A Service-Based Agent Framework for Distributed Symbolic Computation

Ralf-Dieter Schimkat*, Wolfgang Blochinger, Carsten Sinz**, Michael Friedrich, and Wolfgang Küchlin

Symbolic Computation Group, WSI for Computer Science
University of Tübingen, Sand 13, D-72076 Tübingen, Germany
http://www-sr.informatik.uni-tuebingen.de

Abstract. We present OKEANOS, a distributed service-based agent framework implemented in Java, in which agents can act autonomously and make use of stationary services. Each agent's behaviour can be controlled individually by a rule-based knowledge component, and cooperation between agents is supported through the exchange of messages at common meeting points (*agent lounges*). We suggest this general scheme as a new parallelization paradigm for Symbolic Computation, and demonstrate its applicability by an agent-based parallel implementation of a satisfiability (SAT) checker.

1 Introduction

Symbolic Computation comprises Computer Algebra on the one hand, and Computational Logic on the other hand. It is increasingly acknowledged that Symbolic Computation will play an essential role in future problem solving environments (PSEs). Where it is theoretically applicable and practically feasible, it yields answers with the quality of mathematical proofs. Moreover, answers may be given at very high levels of abstraction, containing symbolic parameters.

Due to the very high level of abstraction at which Symbolic Computation takes place, it is characterized by high computational demands and highly irregular and data dependent control flows. At the same time, there is practically no hardware support (e.g. for big integer arithmetic). However, there is a great potential for parallelization. Parallel Symbolic Computation is therefore an interesting research topic. Since automatic parallelization of Symbolic Computation algorithms is rather difficult due to highly dynamic data structures and irregular control flow, our approach is to investigate middleware architectures and associated programming paradigms which will support human programmers in parallelizing their sequential code base. As we move towards more processors and even less homogeneous and larger scale networks, it is time to explore more flexible and loosely coupled parallel architectures.

In this paper we explore the use of agent based middleware with a parallelization paradigm that is based on a stricter separation of the sequential code base from aspects

* Supported by *debis Systemhaus Industry.*

** Partially supported by *Deutsche Forschungsgemeinschaft (DFG)* under grant Ku 966/4-1, and partially supported by *debis Systemhaus Industry.*

dealing with the parallelization, such as making parallelization decisions, communicating, and distributing and balancing computational load. Our OKEANOS middleware provides an infrastructure for *mobile agents* which may access computational *services* and may communicate by passing messages in KQML [5], using Remote Method Invocation only as a general transport layer. The agents themselves are implemented in Java [9] and may contain rule-based knowledge interpreted by the Java expert system shell Jess [6], and they are supported with distributed information about the computational load provided by agent-related middleware services.

The agent based style of parallelization is much less synchronized than a client / server style. In our agent based approach, the sequential code is left largely intact, and the parallelization aspects are factored out to the level of agents. In this sense it bears some resemblance to Aspect-oriented programming [14].

We evaluate the OKEANOS system with a parallel satisfiability (SAT) checker for Boolean logic. Although theoretically intractable, practical advances have opened up important applications to SAT checkers, such as hardware verification or scheduling problems [1, 12]. Due to the large volumes of data in industrial applications, any significant advance in the speed of SAT checking is likely to open up new classes of practical problems. Our implementation is the computational core of an industrial system which we are developing for DaimlerChrysler AG. Its application will be in checking the consistency of symbolic product model data bases for trucks and passenger cars [15].

2 OKEANOS Agent Framework

2.1 Introduction

OKEANOS is an agent-based middleware framework for processing and distributing complex computational tasks. The primary goals of OKEANOS are to hide the process of task distribution within a computer network and to offer high-level interfaces for integrating distributed symbolic applications. The complexity of how, where, and why, the computional processes are running should be hidden behind uniform middleware-related interfaces. The notion of a service within OKEANOS establishes a middleware infrastructure where each participant's behaviour, i.e. a problem solver agent, is characterized either by making use of a collection of available services, or by offering a certain service itself. The trading and consuming of services in OKEANOS is done by exchanging language-neutral messages between participants and service providers.

