Developing Java Grid Applications with Ibis

Kees van Reeuwijk, Rob van Nieuwpoort, and Henri Bal

Vrije Universiteit Amsterdam
{reeuwijk,rob,bal}@cs.vu.nl

Abstract. *Ibis*[1] is a programming environment for the development of grid applications in Java. We aim to support a wide range of applications and parallel platforms, so our example programs should also go beyond small benchmarks. In this paper we describe a number of larger applications we have developed to evaluate Ibis' suitability for writing grid applications: a cellular automata simulator, a solver for the Satisfiability problem, and grammar-based text analysis. We give an overview of the applications, we describe their implementation, and we show performance results on a number of parallel platforms, ranging from a large supercomputer cluster to a real global grid testbed.
Since all of these applications require communication between the processors during execution, it is not surprising that a supercomputer cluster proved to be the most effective platform. However, all of our applications were also efficient on a wide-area cluster system, and some of them even on a grid testbed. Since grid systems are usually only used for trivially parallel systems, we consider these results an encouraging sign that Ibis is indeed an effective environment for grid computing. In particular because for two of the three of the applications the parallelisation required very little additional program code.

1 Introduction

Traditional supercomputing offers large amounts of computational power, but requires tightly controlled homogeneous systems at a single location. For grid computing these restrictions are lifted, and it is assumed that effective computation is still possible on heterogeneous, widely distributed, and independently managed computer systems. The additional flexibility of such a configuration is attractive, but writing efficient software for grid computing is challenging: Differences in processor architecture and power, external loads on the processors, differences in network performance, geographical distance, security measures such as firewalls and proxies, and the possibility of faults in processors and networks, all complicate software development.

For grid computing it is also desirable that processors can join a running computation. This is called *open-world* computation, in contrast to traditional *closed-world* computation. However, the open-world requirement complicates the program, and restricts the way a program can be parallelised.

The complications of grid computing require a solid programming environment to hide these complexities. *Ibis* provides such an environment. It not only allows programs to be written in Java, but is also itself written in Java. Choosing Java already solves many

[1] Ibis is available under an open source licence, and can be downloaded from www.cs.vu.nl/ibis.

J.C. Cunha and P.D. Medeiros (Eds.): Euro-Par 2005, LNCS 3648, pp. 411–420, 2005.
© Springer-Verlag Berlin Heidelberg 2005

software portability issues (very important in such a heterogeneous environment!), and allows the programmer to use a modern high-level language.

Standard Java has some support for distributed computing through the Remote Method Invocation (RMI) library. Ibis provides a very efficient [1] implementation, but RMI only allows client-server style parallel programming, which is not suitable for many parallel problems. Ibis therefore offers a wide range of communication models, including replicated objects, group/collective communication, and a divide-and-conquer programming model. Also, the Ibis implementation layer, primarily designed to support the higher-level models, has proved to be an effective programming model for some problems.

To provide this functionality, other systems typically compromise portability, for example by interfacing to a native MPI library. In Ibis, all parallel programming models are cleanly integrated into Java. However, behind this friendly facade a lot of work is done for the sake of efficiency: native implementations are dynamically selected for high-performance networks such as Myrinet [2], bytecode is rewritten to generate efficient communication and parallel code [3], and when possible the improved functionality of modern JVM implementations is exploited. However, portable implementations are always available as fallback.

We aim to support a wide range of programs, so it is important to go beyond small benchmarks, and try larger applications. In this paper we describe a number of larger applications we have developed to evaluate Ibis' suitability for writing grid applications: a Cellular Automata simulator (§4), a solver for the Satisfiability problem (§5), and grammar-based text analysis (§6). §2 describes our measurement setup and the way we evaluate the measurements. §3 describes the Satin divide-and-conquer framework. In §7 and §8 we show some results for wide-area systems.

