an agent's viewpoint complete distribution (concerning control, data and processors) is achieved with the only exception of supporting heterogeneous processors.

2.3 Base Model

Our model of distribution is developed straightforward from the imagination of real world teamwork. That is why we like to introduce it in terms of team, office, mailport, notice-board, agent and message.

A team consists of quite an arbitrary number of agents each belonging exactly to one office. In the most simple case we have only one office and therefore all agents belong to this office. An office administrates the agents themselves and provides further services for communication and cooperation between agents. These services are represented by different types of communication objects which are mailports, notice-boards and agents. For addressing any of these objects an office has a hierarchically ordered (according to the type of object) name space.

A message within the Base Model consists of a structured addressing part and a data part of variable length. An agent is the only active object within our model that generates messages on its needs and sends them to a communication object by adding the office and name of the recipient. An involved office processes a message according to the type of the addressed communication object.

In case of a mailport the message is simply stored preserving the order of messages in the mailport. If a notice-board is addressed, the previous message in the notice-board will be replaced by the actual one. When a communication object of type agent is addressed the office starts a new instance of the corresponding agent. This instance gets the message sent as its initial input.

Let us summarize the usage of communication objects in the Base Model: The term "sending a message to an object" means to transport information to a passive object (mailport or notice-board) or to an active object (agent that is starting to "live" upon receiving a message).

The sending agent of a message which is addressed to a communication object of type agent (i.e the creator of a job) gets an identification number in return to its request. Using this identification the sending agent can wait for the job to finish and it can receive an answer message.

In order to realize a distribution of agents an initially existing office opens further offices at different locations. Compared to the simple case of only one office this extension is transparent from an agent's view. The addressing field of a message still contains the the

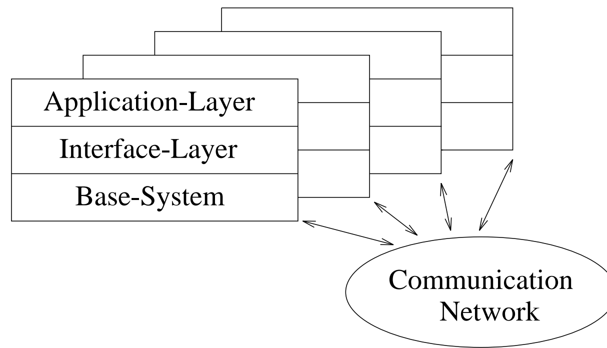

Figure 1: Layer Model

entries of office and object name. In contrast to the simple case (one office) the office now does not need to be the own one. Because a new agent is created by sending a message this occurrence is also still transparent.

A distributed system must be able to react flexibly on an unexpected event e.g. a total failure of an office or a message addressed to a so far unknown object. The Base Model manages these kinds of events by sending appropriate messages to predefined communication objects which are integrated into the model itself. So the predefined communication objects represent a default behavior for unexpected events.

If we now focus on the event of sending a message to an unknown object at least two different reactions may be appropriate. First the message may be discarded silently or secondly the missing object may be created (on demand) and the message is finally delivered. Examining this simple case we can state that an appropriate reaction may only be predicted in the context of a real application, mainly if we wish to achieve the desired behavior. The Base Model therefore allows to replace communication objects (here: especially predefined objects) by application-specific ones to change the default reaction.

So far we have given an informal description of the Base Model in terms of a real world environment. This seems to be adequate because the model is developed from adopting the structure of real world team work.

2.4 Implementation

For building our distributed computing environment a three layered implementation architecture is used. The idea is to become increasingly specific from the bottom layer up to the top layer which is in fact the real world application itself.

The bottom layer implements the Base Model and is named as the Base System. Above this layer a more specific Interface-Layer is used to provide additional support for a specific class of applications. Here the focus is on the class of "population-based agent systems" – a term that is made more precise in the next section. Finally the third layer represents the application itself. Figure 1 gives a snapshot on the layer model distributed on four workstations.

Diverse tests	Results
Local message transfer rate (128 - 1024 Byte) (independent from message size, varying)	2500 - 3500 Messages/s
Remote message transfer rate (128 - 1024 Byte) (results are varying)	200 - 500 Messages/s 40 - 400 Kbyte/s
Average turn around time for a local empty job (independent from size of parameter)	2.4 ms
Average turn around time for a remote empty job (20 Byte request and reply)	6.5 ms
Average turn around time for a remote empty job (1024 Byte request, 20 Byte reply)	7.4 ms

Table 1: Communication performance of the base model implementation.

