A Preliminary Review of Literature on Parallel Constraint Solving

Ian P. Gent, Chris Jefferson, Ian Miguel, Neil C.A. Moore, Peter Nightingale, Patrick Prosser, Chris Unsworth

Computing Science,
Glasgow and St. Andrews Universities, Scotland
pat@dcs.gla.ac.uk

Abstract. With the ubiquity of multicore computing, and the likely expansion of it, it seems irresponsible for constraints researchers to ignore the implications of it. Therefore, the authors have recently begun investigating the literature in constraints on exploitation of parallel systems for constraint solving. We have been compiling an incomplete, biased, and ill-written review of this literature. While accepting these faults, we nevertheless hope that it may provide some useful pointers to others wishing to follow a similar path to us: that is a path from complete to only partial ignorance.

1 Introduction

Why have multicore machines arrived, and how are we to make best use of them? We start our review by looking at the justification for the multicore architecture, the direction it is most likely to take and limiting factors on performance such as Amdahl's law. We then review recent literature on parallel constraint programming and SAT solving. We group this into four areas: parallelizing the search process; parallel and distributed arc-consistency; multi-agent and cooperative search; and combined parallel search and parallel consistency.

2 The Hardware: multicore, GPU and Amdahl's law

Written in 2006, Intel's White Paper [14] starts by saying "... two cores are here now, and quad cores are right around the corner". Now, 12 and 16 core machines are commonly available. But why go multi-core? In the past performance improvements could be taken for granted as clock speeds increased (from 5 MHz to more than 3 GHz), component size decreased, and chip density increased. Three reasons are given for the shift to multi-core. First, although component size continues to fall, power-thermal issues limit performance, so we can no longer simply increase clock speeds. Secondly, power consumption: individual cores can be tuned for different usages (i.e. dedicate hardware resources to specific functions), and when not in use cores can be powered down. And thirdly, rapid design cycles: hardware designs can be reused across generations.

What are the major challenges? Intel put top of their list "programmability", that the platform must address new and existing programming models. And then "adaptability", such that the platform can be dynamically reconfigured to conserve power. Of course "reliability", "trust", and "scalability" are also important, as we increase cores we cannot compromise the correctness of the hardware.

Intel considers development of multi-core software to be amongst the greatest challenges for tera-scale computing, specifically with regard to ensuring that "there are compelling applications and workloads that exploit the massive compute density" and that "multiprocessing adds a time dimension that is extremely difficult for software developers to cope with". They give a further justification for the multi-core architecture: "... why tomorrow's applications need so many threads. The answer is that those advanced, intelligent applications require supercomputing capabilities, and the accompanying parallelism that allows those applications to proceed in real-time. ... it requires an equally massive shift in hardware and software."

Intel's tera-scale computing vision is to aim for hundreds of cores on a chip, giving the capability of performing trillions of calculations per second on trillions of bytes of data with a stated goal (2006) "... a 10X improvement in performance per watt over the next 10 years.". How close are we to that goal? In 2006 two core machines existed, at time of writing in Summer 2011, 24 core machines are commercially available, and Intel's Teraflop Research Chip (Polaris) contains 80 cores.

But there is a shadow cast over this optimism, Amdahl's law. Amdahl's law predicts the maximum speedup that can be expected from a system as we increase the number of processors. The law assumes that a program is composed of a parallel part P and a sequential part S, such that $P + S = 1$. The expected speed up is then $1/(S + P/N)$, where N is the number of processors. As N tends to infinity Amdahl's law predicts that maximum speedup will be $1/S$, as the original P/N term tends to zero. As an example if we had $P = 0.99$, so 99% of our problem can parallelised, 64 processors would run our program 39 times faster. For 128 processors the speedup is 56 times, for 1024 processors it is 91. As the number of processors continues to increase the speedup tends to 100. If $P = 0.9$ the law predicts a maximum speed up of 10, and if half only our program can be parallelized, $P = 0.5$ and maximum speed up is 2, regardless of the number of processors available. It was this argument, in the late 1960's, that encouraged hardware development away from multi-processor and towards faster processors.