2 Measurement Setup and Evaluation

As part of our application descriptions, we will show performance results on a single site of the DAS2 supercomputer cluster system [4]. Each node of this cluster is a dual 1GHz Intel Pentium III system. Unless specified otherwise, we use the IBM 1.4.1 JVM.

On homogeneous systems like the DAS2 cluster, the efficiency of a computation is easily determined. Ideally, a cluster of N processors has a *speedup* of N: it is N times faster than an individual processor. However, for computations with non-identical processors the notion of speedup is meaningless. Instead, we express the efficiency as a fraction of the ideal speed of the system. Given a system with N nodes, and execution times on individual nodes $t_1 \dots t_N$, each processor ideally contributes to a cluster computation inversely proportional to its individual computation time. Thus, the ideal execution time is $t_{ideal} = 1/\sum_{i=1}^{N} 1/t_i$. Given a real cluster computation time t_p, the efficiency of the cluster computation is $\eta = t_{ideal}/t_p$. For a homogeneous system, the execution time t on each processor is the same, so $t_{ideal} = t/N$.

3 Satin

Satin [5, 6] is a divide-and-conquer framework similar to Cilk [7], but built on top of Ibis. In Satin, the user must annotate methods that can be executed in parallel, and

provide an explicit demarcation point where the results of these methods should be available. For example, the following method recursively creates tasks, waits for them to complete (the `sync()` call), and uses the results to compute its own result:

```
// List all parallel methods in a subinterface of Spawnable
interface I extends ibis.satin.Spawnable {
    int f(int n);
}

class F extends ibis.satin.SatinObject implements I {
    int f(int n) {
        if(n<2) return n;
        int x = f(n-1); // these two methods
        int y = f(n-2); // are executed in parallel
        sync(); // Satin method: wait for the f() methods
                // to finish before using their results.
        return x+y;
    }
}
```

This implicitly parallel code is rewritten by Ibis to explicitly parallel code, but that is done on the bytecode, and is invisible to the programmer. This parallel code implements a cluster-aware work-stealing algorithm that usually is very efficient, even on a grid system.

4 A Cellular Automata Simulator

Many simulations can be described as interactions between cells, with each cell in one of a finite number of *states*. Typically the cells are arranged in a 2- or 3-dimensional rectangular matrix. The state of the system progresses in a sequence of discrete steps, where the next state of each cell is determined by the state of the cell itself and that of its immediate neighbours, according to a homogeneous, fixed set of rules. Such a problem is called a *cellular automata* (CA) problem. Such problems occur for example in biology [8] and urban planning [9].

In our example program we implement a simple ecological model where each cell represents a patch of land that can be seeded from one of the eight neighbouring patches with grass or trees, and where woodlands regularly suffer from forest fires. The structure and behaviour of the program is largely independent of the exact update rules, so our findings are applicable to a larger set of problems.

We parallelise the computation by distributing the matrix over multiple processors. Since a cell update requires the state of the neighbours, cell states must be communicated between processors. Since the computation of the new generation on a processor can only be started when its neighbours have completed the previous generation, there is a fairly close synchronisation between the processors.

In our implementation, we organise the processors in a one-dimensional array, and distribute the cells column-wise over the processors. This is simple to implement, but not optimal: a two-dimensional distribution of the cells may require less communication. However, since the gains are often small or non-existent, support is complex, and

	2000^2 50000 it.	**5000^2 10000 it.**	**10000^2 2000 it.**
1	7833.27	9766.39	7825.75
2	4061.70	5051.58	4040.28
4	2120.79	2560.48	2031.04
8	1148.90	1295.72	1023.70
16	607.50	665.54	522.02
32	374.37	360.61	268.04
48	273.67	251.73	188.58
64	247.25	211.94	154.74
72	238.60	198.80	142.71

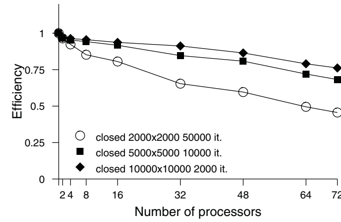

Fig. 1. Parallel execution times in seconds and efficiency of our closed-world CA simulator.

since we want to allow open-world computation, we accept the potential inefficiencies of column-wise distribution.