For implementing the Base System we have chosen the Mach 2.6 and Solaris 2.3 operating systems. Actually an office is implemented in a multi-threaded process. In each office the objects are identified by names. The complete address of an object is a dublet , given by the office number and the object name. Message buffers are structured to store addressing information and the message itself in a continuous chunk of memory, which enables us to avoid unnecessary copying of messages within an office. Consequently sending a message within an office only requires to exchange a reference to a message buffer. An agent is realized by a thread of the process representing an office.

The Base System is accessible by using an additional library which represents the programming interface to our Base Model of distribution. We measured the performance of the implemented Base Model in terms of message throughput, message latency, turn around time for asynchronous jobs and overhead caused by the administration of named objects. According to the limited bandwidth of our network (Ethernet) the measurements match with our expectations of what can be called efficient distributed processing.

Table 1 gives a raw overview of the performance achieved so far. All test were made using two "low-end workstations" of type Sparc Classic (50 Mhz Micro-Sparc Processor) connected via Ethernet. The system performance of this workstation is roughly comparable to modern PC-hardware (Intel 486, 66 Mhz).

In the literature [13] similar tests for the turn around time of a remote job (asynchronous Remote Procedure Call) resulted in 7.3 to 10.9 ms for input parameter sizes from 20 bytes to 1 Kbyte. The tests were carried out on DEC 5240 workstations using the DCE (Distributed Computing Environment of OSF) implementation under the Ultrix operating system.

3 Distributed Problem Solving

In the domain of *Distributed Problem Solving* a major research direction focuses on search and optimization techniques that are inspired by nature, containing such diverse compu-

tational paradigms as *Evolutionary Strategies*, *Genetic Algorithms* and *Cellular Automata* [14], [9]. A common property of these paradigms is that they perform some kind of local search based on a multitude of individual searchers and not on a sequential step by step process of single search trials. The global state of such a system can evolve without centralized control. Solely local interaction between searchers can generate high quality solutions of a search problem.

The difference between these search strategies and multi-agent systems relies on the fact that all involved agents follow the same program. Such a program is an agent's life routine of alternating individual search and communicating with other agents. Although agents work on the same problem, they explore different locations of the search space. The effectivity of shared search arises from specific cooperation schemes adapted from natural processes. All agents participating in such a search process are called a population. We therefore suggest to refer to these paradigms as *population-based agent systems*.

3.1 Population-Based Agent Systems

The most frequently observed difficulty in the application of population-based agent systems to optimization arises from *premature convergence*, i.e. search stagnates early by means of suboptimality, e.g. see Eshelman and Schaffer [4]. In such a state all agents have similar characteristics which confines any cooperation to very small rates of improvement. A promising approach to avoid this phenomenon comes from spatially distributed populations. The feature of isolating agents in space requires the use of communication structures. Hence those components of a cooperation scheme that make decisions according to the overall state of the agent system are affected. Using distributed populations therefore requires to include such components into the communication structure.

In spatially distributed populations communication of agents is restricted to a small number of nearby agents. Hence global premature convergence is alleviated at the expense of local convergence. While the environment changes constantly at a slow pace for agents, cooperation from generation to generation works well. But if locality is introduced, the searching process is too fast to solve some of the problems faced by members of the population.

Therefore we developed a new approach which extends the overall adaptation of a population-based agent system towards the adaptation of a single agent to its neighborhood environment. Here each agent must respond to its own specific environmental conditions. Thus agents are able to change behavior usefully as a function of immediate changes of their environment. This is local short-term learning. We borrowed the basic ideas of our model of behavior-changes from the phenomenon of social hierarchies and individual behavior, which can be found in natural populations. Now we roughly describe the main components of our agent-model, for a detailed description see Mattfeld et. al. [10].