In order to meet the challenges of scalability in large computer networks such as the Internet, the service management itself is strictly decoupled from the utilization of services. Thus, whereas all participants in OKEANOS can communicate asynchronously with each other or with service providers by exchanging messages, a distinct management process is responsible for keeping the service infrastructure consistent. It is up to the middleware to provide appropriate communication facilities as the *facilitators* [7] for exchanging messages between participants in a general manner.

While OKEANOS is completely implemented in Java [9], the implementation of potential service providers is not necessarily bound to Java. For example, they can be integrated into OKEANOS either by using the Java Native Interface (JNI) or by using

a client/server like communication mechanism. In the latter one, the service provider would act as a manager which forwards service requests to a remote server. These kind of requests can be transferred by applying the paradigm of Remote Procedure Call or by using ordinary socket-based communication styles. In Section 3 a symbolic computing engine, a propositional satisfiability checker, is integrated into OKEANOS as a calculation service by interfacing to its C++ implementation using JNI. Thus, OKEANOS forms the core component of a heterogeneous, distributed infrastructure which is characterized by services implemented in a wide range of programming languages. Moreover, by choosing Java as the implementation language, all OKEANOS components can be used uniformly on several platforms, as described in Section 3.5.

In the following sections we give a description of the main design principles of OKEANOS.

2.2 Framework Components

In OKEANOS there are two basic components: lounges and agents. A lounge is located at a host machine within a computer network like the Internet, providing a service-centered processing environment for agents. It offers the computing infrastructure for agents on top of the underlying host machine. A lounge supports the management of various kinds of services like a directory and messaging service. A new lounge is added to OKEANOS by registering itself to a well-known registry server called observer. The observer maintains a table of all registered and active lounges. By registering at the observer, the new lounge receives a list of callback entries of all other lounges. It is up to the directory services to propagate the new lounge and to synchronize its directory entries.

To minimize the complexity of an agent-based middleware framework, there is a strict distinction between two kinds of agents in OKEANOS: stationary and mobile agents. A stationary agent is located at one single lounge which it can not leave. Its primary task is to implement and to provide services to other agents like querying the lounge about locally available and registered application services. The complete communication infrastructure in OKEANOS is designed as a general service which is managed by stationary agents called portal agents. In contrast to stationary agents, mobile agents (MA) are characterized by their potential to move among lounges transparently by using the appropriate communication service available at the local lounge. Since the communication infrastructure is hidden behind the notion of a service, the agent logic of MAs has only to deal with the core application-related logic which facilitates the development of MAs as discussed in Section 3.3 and as illustrated in Figure 1a. When moving between lounges, MAs keep their state and behaviour. The lounge at the destination host is responsible for restarting the transferred MA correctly.

Lounges and agents are designed as framework components to establish a high-level, semi-complete software architecture which can be specialized to produce custom applications [4]. They form an abstract set of classes and interfaces among them which, taken together, set up a generic architecture for a family of related applications [10, 11]. In order to design an agent-based distributed application in OKEANOS, the application requirements have to be clearly identified in order to distinguish between such functionalities which can be addressed at the framework level, i.e. at the service level of

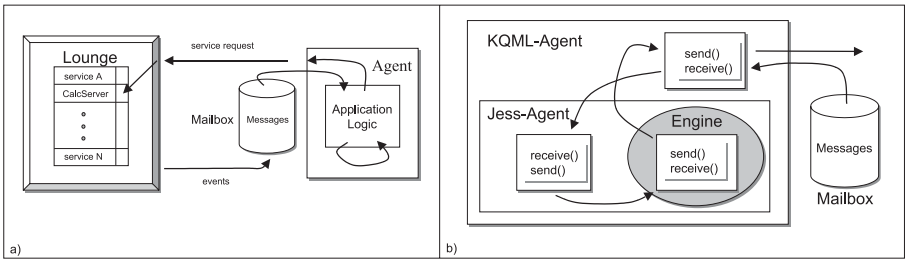

Fig. 1. a) Interaction between application logic and services located at lounges; b) Flow of control within a Jess-based agent

OKEANOS, and the application level. The benefit of shifting requirements to the more abstract framework level is in facilitating the design and implementation of the remaining application logic. Furthermore it increases the reuse by other applications which in turn eases their development. Among the patterns discussed in [13] are the *Layered Agent*, *Broker*, and *Reasoner* pattern which form the basis for the design of agents in OKEANOS.