Our *closed-world* implementation requires a fixed number of processors that is known at the start of the program. All processors have an equal share of the matrix, and initialise their part of the matrix.

We also implemented an *open-world* version. Since the number of participating processors is not known in advance, the first processor to join the computation must initialise the entire grid; when other processors join the computation, they are sent a fair share of the matrix. This requires a significant amount of communication.

The open-world version also dynamically redistributes the computational load. When a processor has completed a generation, it sends work requests to its neighbours. A neighbour that has not completed its own computation sends some columns of the matrix to its faster neighbour. To dampen temporary disturbances, sporadic requests are only honoured by small redistributions, and repeated requests by larger redistributions.

The program was implemented using the Ibis communication layer. This layer was mainly designed to support the higher levels of Ibis, but it was also a good choice for the CA simulator. We estimate that a sequential simulator would require 200 lines of code. The closed-world version is about 400 lines of code, and the open-world version is about 1200 lines of code, mainly because of the load-balancing mechanism.

We do our simulations on a square matrix, denoted as a squaring expression, e.g. 20^2. In Fig. 1 we show execution times of the closed-world simulator for various problem sizes. As the results indicate, the program can achieve good speedups, especially for large grid sizes.

In Fig. 2 we show the results of open-world simulation on a 5000^2 matrix. For comparison we repeat the results of the closed-world simulation. As expected, the fact that one processor starts with the entire matrix, and then sends most of it to other processors, has a significant impact, particularly for larger numbers of processors. To evaluate this effect we also show the results for the last 75% of the iterations. The results indicate that after the initial redistribution, open-world simulation is as efficient as closed-world simulation.

Our measurements indicate that the dynamic redistribution system is very effective in systems with moderate imbalances. However, in extreme cases a slow processor or a high-latency communication link can determine the pace of the computation or can

	closed	open	started
1	9766.39	10628.54	7969.99
2	5051.58	5326.20	3988.71
4	2560.48	2785.53	2085.54
8	1295.72	1386.68	1028.97
16	665.54	774.53	560.16
32	360.61	460.86	313.36
48	251.73	376.60	222.29
64	211.94	344.05	178.47
72	198.80	324.44	136.17

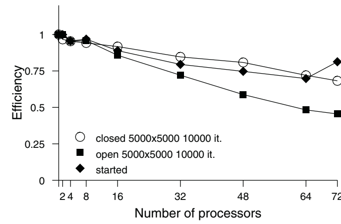

Fig. 2. Parallel execution times in seconds and efficiency for 10000 iterations of our open- and closed-world CA simulators on a 5000^2 matrix. Also, the parallel execution times and efficiency for the last 7500 iterations of our open world simulator.

render the balancing mechanism ineffective. In such cases it would be more effective to withdraw a processor from the computation, but this is currently not supported.

5 Solving the Satisfiability Problem

Given a symbolic Boolean expression, the *Satisfiability* (*SAT*) problem requires that a set of assignments to the variables of the expression is found for which the expression evaluates to true, or that it is established that no such set of assignments exists[2].

The SAT problem plays a pivotal role in theoretical computer science as a representative example of an NP-complete problem. It also occurs in a number of practical applications. Since the problem is NP-complete, no algorithm is known that is guaranteed to solve this problem in polynomial time. Nevertheless, a number of heuristics allow the development of practical SAT solvers.