In the late 1980's Gustafson [11] argued that Amdahl's law is overly pessimistic, as it assumes that as we increase the available parallel processors we continue to keep the workload fixed and hope for reduced runtime. That is, it is a "fixed-size speedup" model and assumes N and P are independent; multiprocessing is only used to improve response time. Gustafson assumes that problem size also scales with the number of processors, i.e. as we get more processors we increase the problem size and that run time, not problem size, is a constant.

Gustafson observed that the parallel or vector parts of a program scales with problem size and the serial part does not (it diminishes proportionally). Consequently as we get more processors the workload grows and P increases resulting in an increase in speedup. This is the "fixed-time speedup" model and an example is weather forecasting, where we use multi-processors to increase the quality of our results (the weather prediction) in a fixed amount of time (before the evening news).

A third model is "memory-bounded speedup" [35]. As we increase the number of processors we increase the amount of memory available. Instead of keeping execution time fixed we increase the size of the problem to fill available memory and increase execution time, i.e. memory capacity is considered the dominant factor.

Hill and Marty [16] looked at asymmetric designs, where individual cores in a multi-core chip have different features. They argue that best performance will be had from chips that have a mix of high and low performance cores. They claim that we need a third Moore's law: the first law is doubling in performance by reducing component size and increasing clock speed; the second law results in doubling of cores per chip; the third law is a doubling in the amount of software that can be run in parallel and that software must become a producer rather than a consumer of performance gains. They also propose that we do not just consider speedup but also "costup": an increase in performance that is greater than the increase in cost, be it measured in money or energy.

Sun and Chen [36] argue that multicore architectures are fundamentally scalable and not limited by Amdahl's law and that the limits to performance will be the "memory wall", i.e. that data access will be the limiting factor. The rate of improvement in processor speed exceeds that of improvements in speed of memory and with time the performance gap will grow excessively large. In 1995 Wulf and McKee [37] predicted that performance degradations due to cache misses, with subsequent memory accesses, would exceed any improvements in processor speed. They predicted that "... in 10 to 15 years memory access will cost, on average, tens or even hundreds of processor cycles.". In 2001 typically execution speed was 1 nanosecond, and a fetch from main memory 100 nanoseconds. By 2005 this gap had grown so a fetch from main memory takes 220 CPU cycles. However, Sun and Chen suggest that multicore architectures might also reduce the effect of the memory wall. Amdahl painted a dark picture and this had a profound effect on the subsequent direction of hardware development. Sun, Chen, Ni, Gustafson, Hill, Marty and others have tried to bring some light back to the picture and "... shatter the pessimistic view of limited scalability of multicore architectures ..."

One current area of great interesting is solving on GPUs. Almost all modern desktops and laptops provide a powerful GPU, and there are several popular methods of utilising GPUs, including CUDA[26] and OpenCL[10]. Using GPUs has lead to orders of magnitude improvement on many important problems, including k nearest neighbour [7], Max-SAT [23] and SAT [20]. One common thread in these papers is that GPU provides the greatest improvements on problems

which can be solved by massively parallel simple calculations. GPUs are not a silver bullet, and direct ports of existing algorithms to GPU often perform poorly.

3 Parallel Consistency and Propagation

In 1990 Kasif [18] showed that the problem of establishing arc-consistency (AC) is P-complete i.e. the problem is not inherently parallelisable under the usual complexity assumptions. This is done by giving log-space reductions of AC to horn-clause satisfaction and vice versa. Kasif shows that in the worst case we cannot establish arc-consistency polynomially faster with a polynomial number of processors. This is no surprise, as we have to do a chain of deductions in arc-consistency, where each depends on (some subset of) the rest. We can read this as being fatal to the enterprise of parallel consistency, but then, it is not fatal to solving constraint problems that they are NP-complete! So we have to take P-completeness into account rather than regard it as fatal.

3.1 Parallel Arc-Consistency

There has been a steady stream of work on distributed consistency algorithms. One of the justifications of this is that the problem itself may be distributed geographically or due to organisational structures (such as in Prosser, Conway and Muller [28]). The other justification is speed.