The initial "social" attitude of agents is an established one, i.e. they all act cooperatively within their environments. Secondly, the elitist attitude follows a conservative behavior pattern. The last attitude is a more critical one, which tends to be risk-prone. The actual behavior of each pattern is rewarded or punished in terms of social interaction. Again

we classify three simple responses which are defined by reinforcements. An agent can be pleased, satisfied or disappointed. The success of the actual behavior carried out may change its attitude and therefore changes its habit in a similar situation within the near future. The agent will react differently and may receive a different reinforcement on the same environmental situation.

In most cases a cooperative agent will be satisfied and therefore does not change its attitude. If pleased by success of its habit, next time it will tend to act conservative trying to keep its previous performance level. With this elitist attitude an agent can only be satisfied or disappointed by the success of its habit. In case of disappointment it will change back to the established attitude. Failing on cooperative behavior brings up a critical attitude of the agent towards its neighborhood environment. It will then tend to a more risk-prone behavior. The critical attitude is kept so long as a disappointing response is still received. If the agent is satisfied by the result of its behavior, it may change to the established pattern again. In rare cases a risk-prone agent will receive a pleasing response. Then it changes towards the elitist attitude and acts conservative.

3.2 Application to Evolutionary Algorithms

The computational paradigms of evolutionary Algorithms fit well into our understanding of population-based agents systems. We now demonstrate its application to Genetic Algorithms (GA). For an introduction into this paradigm see e.g. Goldberg [5]. The main components of GA's are:

- **(a)** A genetic representation which encodes the reachable points of the search space. Each valid instantiation of such a representation scheme describes the characteristics of a feasible solution which can be explored by agents,
- **(b)** A fitness-evaluation function which assigns each agent a real-valued number according to the quality level of its actual solution,
- **(c)** A selection procedure which builds up communication links between agents with respect to their level of fitness and to the predefined population structure,
- **(d)** Genetic operators which generate unexplored points in the search space from the encoded solution of one agent (mutation) or the encoded solutions of two distinct agents (crossover).

With the exception of (c) all components require knowledge from the application problem. In section 4 we will outline these components in detail in order to solve production scheduling applications. Let us now focus on the question of how to incorporate our model of agent-behavior briefly outlined in the previous section into a GA-based agent system.

To simplify matters we consider a population structure where agents reside on a torodial square grid. Communication of agents is restricted to the "North", "South", "West" and "East" neighbors. The selection procedure – component (c) – is carried out locally by a ranking scheme of 40%, 30%, 20% and 10% from the best to the worst-fit neighbor.

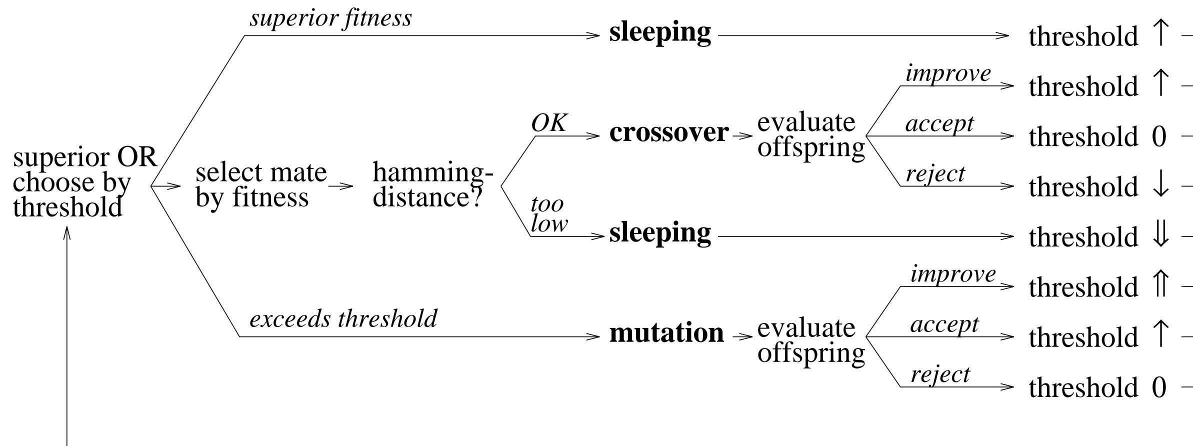

Figure 2: Control model of local recombination.

Whenever an agent generates a new solution which is at most 2.5% worse than the actual solution, it replaces the old by the new solution.