2.3 Communication Concepts

Since interaction between agents can exhibit a complex structure, the paradigm of remote method invocation is too restrictive. Therefore communication within OKEANOS is characterized by the concept of message passing using the Knowledge Query Manipulation Language (KQML) [5]. KQML provides a general message container for possibly different types of message contents. It communicates dedicated properties about the transferred content of a message, rather than communicating within some special language. A mailbox is assigned to each agent which is basically a buffer for incoming and outgoing messages, as depicted in Figure 1a. The mailbox supports asynchronous communication between agents and lounges since it does not interfere directly with the agent's flow of control. Each agent is responsible for the management of its mailbox by itself. In OKEANOS, Remote Method Invocation (RMI) serves as a transport layer for messages which are sent between lounges. Since the communication interface between lounges does not change over time regardless of the number of lounges and type of messages, no additional communication-related code has to be provided [18], i.e. such as compiled static interfaces like stubs and skeletons. Instead, KQML messages are used as generic interfaces to initiate remote agent requests and to transfer mobile agents.

Using messages as the exclusive communication medium provides an extensible and adaptable approach to an uniform management of a possibly wide range of different agent communication interfaces, as discussed in [18]. As far as implementation issues are concerned we need only one communication interface within the entire distributed system which reduces potential incompatibilities between agents in OKEANOS.

Nevertheless, in OKEANOS other communication paradigms can be integrated transparently and made available to the entire system by wrapping them into a service. The interaction of a MA is restricted to the local lounge. If it wishes to communicate with

agents or services at remote lounges, it has to signal its desire to the local lounge by sending an appropriate message to the portal agent which in turn forwards the agent's requested actions.

2.4 Directory Service

The directory service in OKEANOS is designed as a distributed bulletin board for announcing application services. It is kept up to date by using so-called Updater Agents (UA). An UA is small in code size (about 1 KByte). It updates the directory of each lounge. From a lounge's point of view, UAs are handled with high priority since their process of updating each directory is very short in time.

One directory service is assigned to each lounge. There are two types of directory entries: global and local entries. A local entry describes a service which is offered only locally by a service provider at the respective lounge. A global entry specifies a service which is available at the respective lounge and can be accessed by other lounges and agents, respectively. The global part of the directory service is replicated at each lounge in order to minimize remote communication overhead: An agent's service request can be processed locally without remote communication to other lounges. Therefore it caches globally available services at each lounge and speeds up the lookup of services by reducing the overhead of remote communication tremendously. This caching mechanism aids the scalability of the number of lounges and agents in OKEANOS because of the decreased remote communication as opposed to a more central directory server.

The global part of the directory service serves as a snapshot of the state of all services of the distributed system. It is contacted by agents to make application-related decisions which need to have the most up-to-date information of the overall distributed system. The updating mechanism in OKEANOS is characterized by its simplicity and robustness. However, there is a small window of inconsistency of each globally replicated directory entry since the service update is processed in a decentralized manner avoiding a central manager for coordinating and propagating the updates appropriately. To eliminate potential bottlenecks in large-scale systems, the design of OKEANOS does not insist on the overall consistency of the distributed directory services at all times. If an MA contacts such a service at a remote lounge which is actually not available any more, it simply has to adjust its behaviour to the new state of the system. For example, this MA can go back to the lounge it has come from. By this, one has a tradeoff between a strict and a weak consistent point of view of the distributed directory service. Since OKEANOS is a service-centric system with autonomous MAs acting on their own, it is not predictable where and how agents interact with each other [1]. The primary goal of the distribution of the global directory entries is to provide a service for registering and propagating all types of services in a simple, robust and scalable manner.

