Teamwork-PaReDuX: Knowledge-based Search with Multiple Parallel Agents

Jörg Denzinger[1], Carsten Sinz[2], Jürgen Avenhaus[3], and Wolfgang Küchlin[2]
[1]*Computer Science Department, University of Calgary, Canada*
[2]*Fakultät für Informatik, Universität Tübingen, Germany*
[3]*Fachbereich Informatik, Universität Kaiserslautern, Germany*

Abstract

We present the combination of a distribution approach and a parallelization concept for knowledge-based search. We formally characterize distribution and parallelization and present one instantiation of each, TEAMWORK *and PaReDuX.* TEAMWORK-*PaReDuX, the combination of them, employs collaborating parallel search agents to prove equational theorems. Our experiments indicate that the speedups obtained by the single approaches are multiplied when using the combination, thus making good use of networks of multi-processor computers and allowing us to solve harder problems in acceptable time.*

1. Introduction

Knowledge-based search is at the core of most problem solving techniques. In most cases, it has to deal with large search spaces, i.e., for most states generated during a search run there are many possible successor states. So, there are many possibilities to design search control strategies for knowledge-based search systems. Selecting the best control for a given instance of a problem is very difficult. Due to the large search spaces and the many possibilities for traversing them, improving the efficiency of search systems is an important issue for research in many application domains.

The use of several processing units, i.e., processors or whole computers, has proven to be a good way to speed up the time needed to find solutions. Since there are many different paradigms for representing, organizing, and performing search, as for example branch-and-bound-based search, evolutionary methods, local optimization techniques, or tabu search, there are also many different concepts of how to make use of several processing units. In general, we can divide these concepts into two classes, namely *control strategy compliant*, or parallelization, approaches and those which are based on *concurrent*, or distributed, *search*.

A strategy compliant parallelization approach aims at speeding up a given traversal of the search space by making the generation of the next search state faster, i.e. the computations for the transition from one state to another are spread over the processing units. The approach is called strategy compliant because the computation follows the same course through the search space as in the sequential case, only faster. As a consequence, the parallel work is the same as the sequential work (provided that the search control is not dependent on the sequence future steps are evaluated). In contrast, in a concurrent search approach, each processing unit (often called an *agent*) works on different parts of the search space at the same time. Each agent may perform one or several transitions before communicating with other units, and this is repeated until the given search instance is solved. Due to the cooperation of the agents, the search space traversal may differ from a one-agent search (e.g., one concurrent agent may find the solution early), so that the parallel work may be less, with the potential for considerable super-linear speedups.

Since strategy compliant approaches expand nodes one after the other, with only each expansion computation done in parallel, there is less parallel grain size, so that these approaches usually are aimed for use on multi-processor machines with shared memory. Concurrent multi-agent search usually enjoys larger grain sizes, which opens a potential for distributing the agents over clusters of computers or even over the Internet. Naturally, concurrent search approaches can also be realized on multi-processor machines, but usually they can make no special use of a shared memory, because the agents work independently except for short spurts of communication.

For the purpose of this paper, we take the term *parallel* to mean *shared memory one search state parallel*, and we take *distributed* to mean *several search states parallel*. Consequently, we identify *parallel search* with shared memory parallelized one-agent search (which we implemented by multi-threading the agent on a parallel server), and we identify the terms *multi-agent search* and *distributed search* with concurrent multi-agent search (which we implemented by distributing the agents over a cluster).

Since, from the point of view of hardware architecture and configuration, parallelization and distribution approaches are on different levels, combinations of concrete concepts from each of these two classes should be possible, in order to make optimal use of existing hardware. In addition, for each instance of a search problem and each parallelization or distribution approach, there is a maximal number of processing units that can be *sensibly* used. For our applications, this number is not very large, say half a dozen in each case. Adding more units does not lead to speedups, often it even leads to slowdowns. By combining a parallelization and a distribution approach, more units can be sensibly used. Although the possibility of combining parallelization and distribution has often been mentioned in the literature, an experimental evaluation with regard to the benefits is missing. Even in [19], which can be seen as combining aspects of both general ideas, no experiments with respect to the contributions of the ideas to the speedups are reported.

In this paper, we characterize more formally the distinction between parallelization and distribution of search. Then we present the results of the combination of a parallelization approach to completion based equational theorem proving (PaReDuX) with a general search distribution concept improving on the so-called competition approach of search agents (TEAMWORK).

The general idea of PaReDuX is to provide control strategy compliant parallel term completion procedures which have efficient multi-threaded implementations on current shared memory multiprocessors [5, 6, 1]. From the viewpoint of search, this concept parallelizes individual transitions and, moreover, evaluates all possible successor states in parallel (by generating the relevant parts of these states and measuring these parts according to the control strategy of an agent). The general idea of TEAMWORK [7] is to have a set of search agents that work on the given problem instance using different control strategies. Periodically, the agents assess the results they have produced so far, determine the agent with the greatest success, and use all results of this agent together with selected results of all other agents to produce a new start state for all agents.