In 1995 Nguyen and Deville presented a distributed AC-4 algorithm DisAC-4 [24]. A journal version of this work was published in 1998 [25]. The algorithm is based on message passing. The variables are partitioned among the workers, and each worker essentially maintains the AC4 data structures for its set of variables. When a worker deduces a domain deletion, this is broadcast to all other workers. Each worker maintains a list of domain deletions to process (some generated locally and others received from another worker). The worker reaches a fixpoint itself before broadcasting any domain deletions, and waiting for new messages from other workers. The whole system reaches a fixpoint when every worker has processed every domain deletion. It may be a difficult problem to partition the variables such that the work is evenly distributed. The experimental results are mixed, with some experiments showing close to linear speedup, while others show only 1.5 times speedup with 8 processors. In 2002 Hamadi [13] presented an optimal distributed AC algorithm, DisAC-9, optimal with respect to message passing whilst outperforming the fastest centalized algorithms. Therefore most, if not all, seem to demonstrate that the P-completeness of arc-consistency is indeed non-fatal.

3.2 Parallel Propagation of Non-Binary Constraints

In 1998, Ruiz-Andino, Araujo, Sáenz and Ruz [32] presented a distributed propagation algorithm for n-ary functional constraints. These constraints are represented as *indexicals*, where for each constrained variable/value pair there is

a set of $(n-1)$-tuples representing the support set for that pair. The CSP is split into n subsets such that each constraint appears in exactly one subset. If a variable is associated with constraints in more than one set then that variable is duplicated. Each subset is propagated sequentially by its own processor and any domain reductions of variables shared between processors is communicated between processors. The experiments presented show the relative performance gains by increasing the number of cores they make available to their algorithm. First consistency is established, then a variable assignment is made and consistency is re-established. This is repeated until a solution is found or a variable domain is wiped-out. The performance of this technique is highly dependent on the quality of the distribution of the CSP, which is a difficult problem in itself. The conflicting optimisation criteria for quality of a constraint distribution are minimising the network traffic whilst maximising the distribution of the propagation frontier. It appears that this technique will not handle high arity constraints well due to increased communication cost.

Parallel propagation has been proposed for numerical problems, where the variable domains are infinite. Domains are represented as an interval using two floating-point numbers, and the objective of propagation is to narrow the intervals. Although we are ignorant of this area of constraint programming, we would like to mention one paper. In 2000, Granvilliers and Hains [9] proposed a parallel propagation algorithm for non-linear constraints. This was evaluated on a Cray multiprocessor. The gain from using 64 processors (compared to 1 processor) varies from almost nothing to about 6 times, depending on the problem instance.

4 Multi-agent Search

Assuming we have n processors: a speedup of less than n is sub-linear, equal to n linear, greater than n super-linear. Probably the first report of super-linear speed up is due to Rao and Kumar [29] in parallel depth-first search on the 15-puzzle. They argue that if all solutions are uniformly distributed about the state space then average speedup can be superlinear.

The next body of work to report this phenomena was multi-agent search. In multi-agent search we have one problem and a collection of problem solving agents. Each agent is capable of solving the problem independently, agents may be different, and agents might communicate. The agents then work on their copy of the problem, possibly communicating with each other. One of the earliest examples of this is due to Clearwater *et al* [5]. To demonstrate the power of cooperative problem solving the Dynamics of Computation Group investigated the time to solve word puzzles, posed as constraint satisfaction problems, using a collection of agents. Each agent could solve the problem independently. Agents wrote "hints" to a shared blackboard, and agents randomly read hints whilst solving the problem. As the number of agents increases, and the diversity amongst agents increases a "combinatorial implosion" occurs with a subsequent super-linear speed up in problem solving. They present as an explanation of this

phenomena "... the appearance of a lognormal distribution in the effectiveness of an individual agent's problem solving. The enhanced tail of this distribution guarantees the existence of some agents with superior performance." The idea of multi-agent search was further explored in the portfolio-based search proposed by Gomes and Selman [8].