In order to implement individual agent behavior we now transform the metaphor of social attitudes into a so called *local recombination strategy*, see figure 2 and for a general overview [7]. The established attitude corresponds to cooperation with one of the neighbors by crossover. The critical attitude corresponds to a mutation. The conservative behavior tries to save the reached state. The agent performs no active operation (i.e. is sleeping) to avoid replacement by a new solution. The figure show these operations in boldface.

First an agent looks at the fitness of its neighborhood. If its fitness is superior to all neighbors, the conservative behavior will cause the agent to sleep. An inferior agent determines its attitude. The actual behavior is drawn probabilistically from a threshold. Initially the threshold is set to 1, which enforces crossover. Decreasing the threshold increases the probability of mutation. In case of crossover the hamming distance between the encoded solutions of an agent and its selected "mate" is evaluated. If two solutions differ in less than 1% of their characteristics it is not worthwhile to try a crossover. Again, the agent sleeps, but now because of a different reason. If crossover or mutation is carried out, the generated solution is evaluated. It either dominates the fitness of the actual solutions of both "parents" (improve), or the acceptance rule decides whether the agent replaces the new solution (accept/reject).

Summing up all distinct operations we count 8 responses which are tied to reinforcements of the threshold. We modify the threshold by rules of plausibility. The symbols "$\Uparrow\Downarrow\downarrow\uparrow$" express the degree of change of the threshold. This rule set attempts to adjust the behavior of each single agent towards the environment of its actual neighborhood. In our implementation a setting of $(+0.02, +0.05, 0, -0.02, -0.20, +0.15, +0.05, 0)$ performed well. This setting reacts adaptively to local convergence with a strong decrease of the threshold. It favors risky behavior by mutations in further generations. If a mutation succeeds, the threshold is increased which in turn leads to crossover.

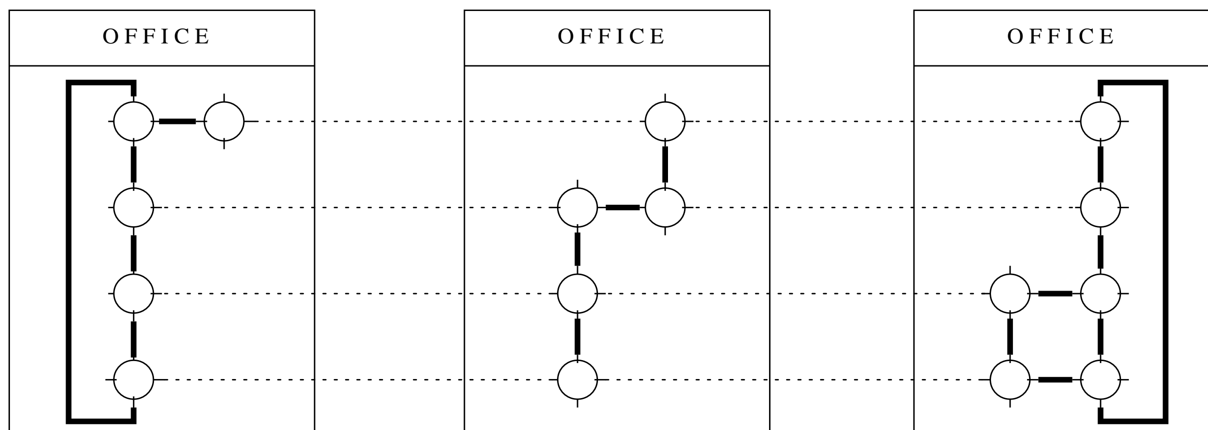

Figure 3: Torodial mapping of agents

3.3 Implementing GA-Based Agent Systems

In order to distribute population-based agent systems in a workstation cluster we use the functionality of the Base Model. Actually this step is executed by the Interface Layer. This section briefly outlines how the interface handles the communication between agents. But let us first become aware of the advantage of formulating distributed search strategies in terms of multi-agent systems. It enables applications to carry over the attributes of distribution by inheritance from the Base Model. From the application's viewpoint no difference arises from using a single workstation or a whole cluster. Of course, in the latter case we get enhanced performance by parallel processing.