When we apply the general TEAMWORK approach of cooperating concurrent agents to the term completion domain and combine it with the multi-threaded agents of PaReDuX, then the TEAMWORK agents, resp. the transitions between states they perform, are parallelized, resulting in a search system employing *multiple parallel agents*. Our experiments in the domain of equational deduction show that the gains which each single concept is able to produce are almost multiplied when the combined search system is applied (which is the best that can be even theoretically achieved).

This paper is organized as follows: In Section 2, we formally characterize first knowledge-based search and then distributed, resp. multi-agent, search and parallel search. In Section 3, we focus on our parallel search agents, again by first presenting the basic search model on which our distribution and parallelization is based and then introducing the concrete distribution and parallelization concepts we combine. Section 4 presents the case study in automated deduction that evaluates our combination method. We conclude with a few remarks on future work.

2 Characterizing parallel and distributed search

We provide some terminology and definitions that allow us to make a distinction between parallelization and distribution approaches. We also briefly sketch the different ideas that are used in developing specific parallelization and distribution concepts. For further details, see [8].

2.1 Knowledge-Based Search

Knowledge-based search algorithms differ in the data structures used, the search spaces to traverse, the knowledge employed, or even the goals of the search. Usually, many algorithms can be used to solve a search problem, and often they behave rather differently for different instances of the search problem. But they all can be described in terms of the search states that can be generated, and the transitions between states that allow to traverse the possible search space.

Definition 2.1 (Search model, search process)
Let S *be a set of possible search states and* T ` SΘS *a set of transitions between states. Then* A = *(*S,T*) is a* search model. *Let further* Env *(the* environment*) be a set of values of (data) structures. The triple* P = *(*A, Env, K*) is a* search process, *if* K*:* SΘ Env ! S *is a function for which* K*(s,e)* 2 fs*'* j *(s,s')* 2 T g *holds for all s* 2 S *and e* 2 Env. K *is called the* search control *of the search process.*

The different search algorithms usually employ different search models, but they can also just differ in the search control functions. We will need the environment Env in order to model the outside influence on the behavior of a search process during the search (e.g., by incorporating information from other search agents, see Section 2.2).

Search models and processes are aimed at providing a mechanism that can be used for different instances of a search problem. Therefore, in order to employ a search process, we need a *search instance* providing it with a start state and a way to detect goal search states representing solutions to the given problem instance (represented by a predicate

G). Then the search process produces a sequence $(s_i)_{i \in \mathbb{N}}$ of search states, called a *search derivation*, where s_0 is the start state, $s_{i+1} = K(s_i; e_i)$, and $e_i \in Env$ is the environmental information accessible in state s_i.

For determining the efficiency of a search process (for a certain problem instance) the run time needed to produce a successful search sequence is the usual measure. If we want to improve this efficiency, often it is very useful to look at the produced search derivation and to compare it to other possible sequences of states leading from s_0 to a state representing a solution. Usually, in the produced derivation (s_i), many states, resp. transitions, can be identified that did not contribute to the found solution.

2.2 Multi-Agent Search

As already stated, the basic idea of distributed, or *multi-agent*, search is to have several search processes that run concurrently and that exchange information in order to help each other. In order to describe how and what the different agents communicate, we need a *communication structure* Kom that provides channels, i.e. data areas $D_1,...,D_l$, that can be used by the agents and that define type and structure of the information that can be put into them. Such a communication structure Kom has at each point in time an actual value *val* which is a tuple $(d_1,...,d_l)$ with $d_i \in D_i$.

We define *search agents* as search processes that have the additional ability to change the value of Kom by employing their communication functions $mes_i: S \times Kom \to Kom$. So, a search agent Ag_i is a triple (P_i,Kom,mes_i). Note that we can model receiving information by assigning Kom or parts of it as the environment Env_i of an agent.

Besides a communication structure and search agents, a multi-agent search system must also define how to generate search instances for all its search agents, and how to bring the results of the search agents together to get a solution for a given problem instance. These are the tasks of two additional agents, namely the start agent and the end agent. Thus we have the following definition:

Definition 2.2 (Multi-agent search system)
Let SP *be a search problem,* Kom *a communication structure and let* Ag_S *(*start agent*) and* Ag_E *(*end agent*) be agents and* $Ag_1,...,Ag_n$ *a set of search agents,* $Ag_i = (P_i,Kom,mes_i)$*. Further, let* $PU = \{pu_1,...,pu_z\}$, $z \geq n$*, be a set of processing units. Then the tuple* $MASS = (Ag_S,Ag_E,Ag_1;...;Ag_n, Kom,PU)$ *is called a* multi-agent search system *(or* distributed search system*) for* SP*, if the following holds:*