The SAT community has been quick to exploit multi-agent search. ManySAT [12] exploits the main weaknesses in DPLL solvers, i.e. their sensitivity to parameter tuning, to create a population of diverse SAT solvers. The SAT solvers are then allowed to share conflict-clauses discovered during search. In essence ManySAT is an invocation of Clearwater, Huberman and Hogg's cooperative problem solving strategy, but rather than share hints agents share nogoods, i.e. facts as to where solutions cannot exist. A similar approach is reported in [17]. The technique is to run multiple independent SAT solver instances at the same time. They may search the same search spaces, at least partially, but this is slightly avoided because they share learned clauses. Specifically, SAT solvers are spawned in a grid and when they eventually fail after timeout, they return learned clauses to the master. The master then doles out more work, but gives subsequent processes some clauses learned by previous processes. The advantages are that few changes are required of solvers to fit into this framework, just the ability to print out their learned clauses at the end. The most successful strategy for choosing what clauses to share is to pick the shortest available ones. The best strategy in terms of nodes is to the pick the clauses that are independently learned by the most different spawned solvers. An analysis of the heavy tail phenomenon of solver runtimes shows that this technique can win big because each solving attempt is randomized compared to previous ones, as well as the fact clauses are shared. It appears that multiple instances with sharing is more successful than just multiple instances. A question not really answered is that the authors don't show if their solver is work efficient, i.e. it wins on wallclock time, but does it win in terms of total hours of effort? If it only wins on wall clock time, the technique is only useful for very hard instances, where a short amount of time is available to get an answer.

In Bordeaux *et al* [2] the context is "massively parallel constraint solving", by which the authors seem to mean 64+ cores. In this context, they are concerned about the cost of communication, and explore approaches where communication can be avoided. Their first idea is to perform search space splitting by adding "hashing" constraints to the original model, which are unique for each core and so sub-divide the search space. Promising results are reported. They then consider portfolio approaches, rightly pointing out that, for arbitrary numbers of cores, the generation of a portfolio has to be automated. To do this sources of "variability" are necessary, for which they identify three desirable qualities: scalable (different settings will result in different runtimes); favourable (varying from the default does not systematically worsen performance); solver-independent (exploiting the features of individual solvers prohibits re-usability). The variable ordering provides such a source of variability. Results (this time on 128 cores) are again very promising.

Plingeling is the parallel version of lingeling [1]. It won the SAT Race 2010 special track for parallel solvers. It takes the sequential solver called lingeling and runs N (mostly) independent instances of that solver. The first solver to return a definitive answer wins the day. Each solver is given a different random seed, different amounts of effort allocated to preprocessing, and a different initial variable and value ordering (which determines the value tried when first assigning a variable). When solvers learn unit clauses (i.e. assignments or non-assignments) they share them with the master thread, which in turn sends them to the other workers. Hence, apart from the sharing of unit clauses, this state of the art parallel solver (as of 2010) depends on the fact that at least one sequential solver can solve the problem quickly.

5 Parallelizing the Search Process

By "parallelizing the search process" we mean search parallelism at the granularity of nodes or search states, hence each worker is close to being a standard sequential constraint process but they are collectively orchestrated to be part of the same overall search process. We refer below to both local and complete search. For local search, parallel search has been used to increase the number of starting points available in the local search [21]. For complete backtracking search, the issues we will highlight include:

- how the search space is divided between workers;
- how workers communicate what portions of search they have completed and what new solutions and improvements to their optimisation function they have found;
- how state is shared (if appropriate);
- how learned constraints are shared between workers (if learning is implemented); and
- specific implementation details and abstractions.

We describe both CSP and also CNF SAT solvers.

One of the earliest reports on parallel search is due to Bill Clocksin's DelPhi principle [6]. The context here is Prolog, which explores AND-OR trees, but the ideas are equally applicable to the OR-trees common in constraint solving. The motivation for this paper is to avoid the overhead incurred by having a shared memory or copying computation state between processors. The central idea is to associate a processor with each *path* through the search tree, hence avoiding overhead due to transferring state between processors mid-path. However this seems like a false economy as it results in duplicated work (think of two branches that differ only at the very bottom). Usually we don't have enough processors to allocate one to each of the possible paths. So if we have n processors we explore all $\log_2 n$ paths. If we find a solution, terminate otherwise consider the $\log_2 k$-bit extensions to these paths (where k is the number of processors we have left - there is some art and strategy to this as k will continue to increase as the original $\log_2 n$ bit paths are explored) and continue. In fact to continue DelPhi search

starts from scratch again in order to be in the right state (the paths are stored very compactly as bit strings).