In the literature diverse models of spatial GA-populations are proposed, e.g. Gorges-Schleuter [6]. The basic idea of these models is to structure populations by discrete topologies (e.g. islands, overlapping neighborhoods). This feature isolates population members according to a measure of space or time. Thus initially fixed areas of local communication result for all agents of the population. The Interface Layer realizes the communication topology of agents within the distributed computing environment of a workstation cluster.

The goal of the Interface Layer is to provide communication topologies as a service to the application. The area of communication (neighborhood) of agents must be fully transparent from its point of view. To illustrate this we now consider the sample population structure of a quadratic torus.

In this structure each agent is connected to a North, East, South and West neighbor. The Interface Layer maps the agents on the cluster with respect to a static load balancing and a minimal number of links across the network. Figure 3 shows the mapping of a population with 16 agents on 3 offices. In this case the distribution results in a different work-load (number of agents) for some offices.

If the population dynamics is synchronized (generation progress is step by step) a uniform work-load is preferable because the longest computing time of a generation step determines the overall progress. When the mapping tries to maximize the number of local links (bold lines) for heavy loaded offices, a slight alleviation results .

For each agent 4 mailports are exclusively used to receive information from its neighborhood. Sending of information to neighbors is oriented via the 4 directions of the compass rose. So we have full location transparency for an agent. Actually there is no need to specify the office when sending information to a neighbor agent.

The Interface-Layer is implemented with the object oriented programming language *C++* in contrast to the Base-System which is realized in *C*. Nevertheless both interfaces are simultaneously available to an application written in *C++*.

4 Application to Production Scheduling

In order to validate our distribution model this section develops a genetic algorithm (GA) for one of the general production scheduling problems going by the terms of *n-job m-machine scheduling* or *static job shop problem*. This optimization problem is briefly described by its three major components:

The manufacturing system: We consider a manufacturing system of m dedicated machines $M_1, \ldots M_m$, i.e. a machine cannot substitute other machines (no parallel machines). Further we assume the system to be idealistic, i.e. no machine breakdown occurs and passing times of tasks between machines are neglected. Tasks do not require a setup time, hence machines are available if they are not busy.

The production program: A production program covers n jobs $J_i, \ldots J_n$ that are released at predetermined points in times r_i. Each job J_i defines a technological order $\mathcal{T}_i = (o_{i\phi_i(1)}, \ldots o_{i\phi_i(m_i)})$ (precedence constraint) of processing m_i operations with processing times $p_{i\phi_i(1)}, \ldots p_{i\phi_i(m_i)}$ by the machines $M_{\phi_I(1)}, \ldots M_{\phi_i(m_i)}$. Hence, if $\phi_i(k) = j$, M_j is the k-th machine that processes J_i. No job has to be processed twice by the same machine and it does not necessarily has to be processed by all machines.

The measure of performance: Scheduling a production program in the manufacturing system means to find a table of starting times t_{ij} of all operations with respect to release times and precedence constraints. In order to optimize scheduling we consider the performance measure of minimizing the makespan for the entire program. Let $C_i = t_{i\phi_i(m_i)} + p_{i\phi_i(m_i)}$ be the completion time of J_i, hence we calculate the makespan by $C_{max} = \max\limits_{1 \leq i \leq n} C_i$.

We now turn to the description of the job-shop-specific GA-components (a), (b) and (d) of section 3.2. A more general description of this approach can be found in [3].

In the job shop model n denotes the number of jobs J_i. If m_i denotes the number of operations of J_i, a permutation with repetition of three jobs that contains $m_i = 3, 4, 3$ tasks, is for example given by

$$\left(J_1, J_2, J_2, J_1, J_3, J_1, J_2, J_3, J_2, J_3\right).$$

Here J_2 has to be processed on all machines whereas J_1 and J_3 have to be processed only by three of them. If we consider operations as tasks on machines the above permutation with repetition is interpreted as a task sequence

$$\left(T_{11}, T_{21}, T_{22}, T_{12}, T_{31}, T_{13}, T_{23}, T_{32}, T_{24}, T_{33}\right).$$