- *For each* $P_i = (A_i, Env_i, K_i)$ *there are* $D_{j1},...,D_{j_{l_i}}$ *in* Kom*, such that* $Env_i = (D_{j1},...,D_{j_{l_i}})$*.*
- *The agents can use the processing units to run their search processes and their* mes*-functions.*
- *The agent* Ag_S *produces for each problem instance* $Inst$ *of* SP *n pairs* $(s_{01}; G_1)$, ..., $(s_{0n}; G_n)$*, such that* $(s_{0i}; G_i)$ *is a search instance to* P_i.
- *If* $A_1,...,A_n$ *are the search derivations produced by* $P_1,...,P_n$ *to* $(s_{01}; G_1)$, ..., $(s_{0n}; G_n)$ *then* Ag_E *produces a solution to the instance* $Inst$ *of* SP*, if* $Inst$ *is solvable and all* Ag_i *terminated their search in a regular fashion.*

Our definition of a multi-agent search system describes the static aspects of such a system. In order to describe and analyze runs of a particular system for a certain problem instance, we have to know for each point in time the actual search states of all agents and the actual value of Kom. In practice, for generating a *search course protocol* we only need such snapshots for certain important points in time. These important points in time are defined by *time frames* and coincide with changes, either to the state of any agent or to the value of any data area in Kom. A more detailed account of the dynamics of a multi-agent search system is given in [8] and [9].

In the literature, there are three basic ideas on how to distribute search among the agents of a multi-agent search system. The *competition approach* has been proposed by several authors in different applications. It consists of running different search agents on the available processing units working on the same problem instance. In the pure version, there is no communication between the agents. *Improvements* on this approach, like the TEAMWORK concept [2, 7], allow for agents to periodically communicate selected results of different types. Then agents can use the results from the others to speedup their own search considerably.

By *dividing the given problem instance into instances of subproblems* that hopefully are smaller and easier to solve, it is very easy to employ several agents. Unfortunately, if such a division is possible, the resulting subproblem instances are often not independent of each other. So, such multi-agent search systems have to implement some kind of negotiation mechanism for the agents to produce solutions to their subproblems that are compatible with each other. Usually, agents inform other agents about conflicts and their particular problem instance, resulting in a more global view on the initial problem instance for the individual agents. For an example of a system employing this basic idea, see [24].

The *use of a common search state* for all search agents is inspired by properties of search models and processes. All the data structures usually employed as search states allow the concurrent execution of several transitions. For example, transitions in tree-based search models usually extend leaf nodes, and several of those extensions can be performed by different agents at the same time. This realizes a kind of partitioning of the search space between the agents. Agents may perform several transitions before they integrate their results (usually their actual state) into the common search state. The common state can either be stored centrally using shared memory to represent the communication structure, or it can be distributed among the agents resp. their environments (see, e.g., [16]).

It should be noted that the three basic ideas do not mutually exclude each other, so that multi-agent search systems are possible that include two or even all ideas. For example, a system based on dividing the problem into subproblems might use several agents for the same subproblem. Such a subteam might then cooperate by competition or by working on a common search state.

While the limit on the number of processing units that can be sensibly used is obvious for concepts dividing the problem instance into subproblems—if there are only 5 subproblems then using more than 5 units does not make sense–-this is not so obvious for the other two basic ideas. But just consider a problem instance and a search control of a search process that leads to a search derivation in which no unnecessary transition was performed and each transition needs the result of the previous transition to be applicable. If one of the agents of a multi-agent search system improving on the competition approach employs this search process then the other agents will not contribute anything to this agent solving the instance (because it already does an optimal search). So, just using this agent is enough. The same is true in a multi-agent search system based on a common search state: agents other than the one optimal agent just perform unnecessary transitions. Naturally, the chance for this to happen is very low, but dependencies between key transitions in addition to rather good search agents can for some problem instances result in rather small numbers of sensibly usable processing units, at least for homogeneous multi-agent search systems.

2.3 Parallel Search

The basic idea of parallel search is to employ several processing units in generating the next search state out of the actual one. This means essentially that performing *one* transition is done in parallel. In order to do so, we have to split performing a transition into several *tasks*. We need a set TA of tasks and a *task function* $tf\colon\ T' \to TA^*$ that assigns a set $\{ta_1,...,ta_k\}$ of tasks to each transition $t = (s;s') \in T' \subseteq T$. Since it might not be possible to divide each transition in T (except for the "trivial" division into a single task) we need the subset T' of T to indicate the really decomposable transitions. There might be tasks for a transition that require that other tasks have already been dealt with. In a parallel search, such dependencies are modeled by an *assignment function*.