Perron was one of the first to report on parallel search in a commercial Constraint Programming toolkit [27], the 1999 update of ILOG's Solver toolkit. The search space is represented as a tree of nodes/search states partitioned into an explored part, a frontier and an unexplored part. New states are entered using recomputation, i.e. make a sequence of decisions to reach the starting point then begin search. This is general enough to allow more exotic search algorithms than DFS but also allows nodes to be allocated to different processors on a shared memory machine. Each processor runs its own search process exploring different parts of this common search tree, with a communications process ensuring work balancing and termination detection. Empirical evaluation was on a 4 processor machine using jobshop scheduling problems, i.e. optimisation problems rather than search for first solution. When using quasi-complete search (variants of LDS) parallelism showed a linear speed up, but with depth first search improvements were less convincing. Therefore results are less than conclusive.

In 2000 Schulte reports on parallel search made simple [34]. Schulte shows how parallel search by work splitting can be achieved using Oz/Mozart, a system which contains constraint reasoning and concurrency, making it a natural implementation choice. The basic unit of computation is the search node (called space in this paper). This is a natural choice because the underlying goal of the paper is to exploit computational resources that may be widely available but also idle, across different machines. Thus it is reasonable to assume that memory is not shared between computational units, and that communication costs between units are relatively large although not prohibitive (i.e. ethernet). The approach is therefore work-sharing, mediated by a single manager, and with a coarse granularity. Results show close to linear speedup (between a 4 and 50% overhead associated with sharing), although on up to only six workers, which by now is a number of cores that a single machine might have, instead of spread across a lab environment.

In 2006 Michel, See and Van Hentenryck presented parallel local search in COMET [21] and deal with distribution (over multiple machines). The paper describes an architecture for distributing the work over multiple (heterogeneous) machines. The distribution of tasks is intended to be nearly independent of the COMET code that describes the search. The COMET code given as examples in the paper describes various local search strategies including genetic algorithms and constraint-based local search [15]. The examples given all work by distributing different starting points in the local search space to different machines, rather than executing different starting points sequentially on one machine. The experimental results demonstrate close to linear speedups. In 2007 they extend their work in [22]. The COMET constraint programming toolkit is enhanced to allow multicore parallel search. This is done "under the hood" so that a constraint programmer need not know that it is taking place or how it is implemented. Each processor supports a worker. When a worker explores, i.e. expands a current search node, it produces new unexpanded search nodes,

where an unexpanded node is a self-contained subproblem specified as a semantic path. Parallel COMET uses a technique called *work stealing* where workers who have run out of work take unexpanded nodes from other workers, leaving them less work to do and keeping all workers busy. It is implemented as follows: Search nodes are then represented as continuations [30] and are added to a central pool. When workers are idle they steal continuations from the central pool, and this is synchronized; in the case of optimization problems workers communicate new bounds on the objective function, again synchronized. Experiments were performed on NQueens, Scene Allocation, Graph Colouring and Golomb Rulers using depth first search and limited discrepancy search (LDS) on 1 to 4 processors. Speed ups were a bit less than linear, although superlinear speedups occured with LDS and this was attributed to the order that continuations were stolen, disrupting the normal search order.

We will now describe a couple of papers on parallelisation of CNF SAT solvers. In these papers, the emphasis is on how learned clauses are shared, since learning is an important component of SAT solvers which is moreover tough to parallelise due to the fact a large amount of memory and time spent processing learned clauses. Hence various techniques are used to encourage only clauses that are likely to propagate elsewhere are shared.

In 2008 Chu and Stuckey presented PMiniSAT [4]. MiniSat 20.0 is parallelized using work stealing. The solver uses several techniques for sharing clauses between threads. Firstly, all clauses with length beneath a certain threshold are shared between *all* threads; this is a previously used technique. Secondly, and this is a new idea, a new measure called *effective length* is used: the effective length of a learned clause is evaluated per thread, and it is the number of non-false literals in the clause, i.e. the number of literals (minus one) that still need to be set false for the clause to unit propagate. Only clauses whose effective length is less than a particular threshold are accepted by other threads. Hence clauses are preferentially shared between threads searching "close" paths in search. Third, another new idea, a thread is able to store clauses in a suspended state until it is working on another chunk of work where it is unit. Unfortunately the details of this are very sketchy and no implementation is described. Together these techniques are somewhat similar to cooperation between agents, i.e. rather than share hints on where solutions might lie, share facts where solutions cannot exist.