[1] In Section 3.5 each performance measurement for a given number of particating lounges ends up with a slightly different runtime because of the unpredictable behaviour of the agents and the non-deterministic nature of the SAT problem.

2.5 Intelligent Agents

Agents in OKEANOS are either pure KQML-agents or Jess-agents. Jess [6] is a rule engine and scripting environment written entirely in Java. Agents based on Jess have the capacity to reason, using knowledge which is supplied in the form of declarative rules. Jess-agents are integrated into OKEANOS in a natural way in that declarative rules are wrapped into KQML messages and vice versa, as shown in Figure 1b. A Jess-agent consists of two parts. The KQML part of the agent is responsible for managing incoming and outgoing messages and for transforming KQML messages into Jess-based declarative rules. Then, after the Jess engine has interpreted the incoming rules, new facts are added (`assert`) to the knowledge base, or existing facts are retracted (`retract`) from it, depending on the content of the incoming messages. The resulting knowledge base of a Jess-agent determines its state and is characterized by its dynamic change. It also depends directly on the processing environment, i.e. the lounge where it is anchored. Therefore, integrating Jess into OKEANOS provides an infrastructure for developing agents which have facilities related to the area of artificial intelligence.

Jess-agents encourage the design and implementation of autonomous MA which adapt to changes of the location of lounges, of the global availability of services, and of application-related knowledge in a rule-based fashion. In contrast to Jess-agents, pure KQML-agents pursue an object-oriented mechanism for managing their dynamic behaviour accordingly. This resulting implementation is characterized by vast quantities of agent code which is hard to maintain and to adjust. Nevertheless, the increased code size of such an agent contradicts the scalability and bandwidth issues of large-scale networks. Since the memory footprint of a Jess-agent in OKEANOS is rather small, it is suitable for designing autonomous agents for the World Wide Web.

2.6 Small Memory Footprints

A further design goal of agents in OKEANOS is to facilitate their use within large-scale distributed systems such as the Internet. Therefore agents and their processing environments, the lounges, have to be small in code size in order to minimize the network bandwidth consumption. Additionally, they require less processor time to be serialised. As far as agents are concerned, Jess-agents fulfill the requirement of minimal code size because most of the application logic is specified as declarative rules. For example, the Jess-based agents described in Section 3.3 contain the entire application logic for a distributed symbolic computation and are as small as 25 KByte. It is adequate to transmit only the rules and facts, since an instance of the Jess engine is provided by each lounge. An agent's serialized code size is even further reduced if the participating lounges use sophisticated compression techniques for transferring agents [2]. As far as a lounge is concerned, the size depends on the number of services which are made globally available. The size of the serialized lounge is between 400 KByte and 600 KByte.

[2] For example, customized socket factories of the Remote Method Invocation facility of Java enables such a data compression by applying several compression formats, such as GZIP or ZIP.

3 SAT: An Application from Symbolic Computation

We consider as an application from the realm of Symbolic Computation the well-known satisfiability problem for Boolean formulae (SAT). Important applications of the SAT algorithm include cryptography [16], planning and scheduling [12], model checking for hardware verification [1], checking formal assembly conditions for motor-cars [15], and finite mathematics (e.g. quasigroup problems [19]). The computational complexity of these problems can vary considerably, where the hardest practical problems can require thousands of hours of running time. So parallelization is an important issue.

The SAT problem asks whether or not a Boolean formula has a model, or, alternatively, whether or not a set C of Boolean constraints has a solution. Usually the constraints are kept in conjunctive normal form (CNF). Each constraint is then also called a clause and consists of a set of literals, where a literal is a variable or its negation. A clause containing exactly one literal is called a unit clause. A solution assigns to each variable a value (either TRUE or FALSE), such that in each clause at least one literal becomes true.