Definition 2.3 (Parallel search system)
Let SP *be a search problem,* $A = (S,T)$ *a search model,* $T' \subseteq T$, *and* $P = (A, Env, K)$ *a search process for* SP. *Furthermore, let* $PU = \{pu_1,...,pu_z\}$ *be a set of processing units,* TA *a set of tasks and* tf *a task function to* A, T' *and* TA. *The tuple* $PSS = (A, P, TA, T', tf, PU)$ *is called a* parallel search system *for* SP, *if the following holds:*

- *For all instances* $Inst$ *of* SP *and their search instances* $(s_0;G)$ P *produces search derivations* $\mathcal{A} = (s_i)_{i \in \mathbb{N}}$, *such that* $(s_i,s_{i+1}) \in T'$.
- *There is an assignment function* $ass\colon\ T' \times TA \to PU \times \mathbb{N}$ *that assigns to each transition* $t \in T'$ *and each task* $ta \in tf(t)$ *a processing unit* $ass_{pu}(t,ta)$ *and a number* $ass_{st}(t,ta)$ *indicating when to start* ta *on the unit.*
- *If* $t = (s, s') \in T'$ *and* $tf(t) = \{ta_1;...;ta_k\}$ *then executing* ta_i *on processing unit* $ass_{pu}(t;ta_i)$ *at moment* $ass_{st}(t;ta_i)$ *for all* ta_i *results in generating s' out of s.*

So, a parallel search system generates the same search derivation $\mathcal{A}$ as the parallelized search process P would produce if not parallelized, provided that all transitions are in T' and the search control does not rely in any way on random factors or time dependent evaluations of transitions. Note that the assignment function is not fixed, in the same sense as we did neither fix the assignment of agents to processing units, nor the search course protocol in case of multi-agent search systems. Once again, for an analysis of parallel search systems snapshots of the system with respect to the points in time represented by $ass_{st}(t,ta_i)$ for all ta_i (and performed transitions t) are needed. We will not go into more detail here.

The parallel search systems in the literature can be located within a spectrum that ranges from systems whose task sets contain only tasks directly concerned with manipulating the search state to systems whose task sets contain only tasks that deal with evaluating all transitions that can be performed in the actual state (and one task for performing the selected transition). Task sets of the first kind are very dependent on details of the search model, and therefore on the particular application. Parallel search systems with task sets that mainly deal with providing the search control with selection information can often be described independently from the particular application, taking only into account the rules that define the possible transitions.

As the definition shows, there is a limit on the number of processing units that can be sensibly used for parallelizing a certain transition, namely the number of tasks. Usually this limit can already not be reached in reality, because some tasks depend on others. So, given the search derivation to a problem instance and the search process we parallelize, the highest number of sensibly usable processing units is at best the maximum of the highest numbers for all the transitions. It should be noted that for concepts with task sets centering on providing information for the search control usually in late stages of the search this highest number can become rather large whereas the other concepts tend to have limits that are more independent of the stage the search is in.

Note that the border between a multi-agent search system and a parallel search system is rather strict if the search model (and especially the definition of the transitions) is given, and not both ideas for utilizing several processing units are integrated into the system. But without a clear search model the border is not very well defined. For example, if we look at a set-based search model and a genetic algorithm search process, we could describe a transition using the notion of a generation, i.e. by generating in each step a certain number of new individuals while eliminating the same number of old ones. An alternative would be to consider the creation and elimination of one individual as a transition. A search system utilizing several processing units in order to compute a new generation in parallel can be classified as a parallel search system if we use the first definition for a transition, while it would be a multi-agent search system (based on a common search state) for someone using the second definition.

3 Parallel search agents

If we look at our definitions in Sections 2.2 and 2.3 then it looks as if each parallelization concept could also be applied to the individual agents of a distributed search system. But this need not be the case as, for example, the communication function of a search agent influences the behavior of other agents via their environments, and this is not dealt with by the tasks of a parallel search system. Therefore we cannot always expect that the combination of a distribution and a parallelization concept results in a *multiplication* of speed-up factors.

A general problem with all parallelization approaches is that performing a single transition may not require enough effort to justify a parallelization. This obviously depends on the search model and the concrete search problem, but there is no guarantee that a parallelization approach achieving satisfactory speed-ups can be found. Another general problem with heterogeneous multi-agent search approaches, i.e. approaches with agents using different search models, is that several parallelization approaches are needed, one for each of the different search processes used.

If the search control of at least one agent relies on decisions that may be time-dependent (FIFO, for example) then small differences in the relative execution progress of the agents can cause quite different time frames. And, unfortunately, the parallelization of agents very often produces such small differences. Even distribution approaches based on a common search state, for which this problem is not very grave, suffer from another problem, namely the already mentioned fuzzy borders between multi-agent and parallel search. If the search model used for the distribution approach differs only a little bit from that of the parallelization approach, then the result of a combination is often not a multiplication, but just an addition of the speed-up factors. And the combination might not overcome the problem of the maximal number of processing units that can be sensibly used for a particular problem instance.

So, although at a first glance combining multi-agent and parallel search seems to be easy and straightforward, the success cannot be guaranteed. Therefore approaches have to be found that really can be put together by parallelizing the agents and then an experimental evaluation is necessary to prove that the combination is good and achieves the multiplication of the speed-ups. In the following, we will first present the basic search model, then the two approaches we have selected to combine and finally our resulting approach for multiple parallel search agents.