A brute-force SAT solver could try all possible combinations of assignments, and see if one of them yields true. However, since an expression with n variables has 2^n combinations, this is rarely practical. A better strategy is to try to eliminate large parts of the solution space at once by evaluating partial assignments. For example, the expression $(a \vee \neg b) \wedge (b \vee c \vee \neg d)$ can never be satisfied with the assignments $a = \texttt{false}, b = \texttt{true}$, since the first clause cannot be satisfied for any assignment to c and d. Yet the assignments $a = \texttt{true}, c = \texttt{true}$ satisfy the entire Boolean expression for all assignments to b and d. Modern SAT solvers use a backtracking search that speculatively assigns values to variables until the problem is either satisfied, or until there is a conflict. Upon a conflict, the solver backtracks. The efficiency of the search process is strongly influenced by the order in which the variables are assigned [10]. A common and effective heuristic is to select variables that satisfy as many unsatisfied clauses as possible. More refined variable selection heuristics tend to require more sophisticated bookkeeping, and are often not deemed to be worth the extra trouble.

The backtracking search maps naturally to a divide-and-conquer implementation: our SAT solver, using a recursive backtracking search as described above, is implemented in 2500 lines of code. Only 25 of these are required by the Satin framework.

[2] For a more detailed overview of the Satisfiability problem and its solvers see for example [10].

	uuf200	FPGA12	FPGA13
1	205.01	4322.26	12635.36
2	108.18	2767.59	8349.74
4	47.42	1335.31	3903.39
8	25.00	631.83	1902.20
16	14.58	313.91	901.26
32	10.45	147.15	437.58
48	9.05	99.00	282.81
64	8.68	76.55	210.07
72	8.07	70.54	186.48

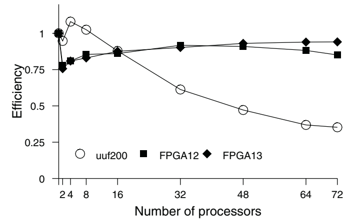

Fig. 3. Parallel execution times in seconds and efficiency for the SAT solver.

Figure 3 shows results for our solver for a number of SAT problems. The *uuf200* problem is from the SATLIB collection [11]. It is a set of 860 random clauses with 3 variables each, and 200 variables in total. Random problems are known to be difficult for many SAT solvers, since there is no structure to guide them, and problems with the chosen proportion of clauses and variables are known to be the most difficult [12]. The *FPGA12* and *FPGA13* problems are described in [13]. They represent routing problems through FPGA switchboxes. FPGA12 has 240 variables and 1344 clauses; FPGA13 has 260 and 1586. All problems are known to be unsatisfiable.

As results of Fig. 3 indicates, for the large problems parallel efficiency is very good, even for large numbers of processors. The uuf200 problem is too small to scale well.

6 Grammar-Based Text Analysis

As a final application we show *Grammy*, a program that analyses text by constructing a grammar that produces the original sentence. For example, for the sentence ‘a long long time ago’ we could construct the grammar

$$\begin{array}{ll} \textbf{start} & \rightarrow \text{`a}\diamond\diamond\text{ time ago'} \\ \diamond & \rightarrow \text{` long'} \end{array}$$

Constructing a compact grammar is useful for text analysis, since it infers hierarchical structure [14]. The analysis is also useful for compression. In fact, the classical compression algorithm LZ78 [15], can be viewed as constructing a grammar.

Choosing the most effective grammar rules is often difficult, since a text usually has many repeated sequences to choose from. The most obvious strategy is to repeatedly select the longest repeat, or the repeat resulting in the largest gain. However, that is not optimal, since each choice may preclude subsequent choices. We approximate optimal compression by considering a number of efficient choices at each step and looking ahead a number of steps. Since each of the possibilities can be evaluated independently, this can be implemented as a recursive parallel process. Our implementation was parallelised very effectively by using Satin: only about 20 of the 1850 lines of code of the program are required by the Satin framework.