3.1 The Davis-Putnam Algorithm

Basically, by trying all possible variable assignments one after the other, one finally finds a solution to a given SAT-instance, provided that such a solution exists. The *Davis-Putnam* (DP) algorithm [3] performs an optimized search by extending partial variable assignments, and by simplifying the resulting subproblems by applying two operations known as *unit propagation* (consisting of *unit subsumption* and *unit resolution*) and *pure literal deletion* (see Figure 2).

```
boolean dp(ClauseSet C)
{
   while ( C contains a unit clause {L} ) {
      delete clauses containing L from C;   // unit-subsumption
      delete ¬L from all clauses in C;      // unit-resolution
   }
   if ( ∅ ∈ S ) return FALSE;               // empty clause?
   pureLiteralDeletion();
   if ( C = ∅ ) return TRUE;                // no clauses?
   choose a literal L occurring in C;       // case-splitting
   if ( dp(C ∪ {L}) ) return TRUE;          // first branch
   else if ( dp(C ∪ {¬L}) ) return TRUE;    // second branch
   else return FALSE;
}
```

Fig. 2. The Davis-Putnam Algorithm

In the following, we associate with each run of the DP algorithm a search tree, which is a finite binary tree generated by the recursive calls of the case splitting step.

The nodes of the tree represent executions of the DP algorithm with a fixed input clause set C. We will label the outgoing edges of each node with the literal L resp. $\overline{L}$ which is added to C to generate the new subproblem.

3.2 Parallel SAT checking using Agents

This section deals with the basic concepts of parallel satisfiability checking with the Davis-Putnam algorithm using an agent approach. Section 3.3 provides detailed information about the realization of a parallel distributed SAT prover within the OKEANOS agent framework.

Overview of the Parallel Execution Process. For the parallel execution of the Davis-Putnam algorithm the search space has to be divided into mutually disjoint portions to be treated in parallel. However, static generation of balanced subproblems is not feasible, since it is impossible to predict the extent of the problem reduction delivered by the unit propagation step in advance. Consequently, when parallelizing the Davis-Putnam algorithm we have do deal with considerably different and completely unpredictable run-times of the subproblems.

We adopt a search space splitting technique presented in [19] which is based on the notion of a *guiding path*. A guiding path describes the current state of the search process. More precisely, a guiding path is a path in the search tree from the root to the current node, with additional labels attached to the edges. Each level of the tree where a case splitting literal is added to clause set C, i.e. each (recursive) call to the DP procedure, corresponds to an entry in the guiding path, and each entry consists in turn of the following information:

1. The literal L which was selected at the corresponding level.
2. A flag indicating whether or not backtracking has already been done for that level; we use B to indicate backtracking and N to state that no backtracking is required.

Each entry in the guiding path with flag B set is a potential candidate for a search space division. The whole subtree rooted at the node corresponding to this entry may be examined by another independent agent.

The guiding-path approach allows dynamic problem decomposition, as at any point of time during the search any agent may decide to further split its portion of the search space. Moreover, the selected literals coincide with the selections of the sequential version. Thus, approved literal selection strategies may be carried over to the parallel agent-based version of the DP algorithm.

Implementation of Search Space Splitting. To allow search space splitting, we have modified the DP algorithm to accept a guiding path object as an additional input parameter. A call to the extended DP algorithm with a non-empty guiding path makes the case-splitting literals to be chosen as indicated by the path element of the corresponding level instead of by querying the literal chooser. Additionally, in all levels backtracking, i.e. the second recursive call to DP, is only performed when the corresponding flag is set B.

When a search-space split is requested, the computation is asynchronously stopped by the agent and the actual guiding path P is used to build two new paths P_1 and P_2. Then a new agent is started with guiding path P_1, and the interrupted agent continues work with a modified guiding path P_2.

The computation starts with one agent to which the whole search space is assigned. The agents are notified about changes concerning the availability of processing capacity. Every time a free processor is found, a split is performed. This happens, for example, when an agent has completed the search in the assigned subtree without finding a model. As mentioned above, due to the nature of the SAT problem the size of a subtree cannot be predicted in advance. In turn this leads to an unpredictable splitting behavior. Dealing with this dynamic evolution of parallelism is the major challenge in parallelizing SAT provers.

3.3 Realization of the SAT prover in OKEANOS

The implementation of the parallel SAT prover in OKEANOS is basically made up of four types of agents, which are discussed in greater detail in the following paragraphs.