Reading it from left to right, a task T_{ij} of job J_i has to be scheduled on machine $M_{\phi_i(j)}$ as determined by the technological order $\mathcal{T}_i$. Assume that technological constraints $\mathcal{T}_1 = (o_{11}, o_{13}, o_{12})$, $\mathcal{T}_2 = (o_{21}, o_{22}, o_{23}, o_{24})$ and $\mathcal{T}_3 = (o_{32}, o_{33}, o_{34})$ are given. This leads to processing orders of tasks on the machines:

$$M_1 := T_{11}, T_{21}; \quad M_2 := T_{22}, T_{31}, T_{13}; \quad M_3 := T_{12}, T_{23}, T_{32}; \quad M_4 := T_{24}, T_{33}.$$

Notice that the described technique allows a unique transformation of every n-permutation with m_i-repetitions into a feasible symbolic solution in terms of processing orders for machines. Thus we can use the permutation scheme to represent the job shop problem genetically, i.e. as GA-component (a).

Any instantiation of the representation scheme describes a possible state of a "searching" agent. In order to evaluate its state, i.e. the fitness of the actual permutation, the agent calculates a minimal makespan for the symbolic solution using the GA-component (b). This process requires to schedule each operation of the production program as early as possible. Thus schedule building means to start an operation immediately after the longer completion time of the last scheduled task of the same job or the same machine. The agent considers the resulting C_{max}-value as its actual fitness. Notice that maximizing fitness means to minimize the measure of performance of this application. As an additional feature of fitness evaluation genetic search can be supported by use of any kind of job shop improvement-heuristic. For a further description of our hybridization approach see Mattfeld [10].

Finally we focus on GA-component(d) which represents the cooperation scheme of agents. In case of mutation actually no cooperation takes place. A mutation performs a simple random exchange of two arbitrary positions in a given n/m_i permutation of a single agent. According to the GA paradigm real cooperation of agents is temporarily restricted to "pairwise-mating" in order to generate new solutions. In analogy to natural genetics the process is called crossover. The crossover operator rebuilds a new instantiation of the genetic representation scheme from the actual solutions of the cooperating agents. In this way it ensures to pass a balanced mix of encoded characteristics with respect to the n/m_i-permutation structure of both solutions to the offspring. This technique can be derived from a generalization of the *Order-Crossover* (Oliver et. al [8]) for simple permutations. A detailed description of *Generalized Order-Crossover* can be found in [3].

5 Computational Validation

The local-genetic recombination strategy, shown in figure 2, was embedded into a population-based agent system as described in section 3.2. The resulting distributed algorithm was implemented by use of the Parnet base library to run on a variable number of UNIX workstations in parallel, see section 2.4. In the following we refer on this program as "Parallel Genetic Algorithm + Social Behavior Patterns" (PGA+SBP). Parallel

n-job/m-machine Benchmarks					PGA+SBP Results			Parameters	
Name	n	m	Cplx	C_{max}^{KNOWN}	$C_{max}^{AVERAGE}$	C_{max}^{BEST}	Trials	Gen.	Torus-Size
mt06	6	6	S	*55	55.0	*55	7628	100	10×10
mt10	10	10	M	*930	947.5	*930	10935	150	10×10
mt20	20	5	M	*1165	1188.2	*1165	11738	150	10×10
la21	15	10	L	1048	1061.5	1053	24956	300	10×10
la24	15	10	L	*935	948.0	938	25335	300	10×10
la25	15	10	L	*977	989.1	*977	25733	300	10×10
la27	20	10	L	1242	1265.7	•1236	24797	300	10×10
la29	20	10	L	1180	1214.4	1184	25013	300	10×10
la38	15	15	L	1203	1222.4	•1201	24026	300	10×10
la40	15	15	H	*1222	1243.5	1228	47881	400	12×12
abz7	15	20	H	667	684.6	672	45812	400	12×12
abz8	15	20	H	670	697.9	683	44664	400	12×12
abz9	15	20	H	691	712.6	703	45004	400	12×12

*optimal solution, •new-best solution.

Table 2: PGA+SBP scheduling results for 13 job shop benchmark problems

computing action was synchronized, hence the quality of PGA+SBP solutions is independent from the degree of parallelism. All computations of PGA+SBP were executed on a cluster of 20 Sparc-Classic machines (Micro-Sparc, 50 Mhz) connected via Ethernet.