3.1 The Basic Search Model

Both, TEAMWORK and PaReDuX, were designed for set-based search models. Thus, a search state consists of a collection $\{f_1, \ldots, f_k\}$ of *facts*, and each transition can add and/or remove facts from the current search state in order to produce the successor state. Each transition is composed of three steps:
(i) selection of a yet unhandled fact,
(ii) deduction of new facts by combining known facts with the selected one, and
(iii) simplification of facts.
Step (ii) helps in evaluating the possible future transitions by computing part of their results. To keep track of which facts have already been processed, the agents distinguish between *active* and *passive* facts. Only active facts are considered during the deduction process; all newly generated facts and facts from the original problem are initially passive (activation by step (i)).

As the size of the search space crucially depends on the selection strategy employed in step (i), choosing the next fact carefully (according to some elaborate heuristic built into the search control) can be worthwhile. Usually, there are several heuristics to do this. They define different search processes that all evaluate all passive facts to select the best one.

3.2 The TEAMWORK Method

In TEAMWORK (see [7], [11]), the agents performing the search processes are called *experts*. They are homogeneous, and thus use the basic search model with different controls. The communication functions of n agents Ag_i work on a communication structure $Kom = (D_{Newstate}, D_{Res,1}, \ldots, D_{Res,n}, D_{Pu,1}, \ldots, D_{Pu,n}, D_{con})$.

The data areas $D_{Newstate}$ and D_{con} are shared by all search agents, and additionally, agent Ag_i has data area $D_{Res,i}$ and $D_{Pu,i}$ in its environment.

The start agent transforms the given problem instance into a search instance that is given to z of the n search agents selected as the start team. In addition, it selects one of these search agents as first *supervisor*, a role that this agent's communication function will play until a new supervisor is chosen in a team meeting. Such a team meeting takes place after a given period of time. In the mean time, all search agents act solely as experts, so that no information exchange takes place. In a team meeting only the communication functions are used (and the search process of the agent selected as new supervisor).

A team meeting starts with the communication functions of all active agents acting as *referees* by computing a *measure of success* for their agents and by selecting their best new facts. Both types of information naturally depend on the search problem. The measure of success is then put into the D_{Res}-area of the agent that is the supervisor at this moment. The communication function of this agent then acts in the supervisor role, compares the measures of success of all agents, and selects the agent with the best measure as next supervisor. This agent (and all other ones) are informed about this change via the D_{con}-area.

All agents put their selected facts into the D_{Res}-area of the new supervisor. Then the supervisor generates the next search state of its agent by adding all the selected facts from the others to its actual search state (and performing the necessary additional actions of an expert). This new state will be the starting point for all agents and therefore the supervisor puts it into the $D_{Newstate}$-area. The supervisor is also responsible for selecting the next team, i.e. the agents that will be active after the meeting is over. To do this, it uses some general knowledge about the agents and the D_{con}-area consisting of reports about the team meetings containing the measures of success of all agents. Agents that performed badly for a while are exchanged by hopefully better agents. All selected agents (the supervisor is always among them) are assigned a different element of PU in their D_{Pu}-areas, while the other agents are assigned "NONE" in their areas. The first transition the active agents perform as experts is to set their states to the new start state and then they continue their work until the next meeting.

3.3 The PaReDuX Approach

Following our definitions of Section 2.3, the parallelization approach represented by PaReDuX [5, 6, 1] can be described as follows. The set TA of tasks to be performed for a transition consists of the following three types:

ta_1**:** Selection and activation of a passive fact f_{new}.

$ta_2(f_1, f_2)$**:** Deduction of new facts by combining two active facts f_1 and f_2.

$ta_3(f)$**:** Simplification of fact f, possibly inactivating or deleting it.

Obviously, task ta_1 has to be performed first, resulting in the selection and activation of f_{new}. Then consequences derivable with f_{new} are generated in parallel, where for each active fact f_a a task $ta_2(f_{new}, f_a)$ is employed. The transition is completed by simplifying all facts, again in parallel, using task $ta_3(f)$ for each fact f. We thus get the following task function:

$$\begin{aligned} tf((s_i, s_{i+1})) = \{ & ta_1\} \cup \\ & \{ta_3(h) \mid h \in F'(s_i)\} \cup \\ & \{ta_2(f_{new}, g) \mid g \in AF(s_i) \cup \{f_{new}\}\} ; \end{aligned}$$

where $AF(s_i)$ and $F'(s_i)$, respectively, denote the set of active facts in state s_i and the set of active or passive facts after having finished all ta_1 and ta_2 tasks initiated in state s_i. s_{i+1} denotes the set of all remaining simplified facts delivered by the ta_3 tasks.

For the processor assignment function let us assume a transition $t = (s_i, s_{i+1})$, a set of active facts $AF(s_i) = \{g_1, \ldots, g_{m-1}\}$ and a set of facts $F'(s_i) = \{h_1, \ldots, h_n\}$ after having finished ta_1 and ta_2. Furthermore (for uniformity), we set $g_m = f_{new}$, and thus get

$ass(t, ta_1) = (pu_1, 0)$;
$ass(t, ta_2(f_{new}, g_j)) = (pu_j, 1)$ for $1 \le j \le m$;
$ass(t, ta_3(h_k)) = (pu_k, 2)$ for $1 \le k \le n$;

where we have made the assumption, that we have a set of (virtual) processors of size at least $\max(m, n)$ at our disposal.[1].