To evaluate our program, we use the following texts: William Shakespeare, *A Midsummer Night’s Dream* (96508 Bytes, *text 1*), Sir Arthur Conan Doyle, *The Adventure*

	text 1	text 2	text 3	text 4
1	173737.98	61247.16	6812.21	6111.66
2		32684.88	3704.80	3334.36
4	46890.91	16287.13	1894.03	1672.51
8	23538.16	8290.08	1004.83	905.42
16	12023.00	4398.25	593.16	537.00
32	6647.05	2539.03	394.94	360.50
48	4862.56	1954.54	335.84	306.44
64	3992.91	1647.83	308.76	282.70
72	3694.21	1547.16	296.34	275.70

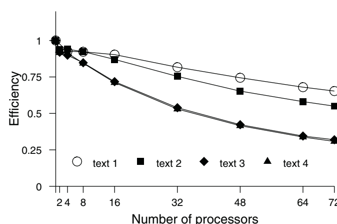

Fig. 4. Parallel execution times in seconds and efficiency for grammar construction.

of the Red Circle (53394 Bytes, *text 2*), the *grep* man page of a recent Debian system (20047 Bytes, *text 3*), and the file `SuffixArray.java` from the Grammy source code (28572 Bytes, *text 4*).

Figure 4 shows the execution times and efficiency for these texts. All runs were done with a lookahead of 5 steps and with at most 7 candidates at each step. These results show that for large texts the computation scales quite well to larger numbers of processors. This is because large texts tend to have many repeats, resulting in sufficient parallelism to keep all processors busy.

7 Results on Wide-Area Clusters

In §4, §5, and §6 we showed results for clusters of processors on a single DAS2 site. Figure 5 shows results for the same programs on clusters of processors on two and four DAS sites in the Netherlands. In all cases we use an equal number of processors on the participating sites, so if we run a program on 64 processors on four sites, each site has 16 processors. As for computation on a single site, we compute the efficiency of the parallel computation relative to computation on a single node.

Since the CA computation requires information exchange after each iteration, the processors run in tight lockstep. The larger latency of the wide-area links therefore has a noticeable influence on the efficiency of the computation. Nevertheless, the computation is efficient enough to be useful.

The SAT solver and the text analysis also require communication for work stealing, but the Satin framework was able to hide the higher latencies of the wide-area links. In fact, in a few cases wide-area execution was more efficient than execution on a single site, presumably due to reduced contention on shared resources such as local communication channels.

8 Results on a Global Grid

Finally, we have executed a number of runs on a grid testbed. Efficient computation on such a system requires a careful choice of communication structure. Often communication between grid nodes is avoided entirely, but this obviously restricts the use of grid systems to trivially parallel systems, and all of our example programs require more.