5.1 Scheduling Performance

PGA+SBP has been applied to a suite of 13 job shop benchmarks. It includes the 3 famous Muth-Thompson problems (mt06, mt10, mt20) and the 10 most difficult problems of the 53 benchmark-suite provided by Applegate and Cook [1]. A short description of the test suite is listed in table 2. According to Applegate and Cook the scaling complexity of the problems can be classified by “S” for small, “M” for moderate, “L” for large and “H” for huge scale. It can be seen from the table that some of the problems are still open, i.e. not all best-known solutions are proved to be optimal. PGA+SBP is parameterized according to the assumed complexity of a problem, i.e. the population-size was set to 100 and 144 respectively and termination occurred after 100, 150, 300 or 400 generations.

PGA+SBP were run for a total of 30 iterations on each problem. Computational results in terms of the average generated makespan and the best found makespan appear in table 2 as well. Notice that these results were achieved by use of a computationally expensive improvement-heuristic for makespan evaluation as remarked in section 4. The column “Trials” refers to the average number of fitness evaluations in a single run. Notice that the value is approximately 20% smaller than the theoretical maximum number of trials (generations times population-size). This is caused by “sleeping” of agents, hence agents avoid makespan evaluation in about one of five generations.

In all runs solutions generated by PGA+SBP are close to the best-known solutions. The average generated makespan maximally differs by 4% from the best-known makespan of

No. of Workstations	1	2	4	6	8	10	12	14	16	18	20
Agents/Workstation	100	50	25	16–17	12–13	10	8-9	7-8	6-7	5-6	5
Problem	Runtime in sec.										
mt06	133	88	45	35	29	22	21	18	16	14	12
mt10	621	313	157	116	90	71	65	57	51	43	38
la27	2428	1174	664	452	354	278	244	222	202	171	144

Table 3: PGA+SBP runtimes for a small, moderate and difficult benchmark.

problem “abz8”. The best at all found solutions are within a 13-unit range of the best known solutions. Four benchmarks were solved to optimality by PGA+SBP and for two other benchmarks even new best solutions were found. Notice that “la27” is solved by a makespan of 1236. This value differs by only one unit from the theoretical lower bound, thus we assume this problem to be solved to optimality, too.

5.2 Runtime Performance

To validate the computational performance we choose three problems, in particular one of small, moderate and large-scale complexity (mt06, mt10, la27). Runtime performance of PGA+SBP (population size = 100 agents, generations = 100) on a varying number of workstations (1-20) is presented in table 3. The work load of a single workstation varies according to unbalanced mappings of agents as shown in the second line of the table.

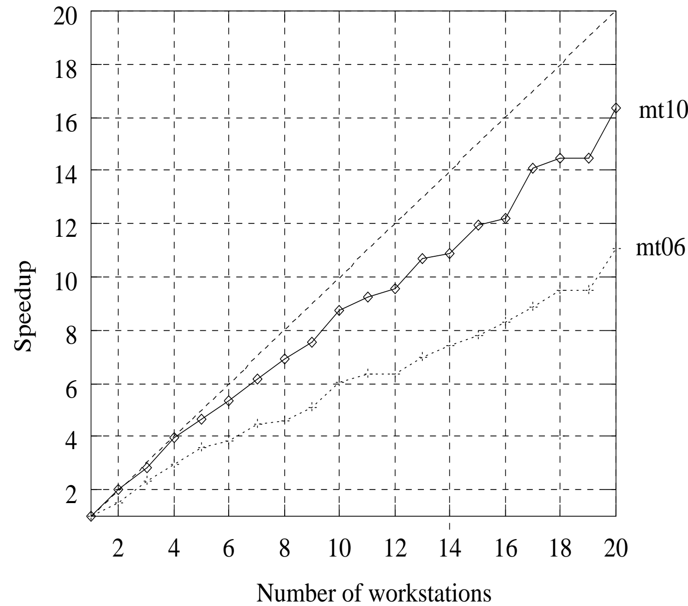

Figure 4: Speedup for 2 job shop problems

The speedup for “mt06” and “mt10” is charted in figure 4. Because the speedup for “la27” nearly matches the “mt10” curve within the range of 20 available workstations it is not displayed.