3.4 TEAMWORK-PaReDuX

If we combine TEAMWORK and PaReDuX then the experts represent the search process that PaReDuX is aimed at parallelizing. So, after the team meeting and after setting the actual search state to the value of $D_{Newstate}$, PaReDuX is used to perform the subsequent transitions of an agent until it has to act as referee. Note that this is possible because TEAMWORK does not allow any communication between agents acting as experts so that we do not have to develop any "communication tasks" for PaReDuX.

But the experts are not the only use of PaReDuX that is possible. When the new supervisor integrates the selected facts of the other agents into its search state, this is essentially a transition, only that f_{new} does not have to be selected out of the set of passive facts but it is one of the selected facts. So, there are still the ta_2 and ta_3 tasks to perform that are exactly the tasks that are parallelized by PaReDuX. In TEAMWORK-PaReDuX this possibility to use

[1]This assumption is even made in the implementation, where it is resolved by the `VSThreads` middleware [13, 14, 15].

the PaReDuX approach is also employed, so that not only in the time between team meetings PaReDuX is used to speed-up the system, but also during the team meetings the processing intensive part of the supervisor's work is parallelized (which keeps the percentage of time spent for overhead computations nearly the same between TEAMWORK alone and TEAMWORK-PaReDuX).

4 A case study: Experiments in Automated Deduction

Both, PaReDuX and TEAMWORK, have already been applied separately to parallel and distributed equational deduction using unfailing completion (see [18, 10]). Therefore, and because equational deduction is a rather hard search problem, we also have chosen this application for their combination. The goal of equational deduction is to show that a *goal equation u=v* is the logical consequence of a set E of equations, $E = \{s_1 = t_1, \ldots, s_n = t_n\}$. For a general discussion of parallel completion see [1].

Unfailing completion [3] solves this search problem using a set-based search model, in which a state consists of three components: a set of valid rules[2] and equations, a set of unhandled facts called *critical pairs*, and the goal to be proved in normal form, i.e. simplified with respect to the directed and undirected equations. In terms of Section 3.1, the equation set is the set of active facts, whereas passive facts are critical pairs. Each transition thus first selects a critical pair and transforms it into a rule or equation, depending on whether or not it can be oriented. Then new critical pairs are generated, which are finally simplified by computing their normal forms. Simplification also encompasses interreduction of the rule and equation sets, as well as the elimination of redundant equations and critical pairs.

For our case study we used the PaReDuX system as the implementation of the single parallel search agent, we implemented several of the special heuristics that result in different and well cooperating agents from the TEAMWORK point of view within PaReDuX, and we put these parallel agents into the TEAMWORK framework represented by the TWlib class library (see [12]). For our experiments with the resulting system we selected examples from the TPTP problem collection [22]. Two further problems (LUKA3 and RA007[3]) were additionally included. Table 1 reports on the results of these experiments. The reported examples represent problems that are hard (around 100 seconds of run time or more) for the best sequential control strategy in PaReDuX. The time needed by the best strategy is reported in the column **B. Seq.**.

[2]In this setting, directed equations are called *rules*.

[3]LUKA3 is taken from [10], RA007 proves that A ˘ A fi 1 in all relation algebras using the Tarski/Givant axiomatization [23].

This best control strategy was also used by the parallel runs of PaReDuX, which are reported in column **Par.** in the table. TEAMWORK employed the best two control strategies for each example and the run times are reported in column **Distr.**. The team of parallel agents consisted of the same two strategies, but now parallelized. The run times of the parallel agents are in column **Par. Ag.**.In order to get a clear idea of how well the concepts work together, we did not make use of the reactive planning possibilities of TEAMWORK and also did not choose individual cycle times for the different examples. Instead, a fixed cycle time of 10s in the case of sequential agents and 2.5s for parallel agents was used. The experiments were performed using two Sun E450 machines with four processors each. The control strategies available have been Occnest, AddWeight, MaxWeight and Goal-in-CP (see [10], [11]). All four processors of the machines were used in the parallel runs (i.e. **Par.** and **Par. Ag.**). For the sequential agent runs (**B. Seq.** and **Distr.**), only one processor of each machine was used.

As the right part of Table 1 shows, the speed-ups obtained by using the parallelization method (column **Par.** under speed-up factors) only cover a range between a little better than no speed-up (i.e. 1.36) and the theoretically best possible factor of 4 (with 3.36 being the highest actually achieved). Due to the synergetic effects of TEAMWORK, the speed-ups achieved by distribution cover a much larger spectrum (up to a factor of nearly 260 with only two machines) as column **Distr.** shows. But, as BOO022-1 shows, TEAMWORK is not always successful (at least with the reported way of choosing the team) and might even slow down the search. The examples BOO007-4, LUKA3 and RA007 also show that TEAMWORK is capable of solving examples that no single agent could cope with (we imposed a limit of 2400s on the system).