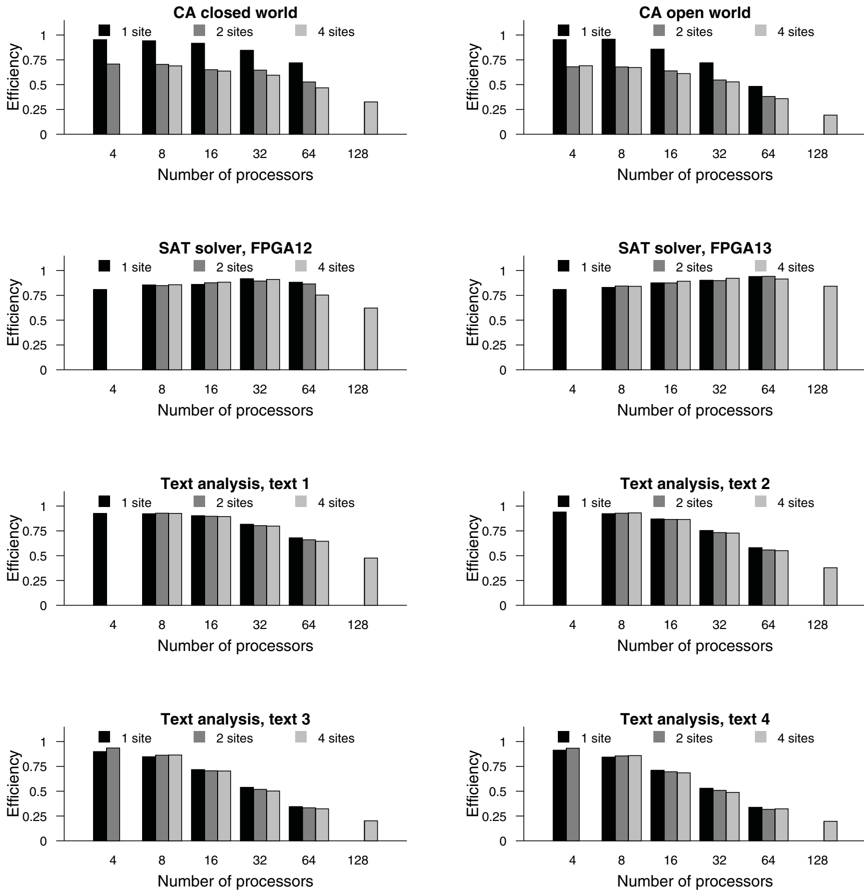

Fig. 5. Efficiency of computations on two and four DAS sites, compared to a single site.

Our closed-world CA simulator is clearly unsuitable, since it divides the computation equally over the processors. The other programs were run using the following set of systems and JVM implementations (site 1 is a DAS2 site):

site	CPUs	Architecture	JVM	Location
1	16	Intel Pentium III 1GHz	IBM 1.4.1 32 bit	the Netherlands
2	4	Intel Xeon 3GHz	SUN 1.4.2 32 bit server VM	Czech Republic
3	2	Intel IA64 (Itanium) 1.4 GHz	SUN 1.4.2 64 bit server VM	Poland
4	4	Intel Xeon 2GHz	SUN 1.4.2 32 bit server VM	Louisiana, USA
5	2	Intel Xeon CPU 2.4 GHz	SUN 1.4.2 32 bit server VM	Germany
	28	**total**		

Table 1 shows the results for these runs. The SAT solver performs very well on a world-wide grid. The other two applications performed almost as well as on the wide-area DAS-2 system after we removed two sites. We removed site 3 because the (64 bit) JVM on that site performs very poorly on the CA and text analysis applications, and did not

Table 1. Execution times in seconds on individual grid sites, on the combined set of processors (t_p), and efficiency of the various programs on a global grid.

Benchmark	system 1	2	3	4	5	t_{ideal}	t_p	η
SAT solver, FPGA12	318.88	615.41	1263.70	727.50	1294.16	129.88	146.85	0.88
SAT solver, FPGA12	318.88	615.41	-	-	1294.16	180.71	217.99	0.83
Text analysis, text 4	532.68	1052.97	-	-	2350.49	307.46	459.82	0.67
CA open 5000x5000, 2500 it	284.40	652.44	-	-	1652.83	176.87	266.39	0.66

contribute anything. Site 4 is located in the USA and has a high latency link to Europe. This interferes with the load balancing algorithm of the CA application: due to the high latency, steal requests arrive after the victim has finished the iteration. The text analysis application does not perform well if site 4 is used due to the limited amount of parallelism that is generated. Jobs are relatively small, and the transfer of jobs over the slow WAN link does not outweigh the cost. For comparison we also show the results for the SAT solver without these sites.

9 Conclusions and Future Work

In this paper we have shown the use of Ibis for a number of larger applications. Ibis proved to be very effective. Both the SAT solver and the text analyser could be developed as mainly sequential programs, with only a few additional lines of code to interface to the Satin framework. Parallelisation doesn't get much simpler than this. Although the Cellular automata simulator required more explicitly parallel code, the amount of parallel code was still limited, even for the load-balancing mechanism in the open-world version.

All programs performed well on a traditional supercomputer cluster, and a wide-area cluster system. Since all of the programs require communication between the processors, execution on a grid system was not always efficient, but even there very creditable results could be achieved, in particular for the SAT solver.