In 2009 we have PaMiraXT [33], a hybrid multicore and multicomputer SAT solver. On each available computer it uses all available cores, and it uses all available computers. The whole system searches a single SAT model using the *guiding path* algorithm. This means that they all search different parts of the same tree and steal work from each other when one process runs out. As this is about SAT, a lot of the paper is concerned with how to share clauses efficiently and effectively. On each computer, the solvers all share a single clause database and they all propagate all the clauses of the other processes. The clauses are all stored in a shared read only memory segment and the solvers keep their watches in their own memory space, to avoid contention. They also keep some extra pointers to literals they believe are more likely to be unset, and apparently in

85% of propagations they don't need to look at the shared clause DB at all. To share work within a computer, there is a controlling process which steals work and passes it to idle processes. The work stealing technique steals work near to the root, i.e. the first left branch has its right branch stolen. The process that is cannibalised is the one with the shortest guiding path, i.e. the fewest leading closed assignments in its partial assignment. This is done because it intuitively corresponds to a larger chunk of the search space. Between computers, the clauses pass clauses of length 3 and under, which are integrated into the clause DBs on each machine. These are passed using MPI in bundles of 50 clauses. The work stealing between computers is elegant. The exact same mechanism is used as within computer, i.e. a process requests some work from the controller, but instead of solving it directly it just sends it to another computer for solving. The experiments suggest that it's a good technique. The solver is very fast on a single core, which obviously helps. When it is applied to multiple cores and computers usually adding another core speeds up the wallclock time. Unfortunately total solve time is not provided because experiments, are based on wallclock time. Hence this work makes finding a solution faster, but it is not possible to tell how work-efficient it is.

In [3] Confidence-based Work Stealing in Parallel Constraint Programming is presented. The authors point out that work stealing not only allows load balancing, but also influences the order in which parts of the search tree are explored. In the papers described above the strategy was usually to steal from as close to the root as possible. The paper includes a examples showing how it can sometimes be much better to steal "left and low" (i.e. as deep as possible) but sometimes stealing high can be better. The message of course is that it should be dynamic. They claim that the presence of a branching heuristic complicates the process of finding a good stealing strategy. Consequently the plan is to steal based on confidence: the estimated distribution of solution densities among the children of a node. When doing binary branching, this is equivalent to confidence in the heuristic (since it chooses that left branch). Ideally the user should provide a confidence function for the heuristic, but the authors show how a simple substitute can still work when this is not available. Confidence values are updated during search. However, in practice they found that stealing too low tends to increase the communication cost. Hence they set a bound above the average fail depth below which worked cannot be stolen. Experimental results demonstrate the technique's effectiveness ranging from speedups of 7 times to superlinear on the benchmarks in the paper.

Kotthoff and Moore [19] describe an implementation technique for distributed search with no sharing between workers. The implementation assumes a job queueing system is available as a substrate. A standard solver is wrapped in a script that does the following: (1) run until time limit reached or completion (1.1) if solution found then terminate all jobs and return solution (1.2) if search space complete then do nothing and stop (1.3) else split the variable we are currently trying to assign into n parts (where n is a constant) (1.3.1) to each job add a set of restart nogoods that rules out the search space explored so far in current

solver (1.3.2) submit each slice to the job server. Hence the technique is similar to work stealing but less dynamic – work cannot be stolen until a time limit is reached. The novelty in this approach is twofold: (a) using a job server to avoid implementing distribution, fault tolerance, etc.; and (b) using restart nogoods rather than recomputation to rule out previously explored search. The former should be self-explanatory. The latter may not be: Recomputation implicitly assumes that the parent will continue to search and it is giving away only a part of its search space. This technique works instead by the parent giving up all its search space and splitting itself into n parts. Restart nogoods ensure that the child processes are together given the *remaining* part of the parent search space, and made to search different portions of it by the splitting constraints. It is not determined if this was actually better than recomputation, it is merely a convenient implementation. It is more flexible however, in the sense that the children are allowed to search the remaining space any way they wish, instead of always having to stick to the early part of the parent's variable ordering. This allows each solver to use a different search strategy, but no experiments were carried out doing this.