Column **Par. Θ Distr.** represents the theoretical outcome of combining the parallelization and distribution approaches, namely the multiplication of the speed-ups of the two columns to the left of it. In the rightmost column (i.e. **Par. Ag.**) we report the empirically determined speed-ups of our system of cooperating parallel search agents. For several of the examples we nearly reached the optimum and for four examples we were even better than the theoretical best. For the three examples requiring the TEAMWORK concept to be solved, parallelizing the agents leads to a multiplication of the speed-up with between 2.63 and 4.09, well within the range of the other results.

The fact that we have better practical results than the theoretically predicted outcome needs an explanation. We found it by looking at the number of transitions made by the individual agents during a working cycle. In order to produce facts needed by other agents to find a proof, agents have to perform certain numbers of transitions according to their control. It would be optimal if team meetings hap-

Table 1. Experimental Results

Problem	Run times				Speed-up factors			
	B. Seq.	**Par.**	**Distr.**	**Par. Ag.**	**Par.**	**Distr.**	**Par. Θ Distr.**	**Par. Ag.**
BOO002-2	411.95	154.02	194.80	76.19	2.67	2.11	5.63	5.41
BOO007-4	-	-	554.31	163.68	-	-	-	-
BOO016-2	96.20	70.98	5.32	2.73	1.36	18.08	24.59	35.24
BOO022-1	148.50	56.34	226.44	57.08	2.64	0.66	1.74	2.60
COL003-12	364.89	111.21	262.08	86.56	3.28	1.39	4.60	4.21
COL003-20	285.48	93.65	1.10	0.41	3.05	259.53	751.57	696.29
GRP002-4	205.93	83.36	14.00	6.22	2.47	14.71	36.33	33.11
GRP119-1	867.70	258.45	423.86	99.81	3.36	2.07	6.96	8.69
GRP122-1	752.73	218.73	155.31	49.23	3.32	4.85	16.10	15.29
GRP175-3	1422.15	555.19	63.70	29.04	2.56	22.33	57.17	48.97
GRP175-4	464.33	163.02	243.50	87.61	2.85	1.91	5.44	5.30
LUKA3	-	-	135.73	33.17	-	-	-	-
RA007	-	-	67.22	25.58	-	-	-	-
ROB004-1	1683.82	670.16	40.73	14.60	2.51	41.34	103.76	115.33

pened immediately after the production of such facts, because then the other agents can make immediate use of them. But it is also possible that such facts are produced immediately after a team meeting, so that the whole time until the next team meeting is “lost”, because agents cannot produce the follow-up facts. This timing between agents can change due to parallelization. So, the timing of team meetings has a certain influence on the time needed to solve a problem, and parallelizing the agents can change this timing a little bit compared to the sequential agent. This can be both positive and negative, as our experiments have shown. In order to give an impression of the reliability of the combination, we have chosen to use fixed cycle times in Table 1.

As presented here, we assume that the distributed architecture consists of an essentially homogeneous cluster (processors of similar speed) with uniform network speed, such as our cluster of multiprocessor SUN ES450 servers. If one of the agents is delayed by a slow system or network, then either the team meetings can be delayed, or we accept that the contribution by the agent will be less than the contribution of the others. As our explanation in the previous paragraph showed, we have chosen the latter to keep the scheduled team meetings. Different numbers of physical processors can be evened out by user level threads systems supporting symbolic computation [13, 14, 15]. On inhomogeneous processor and operating system architectures our DOTS distribution middleware can provide a logically homogeneous programming environment based on asynchronous remote procedure calls; DOTS has been successfully applied to distributed algebraic computation and satisfiability checking [4, 20, 21].

5 Conclusion and Future Work

Based on basic definitions and coarse classifications of distribution and parallelization approaches for knowledge-based search, we presented experimental evidence that loosely coupled parallel agents, i.e. distributed search based on improvements of the competition approach combined with parallel search mostly based on evaluating all transitions in parallel, achieve a multiplication of the speed-ups resulting from distribution and parallelization alone. This makes this combination of distribution and parallelization very interesting for usage in systems aimed at utilizing networks of homogeneous multi-processor computers, because of the good use of the characteristics of such a hardware platform.

Although we presented only a single case study, our experimental results show the anticipated problems of the combination of the general types of distribution and parallelization, i.e. problems with timing between agents. But they also indicate that these problems have only a minor influence on the performance of the combined search system. For other possible combinations this might not work out as well. Future work could be directed to using our parallel search agents in other application domains, using other cooperation concepts based on the competition approach, and trying out parallelization approaches more towards constructing the new state in parallel.

Acknowledgements

This work is based in part upon research funded by the German National Science Foundation DFG under grants Ku 966/4-1 and Av 10/7-1 within the special purpose program “SPP Deduktion.” One of the ES450 servers is a gift to the

University of Tübingen under Sun's Academic Equipment Grant program.