DP Service Agent Agents of class *DPServiceAgent* are stationary agents that actually perform the search in a subtree by executing the DP algorithm. The core Davis-Putnam algorithm is a legacy application implemented in C++. It is integrated as a native library module available in every lounge of the system. *DPServiceAgents* can use the methods of the native code via the Java Native Interface. The C++ implementation is wrapped by the *DPServiceAgents* and thus can act as a service provider for the *CalcService* to all OKEANOS agents. The *CalcService* is only registered as long as local processing capacity is available.

DP Master Agent. Each search process is initiated by a stationary agent of class *DP-MasterAgent*, which first creates an initial agent of type *DistributorAgent*, and then sets it up by sending several KQML performatives. These messages contain the input clauses and the guiding path, which in this case is empty. Thus, the whole search space is assigned to the first agent.

During its lifetime, the *DPMasterAgent* waits for replies from *DistributorAgents* containing their partial results. As soon as a model is found, the whole search is terminated. Until then, the master agent periodically checks for *CalcServices* that are currently occupied and thus indicate ongoing search. If no such *CalcService* exists, all work is done and consequently the whole search space has been traversed without finding a model. In this case the set of input clauses is not satisfiable and the computation is completed. As *DistributorAgents* keep track of which part of the search tree they have handled or given to other agents, the *DPMasterAgent* can finally detect missing agents (e.g. due to a crashed lounge) and restart a partial search when necessary in order to complete the whole search.

Distributor Agent Agents of class *DistributorAgent* are mobile agents that are responsible for solving a given SAT problem. They are moving around, looking for a service

that is capable of solving such a kind of problem (*CalcService* in our case). They compete with other agents for these special resources. The task of finding a suitable service is supported by an expert system built into every *DistributorAgent*. As soon as a *DistributorAgent* has gained access to a *CalcService*, it places a request for calculation, and waits until this request is granted. Then the input clauses and guiding path are passed to the *CalcService*. The *CalcService* reports the result of the computation as soon as it is available, which is then taken back to the *DPMasterAgent* by the *DistributorAgent*. Access to the *CalcService* is always managed and synchronized via a *DPServiceAgent*, which acts as a service provider within OKEANOS for this service.

Strategy Service Agent At each lounge where a *DPServiceAgent* resides, there is another stationary agent called *StrategyServiceAgent* which offers a *StrategyService*. *DistributorAgents* may consult this service about whether or not they should split their search space. The *StrategyService* can be accomodated to the specific environment of the lounge, such as the processor speed or kind of network connection. In case a decision is made to perform a split, the following steps are taken:

1. The *DistributorAgent* requests the *DPServiceAgent* to asynchronously stop the calculation and return the current guiding path as result.
2. The *DistributorAgent* splits the guiding path and passes one of these guiding paths together with the clause list to a newly created *DistributorAgent*.
3. The new *DistributorAgent* is sent off to a lounge with an unoccupied *CalcService*.
4. Finally, the interrupted *CalcService* is restarted with the second guiding path generated in step 2.

3.4 Benefits of the Agents Approach

The key benefit of using the described approach is the service-oriented character of OKEANOS agents. Each computation-related task such as the lookup of the calculation service, the determination of split times and the right strategy for splitting is encapsulated within different kinds of services. Thus there are strategy and calculation services provided by each lounge. Since each of these services is only attached to one lounge, they can be tailored to the specific lounge's environment, such as the type and location of the host machine. For example each lounge can have its own dedicated strategy service for carrying out the best splits at this lounge. However, agents are not bound to use the provided services at all. They also can work out splits completely autonomously without involving such a local service provider.

In contrast to a central master process in a classical master slave parallelization technique, our proposed agent approach is more flexible because by employing autonomous agents reaction to load changes is possible in a decentralized manner. Basically, parallelism in the agent approach is achieved through the process of migration of agents within a pool of compute-service providers.

3.5 Empirical Results

For the realization of performance measurements we have selected three benchmarks (dubois20, dubois23 and dubois26) from the publicly available [3] DIMACS benchmark suite for SAT provers. The chosen benchmarks have different sequential running times and all exhibit a highly irregular search splitting behaviour.