Due to their complexity both examples show a reasonable speedup. Obviously, "mt10" shows a better speedup in comparison to "mt6" because the quotient of communication (equal in terms of the number of messages for both problems) to computation load (less expensive fitness evaluation for the smaller problem) decreases strongly. But within the range of 20 workstations a saturation of scaleability does not happen even for the small problem. A slight staircase shape can be seen clearly in both curves. This is caused by a similar work load on the heaviest loaded workstation, e.g. 5-6 agents share one station while using a total of 17, 18 or 19 stations.

6 Conclusions

The idea to solve large-scale application problems by a team of agents and by the use of the computational power of a workstation cluster leads to the development of a Base Model and its implementation which makes a sophisticated and easy to use Distributed Computing Environment available.

In a next step the adaptation of Genetic Algorithms to the complex needs of production scheduling and to the technical limits (communication bandwidth) and attributes (mainly the autonomy of agents) of DCE shows two main results:

- At least for a cluster of 20 workstation the communication bandwidth of a standard Ethernet suffices for the communication needs of Genetic Algorithms. Hence a specialized parallel computing device is not needed.
- The solution quality does not suffer from distribution as shown by the computational validation. In a suite of well known benchmarks even two new best solutions were found.

These promising results underline the importance of recent research in distributed problem solving and machine learning for management science and operations research.

Acknowledgements

This research is supported by the Deutsche Forschungsgemeinschaft (Project Parnet). Furthermore, we like to thank Dirk Mattfeld and Dr. Klaus Schebesch for many helpful discussions.

References

[1] D. Applegate, W. Cook: *A Computational Study of the Job-Shop Scheduling Problem*, ORSA Journal on Computing Vol. 2, No. 2 (1991) 149-156

[2] C. Bierwirth, H. Kopfer, D. Mattfeld, T. Utecht, *PARNET: Distributed Realization of Genetic Algorithms in a Workstation Cluster*, Conference Preprints of APMOD93, Budapest (1993) 41-48

[3] C. Bierwirth, *A Generalized Permutation Approach to Job Shop Scheduling with Genetic Algorithms*, submitted to E. Pesch, S. Voss (eds.): Local Search, OR Spektrum (1994)

[4] L. Eshelman, D. Schaffer: *Preventing Premature Convergence in Genetic Algorithms by Preventing Incest*, in: 4. Int. Conf. on Genetic Algorithms, Morgan Kaufmann Publishers (1991) 115-122

[5] D. Goldberg: *Genetic Algorithms in Search, Optimization and Machine Learning*, Addison Wesley (1989)

[6] M. Gorges-Schleuter: *Explicit Parallelism of Genetic Algorithm through Populations Structures*, in [14] 150-159

[7] M. Gorges-Schleuter: *A Comparison of Local Mating Strategies in Massively Parallel Genetic Algorithms*, in [9] 553-562

[8] I. Oliver, D. Smith, J. Holland: *A Study of Permutation Crossover Operators on the Traveling Salesman Problem*, in: Proceedings of the 2nd Int. Conf. on Genetic Algorithms and their Applications, Lawrence Erlbaum Associates, Hillsdale (1987) 224-230

[9] R. Männer, B. Manderick (eds.): Parallel Problem Solving from Nature, 2, North-Holland (1992)

[10] D. Mattfeld, H. Kopfer, C. Bierwirth: *Control of Parallel Population Dynamics: Social-like Behavior of GA-Individuals Solves Large-Scale Job Shop Problems*, submitted to: Parallel Problem Solving from Nature 3, Jerusalem (1994)

[11] V. Nissen, *Evolutionary Algorithms in Management Science*, Papers on Economics and Evolution No. 9303, European Study Group for Evolutionary Economics, University of Göttingen (1993)

[12] Open Software Foundation: *Introduction to OSF DCE*, Open Software Foundation, Cambridge, USA (1992)

[13] A. Schill, *DCE – Das OSF Distributed Computing Environment*, Springer (1993)

[14] H. P. Schwefel, R. Männer (eds.): *Parallel Problem Solving from Nature*, Lecture Notes in Computer Science No. 496, Springer-Verlag (1990)

[15] M. Sloman, J. Kramer, *Distributed Systems and Computer Networks*, Prentice-Hall (1987)

[16] M. Weigelt, P. Mertens, *Theorie und Anwendungen von Agenten-Systemen in der Produktionssteuerung*, Wirtschaftsinformatik Working-Paper No. 12, University of Erlangen-Nürnberg (1992)