References

[1] B. Amrhein, R. Bündgen, and W. Küchlin (1998). Parallel completion techniques. In M. Bronstein, J. Grabmeier, and V. Weispfenning, (eds), *Symbolic Rewriting Techniques*, volume 15 of *Progress in Computer Science and Applied Logic*, Birkhäuser, pp. 1–34.

[2] J. Avenhaus and J. Denzinger (1993). Distributing equational theorem proving. Proc. RTA'93, LNCS 690, pp. 62–76.

[3] L. Bachmair, N. Dershowitz, and D. Plaisted (1989). Completion without Failure. *Coll. on the Resolution of Equations in Algebraic Structures*, Austin (1987), Academic Press.

[4] W. Blochinger, W. Küchlin, C. Ludwig, and A. Weber (1999). An object-oriented platform for distributed high-performance symbolic computation. *Mathematics and Computers in Simulation*, 49:161–178.

[5] R. Bündgen, M. Göbel, and W. Küchlin (1996). Strategy compliant multi-threaded term completion. *J. Symbolic Computation*, 21(4–6):475–505.

[6] R. Bündgen, M. Göbel, W. Küchlin, and A. Weber (1998). Parallel Term Rewriting with PaReDuX. in Bibel, Schmitt (eds.): Automated Deduction–A Basis for Applications, Vol. II, Kluwer, pp. 231–260.

[7] J. Denzinger (1995). Knowledge-Based Distributed Search Using Teamwork. Proc. ICMAS-95, AAAI-Press, pp. 81–88.

[8] J. Denzinger (2000). Distributed Knowledge-based Search. Habilitationsschrift, Computer Science Department, University of Kaiserslautern.

[9] J. Denzinger (2000). Conflict Handling in Collaborative Search. In Tessier, Chaudron, Müller (eds.): Conflicting Agents: Conflict management in multi-agent systems. Kluwer, pp. 251–278.

[10] J. Denzinger and M. Fuchs (1994). Goal Oriented Equational Theorem Proving using Teamwork. Proc. 18th KI-94, LNAI 861, pp. 343-354.

[11] J. Denzinger, Mark Fuchs, and Matthias Fuchs (1997). High Performance ATP Systems by Combining Several AI Methods. Proc. IJCAI-97, Morgan Kaufmann, pp. 102–107.

[12] J. Denzinger and J. Lind (1996). TWlib - a Library for Distributed Search Applications. Proc. ICS'96-AI, pp. 101–108.

[13] W. Küchlin (1992). The S-threads environment for parallel symbolic computation. In R. Zippel (ed), *Computer Algebra and Parallelism*, Springer LNCS 584, pp. 1–18.

[14] W. Küchlin and N. J. Nevin (1991). On multi-threaded list-processing and garbage collection. Proc. Third IEEE Symp. on Parallel and Distributed Processing, IEEE Press, pp. 894–897.

[15] W. Küchlin and J. A. Ward (1992). Experiments with virtual C Threads. Proc. Fourth IEEE Symp. on Parallel and Distributed Processing, IEEE Press, pp. 50–55.

[16] V. Kumar, K. Ramesh, and V. Nageshwara Rao (1988). Parallel Best-First Search of State-Space Graphs: A Summary of Results. Proc. AAAI-88, AAAI Press, pp. 122–127.

[17] S.E. Lander and V.R. Lesser (1992). Customizing Distributed Search Among Agents with Heterogeneous Knowledge. Proc. 1st Intern. Conf. on Information and Knowledge Management.

[18] P. Maier, M. Göbel, and R. Bündgen (1995). A Multi-Threaded Unfailing Completion. Technical Report 95-06, Wilhelm-Schickard-Institut, Universität Tübingen, Germany.

[19] P. Nangsue and S.E. Conry (1998). Fine-Grained Multiagent Systems for the Internet. Proc. ICMAS'98, IEEE Press, pp. 198–205.

[20] R.-D. Schimkat, W. Blochinger, C. Sinz, M. Friedrich, and W. Küchlin (2000). A service-based agent framework for distributed Symbolic Computation. Proc. 8th Intl. Conf. on High Performance Computing and Networking Europe, HPCN 2000, Springer LNCS 1823, pp. 644–656.

[21] C. Sinz, W. Blochinger, and W. Küchlin (2001). PaSAT - parallel SAT-checking with lemma exchange: Implementation and applications. In H. Kautz and B. Selman, editors, *LICS'2001 WS on Theory and Applications of Satisfiability Testing (SAT'2001)*, volume 9 of *Electronic Notes in Discrete Mathematics*, Elsevier.

[22] G. Sutcliffe, C.B. Suttner, and T. Yemenis (1994). The TPTP Problem Library. Proc. 12th CADE, LNAI 814, pp. 252–266.

[23] A. Tarski and S. Givant (1987). A Formalization of Set Theory without Variables, vol 41 Coll. Publ. Amer. Math. Society.

[24] M. Yokoo, E.H. Durfee, T. Ishida, and K. Kuwabara (1992). Distributed constraint satisfaction for formalizing distributed problem solving. Proc. 12th IEEE Conf. on Distributed Computing Systems, pp. 614–621.