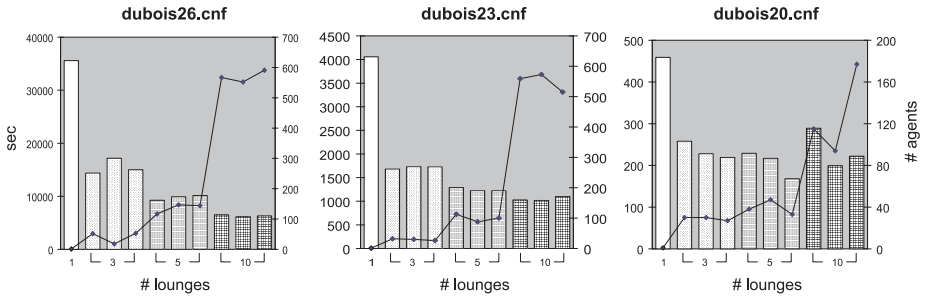

Fig. 3. Results of the Performance Measurements

The measurements where carried out on a heterogeneous computing pool consisting of machines of the following types: Two Sun Ultra E450 with 4 processors each at 400MHz running under Solaris 7, and up to four PentiumII PCs at 400MHz running under Windows NT 4.0. For all measurements the wall clock time in seconds was taken.

Figure 3 shows the results for 3, 5 and 10 lounges and relates them to the sequential running times. For each number of lounges the times of three program runs are shown. Figure 3 shows also the total number of agents that were involved during the computation. Each lounge provided a simple strategy service which suggests to initiate a split and sends a distributor agent as soon as a calculation service within OKEANOS is made available. This strategy did not take into account the overall number of lounges and number of calculation services. Each performance measurement for a given number of participating lounges ends up with a slightly different runtime because of the unpredictable behaviour of the agents and the non-deterministic nature of the SAT problem.

Generally, the overall speedup is well suited for parallelizing large search spaces (dubois26.cnf, seq. running time about 10 hours). If the sequential running time is rather short (dubois20.cnf, seq. running time about 8 minutes), the performance gain of our agent approach is less evident. When adding more lounges to the distributed computation infrastructure, the number of generated agents increases accordingly.

Our future work will focus on more sophisticated splitting strategies in order to adapt the number of generated agents more appropriately to the current number of globally available computing services. The main goal is to find adequate, cooperative strategies which match the dynamic and unpredictable character of SAT problems.

[3] ftp://dimacs.rutgers.edu/pub/challenge/sat/benchmarks/cnf/

4 Related Work

JavaParty [17] provides transparent remote Java objects and remote threads using precompiling techniques. *Java// (ProActive)* [2] is a 100% Java library for seamless cross-paradigm high performance computing. Both environments are targeted towards high performance computing in pure Java. In contrast, our approach uses Java for the parallelization infrastructure, while the actual algorithm is provided as a native code module and is accessible as a service.

MATS [8] is a mobile agent system for distributed processing. It is based on collections of agents which form teams to solve distributed tasks. The user is may be forced to structure the problem in such a way that it can be easily broken into a set of cooperating tasks, whereas in OKEANOS it is the responsibility of the autonomous agent to figure out an appropriate plan to distribute the tasks. The granularity of distribution in OKEANOS is directly related to each agent's strategy which is determined by applying techniques from the area of artificial intelligence.

PSATO [19] is a distributed/parallel prover for propositional satisfiability for a network of workstations. In contrast to our work, a master slave model is applied, where a central master is responsible for the division of the search space and for assigning the subtasks to the slaves.

5 Conclusion

In this paper, we have described a distributed infrastructure that provides mobility and application-level services to software agents. Its design as a framework enables the conceptual and operational reuse of services in a generic manner to support the scalability issues for agents in large computer networks. The agent framework supports asynchronous communication and uses message passing via an open and standardized message format. By integrating techniques from the area of artificial intelligence, the complexity of the design and implementation of autonomous mobile agents for distributing tasks is reduced tremendously. We have implemented the service agent framework in Java to illustrate its feasibility. A distributed symbolic computation is then used to show the benefits.