6 Combined Parallel Search and Parallel Consistency

We finish this review with a single paper, probably one that best represents the state of the art [31]. This is an extension of the same authors' 2009 paper. The object of this work is to parallelise both search and consistency. By the latter they mean splitting the set of constraints to be propagated among threads, rather than parallelising the work of a single constraint. They begin with an example demonstrating that a simple search parallelism scheme is at the mercy of the location of the solution(s) in the search tree - if they are all going to be found by the first thread anyway, the others are just adding overhead. They introduce some terminology: parallel search = OR parallelism = data parallelism and parallel consistency = AND parallelism = task parallelism. They claim that, for many models, solvers spend an order of magnitude more time enforcing consistency than they do searching, in which case data parallelism is less suitable. Another flaw is that data parallelism naturally puts more stress on the memory bus (for this they point to Sun & Chen [36]). In their approach to parallel consistency, they require synchronization of pruning, but do not share data during pruning to avoid upsetting the internal data structures of global constraints. Rather than fixing which threads deal with which constraints, at each node each consistency thread takes a set of constraints to propagate from the queue. When all constraints in the queue have been processed, updates are actually committed. The process can stop early if one of the threads detects inconsistency. When combining both parallel search and consistency, each search thread gets an associated set of consistency threads. An alternative architecture is briefly discussed in which all threads take from a shared work pool, but the authors claim that scheduling uptake from this pool could be prohibitively complex. Experiments are on Sudoku and n-queens using up to 64 threads on

8 cores. The gains are modest. They identify three problems: inefficiencies in parallel consistency caused by not sharing data, the synchronization of pruning described, and third the memory bus.

7 Conclusion

This paper has presented a preliminary survey of the literature on parallel constraint solving. Despite early pessimism based upon Amdahl's law and Kasif's proof of P-completeness, more recent results using smart work stealing, partitioning of the search space and multi-agent approaches give considerable cause for optimism. Irrespective, the apparent trend is that multicore computing is the norm, with large increases in the number of cores in the immediate future. Whether we would prefer the comfort of a fast single-processor world or not, therefore, we must embrace this paradigm.

References

1. A. Biere. Lingeling, Plingeling, PicoSAT and PrecoSAT at SAT race 2010. In *Technical Report 10/1, Institute for Formal Models and Verification, Johannes Kepler University*, 2010.
2. L. Bordeaux, Y. Hamadi, and H. Samulowitz. Experiments with massively parallel constraint solving. In *IJCAI2009*, pages 443–448, 2009.
3. G. Chu, C. Schulte, and P.J. Stuckey. Confidence-based Work Stealing in Parallel Constraint Programming. In *Principles and Practices of Constraint Programming*, pages 226–241, 2009.
4. G. Chu and P.J. Stuckey. PMiniSAT: A Parallelization of MiniSAT 2.0. In *SAT Race 2008*, 2008.
5. Scott H. Clearwater, Bernardo A. Huberman, and Tad Hogg. Cooperative Solution of Constraint Satisfaction Problems. *Science*, 254, 1991.
6. W. Clocksin. Principles of the DelPhi parallel inference machine. *Computer Journal*, 30(5):386–392, 1987.
7. Vincent Garcia, Eric Debreuve, and Michel Barlaud. Fast k nearest neighbor search using gpu. *Computer Vision and Pattern Recognition Workshop*, 0:1–6, 2008.
8. C.P. Gomes and B. Selman. Algorithm Portfolios. *Artificial Intelligence*, 126(1-2):43–62, 2001.
9. Laurent Granvilliers and Gaétan Hains. A conservative scheme for parallel interval narrowing. *Information Processing Letters*, 74(3-4):141–146, 2000.
10. Khronos Group. Opencl. `http://www.khronos.org/opencl/`.
11. J. L. Gustafson. Revaluating Amdahl's law. *Comm ACM*, 31, Number 5:532–533, 1988.
12. Y. Hamadi, S. Jabbour, and L. Sais. ManySAT: Solver Description. In *SAT Race 2008*, 2008.
13. Youssef Hamadi. Optimal Distributed Arc-Consistency. *Constraints*, 7(3-4):367–385, 2002.
14. J. Held, J. Bautista, and S Koehl. From a few cores to many: A tera-scale computing research overview. *intel White Paper*, 2006.
15. Pascal Van Hentenryck and Laurent Michel. *Constraint-Based Local Search*. The MIT Press, 2009.