We are currently extending Ibis with support for fault tolerance, more elaborate automatic configuration, and peer-to-peer computing. Other areas of study are performance debugging and additional high-level parallel programming models.

Acknowledgements

This work was partially supported by the Dutch Organisation for Scientific research (NWO). This work was part of the Virtual Laboratory for e-Science project (www.vl-e.nl). This project is supported by a BSIK grant from the Dutch Ministry of Education, Culture and Science (OC&W) and is part of the ICT innovation program of the Ministry of Economic Affairs (EZ).

References

1. Nieuwpoort, R.V.v., Maassen, J., Wrzesinska, G., Hofman, R., Jacobs, C., Kielmann, T., Bal, H.E.: Ibis: a flexible and efficient Java-based grid programming environment. Concurrency and Computation: Practice and Experience **16** (2003) 1–29 Published online in Wiley InterScience (www.interscience.wiley.com). DOI 10.1002/cpe.860.
2. Aumage, O., Hofman, R., Bal, H.: Netibis: An efficient and dynamic communication system for heterogeneous grids. In: Proc. of CCGrid. (2005) (accepted for publication)
3. Nieuwpoort, R.V.v., Maassen, J., Hofman, R., Kielmann, T., Bal, H.E.: Ibis: an efficient Java-based grid programming environment. In: Joint ACM Java Grande - ISCOPE 2002 Conference, Seattle, Washington, USA (2002) 18–27
4. Bal, H., et al.: The distributed ASCI supercomputer project. ACM SIG, Operating System Review **34** (2000) 76–96
5. Nieuwpoort, R.V.v., Maassen, J., Hofman, R., Kielmann, T., Bal, H.E.: Satin: Simple and efficient Java-based grid programming. In: AGridM 2003 Workshop on Adaptive Grid Middleware, New Orleans (2003)
6. Nieuwpoort, R.V.v., Kielmann, T., Bal, H.E.: Efficient load balancing for wide-area divide-and-conquer applications. In: Proc. Eight ACM SIGPLAN Symp. on Princ. and Practice of Par. Progr. (PPoPP), Snowbird, UT, USA (2001)
7. Blumofe, R.D., Joerg, C.F., Kuszmaul, B.C., Leiserson, C.E., Randall, K.H., Zhou, Y.: Cilk: An efficient multithreaded runtime system. J. of Par. and Distr. Computing **37** (1996) 55–69
8. Ermentrout, G.B., Edelstein-Keshet, L.: Cellular automata approaches to biological modeling. J. theor. Biol. **160** (1993) 97–133
9. Colonna, A., Stefano, V.d., Lombardo, S., Papini, L., Rabino, G.A.: Learning cellular automata: Modelling urban modelling. In: Proc. 3rd Intl Conf. on GeoComputation, University of Bristol, UK (1998) 388–395
10. Zhang, L., Malik, S.: The quest for efficient boolean satisfiability solvers. In: Proc. of the 18th Intl Conf. on Automated Deduction, Copenhagen, ACM (2002) 295–313
11. Hoos, H.H., Stützle, T.: Satlib - the satisfiability library. webpage (2000) www.satlib.org.
12. Cheeseman, P., Kanefsky, B., Taylor, W.M.: Where the Really Hard Problems Are. In: Proceedings of the Twelfth International Joint Conference on Artificial Intelligence, IJCAI-91, Sidney, Australia (1991) 331–337
13. Aloul, F., Ramani, A., Markov, I., Sakallah, K.: Solving difficult SAT instances in the presence of symmetry. In: Proc. of the Design Automation Conf. (DAC), New Orleans (2002)
14. Nevill-Manning, C.G., Witten, I.H.: Compression and explanation using hierarchical grammars. The Computer Journal **40** (1997) 103–116
15. Ziv, J., Lempel, A.: Compression of individual sequences via variable-rate coding. IEEE Transactions on Information Theory **24** (1978) 530–536