It turns out that a service framework for mobile agents is very suitable for distributed, symbolic computing environments as it allows to benefit from its framework services in a general acessible manner and provides and extensible computing platform over wide area networks.

References

1. A. Biere, A. Cimatti, E. Clarke, and Y. Zhu. Symbolic Model Checking without BDDs. In *Tools and Algorithms for the Analysis and Construction of Systems (TACAS'99)*, number 1579 in LNCS. Springer-Verlag, 1999.
2. D. Caromel, W. Klauser, and J. Vayssiere. Towards Seamless Computing and Metacomputing in Java. *Concurrency: Practice and Experience*, 10(11–13):1043–1061, 1998.

3. M. Davis and H. Putnam. A Computing Procedure for Quantification Theory. In *Journal of the ACM*, volume 7, pages 201–215, 1960.
4. M. Fayad and D. Schmidt. Object-Oriented Application Frameworks. *Communications of the ACM*, 40(10), October 1997.
5. T. Finn, Y. Labrou, and J. Mayfield. KQML as an Agent Communication Language. In J.M. Bradshaw, editor, *Software Agents*, pages 291–316. MIT Press, 1997.
6. E.J. Friedman-Hill. Jess, The Java Expert System Shell. Available at the URL: http://herzberg.ca.sandia.gov/jess/, 1999.
7. M.R. Genesereth. An Agent-Based Framework for Interoperability. In J.M. Bradshaw, editor, *Software Agents*, pages 317–345. MIT Press, 1997.
8. M. Ghanea-Hercock, J.C. Collis, and D.T. Ndumu. Heterogenous Mobile Agents for Distributed Processing. In *Proceedings of the Third International Conference on Autonomous Agents (Agents '99)*, May 1999. (Workshop on Agent-based Highperformance Computing, Seattle, USA).
9. J. Gosling and K. Arnold. *The Java Programming Language*. Addison-Wesley, Reading, Massachusetts, 1996.
10. R. Johnson and B. Foote. Designing Reusable Classes. *Object-Oriented Programming*, 1(2):22–35, 1988.
11. R. Johnson and V. Russo. Reusing Object-Oriented Design. Technical Report 91-1996, University of Illinois, 1991.
12. H. Kautz and B. Selman. Pushing the Envelope: Planning, Propositional Logic, and Stochastic Search. In *Proceedings of the Thirteenth National Conference on Artificial Intelligence (AAAI-96)*, 1996.
13. E.A. Kendall and M.T. Malkoun. The Layered Agent Patterns. Available at the URL: http://www.cse.rmit.edu.au/˜rdsek/, 1997.
14. G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, J.-M. Loingtier C. Lopes, and J. Irwin. Aspect-Oriented Programming. Technical Report SPL97-008 P9710042, XEROX Palo Alto Res. Center, February 1997.
15. W. Küchlin and C. Sinz. Proving Consistency Assertions for Automotive Product Data Management. In I. P. Gent and T. Walsh, editors, *Journal of Automated Reasoning*, volume 24, pages 145–163. Kluwer Academic Publishers, Feb. 2000.
16. F. Massacci and L. Marraro. Logical Cryptoanalysis as a SAT Problem. In I. P. Gent and T. Walsh, editors, *Journal of Automated Reasoning*, volume 24, pages 165–203. Kluwer Academic Publishers, Feb. 2000.
17. M. Philippsen and M. Zenger. JavaParty – Transparent Remote Objects in Java. *Concurrency: Practice and Experience*, 9(11):1225–1242, 1997.
18. R. Schimkat, S. Müller, W. Küchlin, and R. Krautter. A Lightweight, Message-Oriented Application Server for the WWW. In *ACM 2000 Symposium on Applied Computing*, Como, Italy, March 2000. Association for Computing Machinery.
19. H. Zhang, M. P. Bonacina, and J. Hsiang. PSATO: A Distributed Propositional Prover and its Application to Quasigroup Problems. In *Journal of Symbolic Computation*, volume 21, pages 543–560. Academic Press, 1996.