16. M.D. Hill and M.R. Marty. Amdahl's law in the Multicore Era. *IEEE Computer*, pages 33–38, 2008.
17. A.E.J. Hyvärinen, T. Junttila, and I. Niemelä. Incorporating clause learning in grid-based randomized sat solving. *JSAT*, 6, 2009.
18. Simon Kasif. On the parallel complexity of discrete relaxation in constraint satisfaction networks. *Artificial Intelligence*, pages 275–286, 1990.
19. L. Kotthoff and N.C.A. Moore. Distributed Solving Through Model Splitting. In *TRICS 2010*, 2010.
20. Panagiotis Manolios and Yimin Zhang. Implementing Survey Propagation on Graphics Processing Units. In *Theory and Applications of Satisfiability Testing – SAT 2006*, volume 4121/2006 of *LNCS*, pages 311–324, 2006.
21. L. Michel, A. See, and P. Van Hentenryck. Distributed Constraint-based local search. In *Principles and Practices of Constraint Programming*, pages 344–356, 2006.
22. Laurent Michel, Andrew See, and Pascal Van Hentenryck. Parallelizing Constraint Programs Transparently. In *Principles and Practices of Constraint Programming*, pages 514–528, 2007.
23. Asim Munawar, Mohamed Wahib, Masaharu Munetomo, and Kiyoshi Akama. Hybrid of genetic algorithm and local search to solve max-sat problem using nvidia cuda framework. *Genetic Programming and Evolvable Machines*, 10:391–415, December 2009.
24. Thang Nguyen and Yves Deville. A distributed arc-consistency algorithm. In *First International Workshop on Concurrent Constraint Satisfaction*, 1995.
25. Thang Nguyen and Yves Deville. A distributed arc-consistency algorithm. *Science of Computer Programming*, 30:227–250, 1998.
26. NVIDIA. Cuda. `http://www.nvidia.com/object/cuda_home_new.html`.
27. L. Perron. Search Procedures and parallelism in constraint programming. In *Principles and Practices of Constraint Programming*, pages 346–361, 1999.
28. P. Prosser, C. Conway, and C. Muller. A constraint maintenance system for the distributed resource allocation problem. *Intelligent Systems Engineering*, pages 76–83, 1992.
29. V.N. Rao and V. Kumar. Supelinear speedup in parallel state-space search. In *8th Conference on Foundations of Software Technology and Theoretical Computer Science FSTTCS (LNCS 338)*, pages 161–174, 1988.
30. John C. Reynolds. The discoveries of continuations. *Lisp Symb. Comput.*, 6:233–248, November 1993.
31. C.C. Rolf and K. Kuchcinski. Combining parallel search and parallel consistency in constraint programming. In *TRICS workshop at CP2010*, pages 38–52, 2010.
32. Alvaro Ruiz-Andino, Lourdes Araujo, Fernando Sáenz, and José J. Ruz. Parallel Arc-Consistency for Functional Constraints. In *Workshop on Implementation Technology for Programming Languages based on Logic, ICLP*, pages 86–100, 1998.
33. T. Schubert, M. Lewis, and B. Becker. PaMiraXT: Parallel SAT solving with threads and message passing. *JSAT*, 6:203–222, 2009.
34. Christian Schulte. Parallel Search Made Simple. In *TRICS2000*, pages 41–57, 2000.
35. X.H. Sun and L.M. Ni. Another View of Parallel Speedup. In *Proc Supercomputing*, pages 324–333, 1990.
36. Xian-He Sun and Yong Chen. Reevaluating Amdahl's Law in the multicore era. *Journal of Parallel and Distributed Computing*, 70:183–188, 2010.
37. W.A. Wulf and S.A. McKee. Hitting the memory wall: implications of the obvious. *ACM SIGARCH*, 23, Issue 1:20–24, 1995.