Parallel Search Made Simple

Christian Schulte

Programming Systems Lab, Universität des Saarlandes
Postfach 15 11 50, 66041 Saarbrücken, Germany
schulte@ps.uni-sb.de

Abstract. Search in constraint programming is a time consuming task. Search can be speeded up by exploring subtrees of a search tree in parallel. This paper presents distributed search engines that achieve parallelism by distribution across networked computers. The main point of the paper is a simple design of the parallel search engine. Simplicity comes as an immediate consequence of clearly separating search, concurrency, and distribution. The obtained distributed search engines are simple yet offer substantial speedup on standard network computers.

1 Introduction

Search in constraint programming is a time consuming task. Search can be speeded up by exploring several subtrees of a search tree in parallel by cooperating search engines called workers.

The paper develops search engines that achieve parallelism by distributing workers across standard networked computers. The paper has two main points. The first point is to provide a simple, high-level, and reusable design for parallel search. The second point is to obtain good speedup rather than good resource utilization.

Simple and Reusable Design Parallel search is made simple by separating three issues: search, concurrency, and distribution.

Search Workers are search engines that explicitly manipulate their state. The state corresponds to yet to be explored subtrees of the search tree. Explicit manipulation is mandatory since subtrees need to be shared between workers.

Concurrency The main contribution of the paper is the design of a concurrent search engine that adds communication and cooperation between workers. Communication and cooperation presupposes concurrency.

Distribution How workers are distributed across networked computers is considered independently of the architecture of the concurrent engine. An important technique for sharing nodes across the network is recomputation.

The approach obviously simplifies the design since it allows to address concerns independently. It also allows to reuse the concurrent architecture for other purposes, such as parallel execution on shared-memory multiprocessors and cooperative search for multi-agent systems.

The approach presupposes that search is encapsulated and combines well with concurrency and distribution. As implementation language we use Oz [19], a concurrent language that supports distribution and programming of search engines from computation spaces [16]. Since spaces are concurrency-enabled, parallel search engines can be programmed entirely in Oz.

The distributed search engines described in this paper are implemented in the Mozart [10] implementation of Oz. The programming effort needed is less than one thousand lines of Oz code.

Obtaining Speedup Networked computers are cheap, ubiquitous, and mostly idle. Hence our criterion of success is whether a simple distributed search engine can offer substantial speedup. This differs from the traditional criterion of success for parallel search that aims at good utilization of specialized, expensive, and not widely available hardware.

A performance evaluation shows that the simple distributed engine offers substantial speedup already for small search trees. Large search trees as common for complex constraint problems provide almost linear speedup.

Related Work There has been considerable work in the area of parallel search. Rao and Kumar discuss and analyze the implementation of parallel depth-first search in [14,8]. Their focus is on the impact of the underlying hardware architecture and in particular how to best utilize the resources of the parallel architecture. Parallel execution on shared-memory multiprocessors and to a lesser extent on networked computers has received great attention in logic programming, for an overview see [1]. Early work that also uses recomputation to distribute work is the Delphi Prolog system by Clocksin and Alshawi [2,3].

Mudambi and Schimpf discuss in [11] distributed search that also relies on recomputation. A refinement of this work also addresses branch-and-bound search [13]. Perron briefly sketches parallel search for ILOG Solver in [12]. All these approaches have in common that they are mostly focused on the description how each separate engine works. The discussion of the architecture by which the parallel engines communicate is missing or is at a low-level of abstraction. In contrast, we are concerned with developing a high-level concurrent architecture underlying parallel search engines.

Plan of the Paper Section 2 gives an overview of programming search, concurrent programming, and distributed programming. The architecture for concurrent search engines is introduced in Section 3, followed by how the concurrent search engines are distributed across networked computers. The distributed search engines are evaluated in Section 5.

2 Programming

This section gives an overview of the three different paradigms that are essential for programming distributed search engines: programming search, concur-

rent programming, and distributed programming. The underlying programming model that supports all three paradigms is the Oz programming model [19].

2.1 Programming Search

Most constraint programming systems come with a fixed set of predefined search engines that cannot be extended by the user. Our approach is different in that it makes search programmable by the user: the appropriate primitives are integrated into a programming language.

Oz offers computation spaces as primitives to program search. A more detailed treatment of programming search engines from computation spaces can be found in [16]. Spaces are applied to visual and interactive search in [15] and to concurrent deep-guard combinators in [18].

We focus on search engines that explicitly maintain their state as a collection of nodes (spaces) that require exploration. Having access to the nodes is mandatory for concurrent search engines since nodes are subject to sharing between several concurrent search engines (workers).

`NewSpace`	: *Script* → *Space*
`Ask`	: *Space* → *Status*
`Access`	: *Space* → *Solution*
`Clone`	: *Space* → *Space*
`Commit`	: *Space* × *Int* → *Unit*
`Inject`	: *Space* × *Script* → *Unit*

Fig. 1. Operation on computation spaces.

Spaces Spaces are first-class entities in the programming language. The operations on computation spaces are listed in Figure 1. `NewSpace` takes a script (a procedure that defines the constraint problem to be solved) and returns a space that executes the script. `Ask` synchronizes until computation in the space has reached a stable state. It then returns the status of the space, that is, whether the space is `failed`, `solved`, or has `alternatives`. Alternatives are then resolved by search. `Access` returns the solution stored in a space. `Clone` returns a copy of a space. `Commit` selects an alternative of a choice point. `Inject` adds constraints to a space. How the operations are employed for programming search becomes clear in the following.

Programming Exploration Figure 2 shows a formulation of depth-first exploration (DFE) that explicitly maintains the state as a stack of spaces (implemented as list). `DFE` returns either the empty list, if no solution is found, or a singleton list containing the solution. If a space needs to be resolved by search, the space is copied (by application of `Clone`) and exploration follows the left alternative (`{Commit S 1}`) and later the right alternative (`{Commit C 2}`).

```
fun {DFE Ss}
   case Ss of nil then nil
   [] S|Sr then
      case {Ask S} of failed then {DFE Sr}
      [] solved then [{Access S}]
      [] alternatives then C={Clone S} in
         {Commit S 1} {Commit C 2} {DFE S|C|Sr}
      end
   end
end
```

Fig. 2. Depth-first exploration with explicit state.

The complete search engine is obtained by adding space creation according to the problem `P` to be solved:

```
fun {SearchOne P}
   {DFE [{NewSpace P}]}
end
```

Branch-and-bound Search Best-solution search determines a best solution with respect to a problem-dependent ordering among all solutions. The art of best-solution search is to prune the search space as much as possible by information computed from previously found solutions.

The following procedure injects information on a previously found solution `Sol` (passed as singleton list) into a space `S`. The binary procedure `O` implements the order between solutions.

```
proc {Constrain S [Sol]}
   {Inject S proc {$ NR} {O Sol NR} end}
end
```

Branch-and-bound search organizes the spaces to be explored into two stacks: the *foreground stack* (f-stack) and the *background stack* (b-stack). Spaces on the f-stack are guaranteed to yield a better solution. Spaces that are not known to guarantee this invariant are on the b-stack. The engine can be characterized by how it maintains the invariants for the two stacks:

- Initially, the b-stack is empty and the f-stack contains the root space.
- If the f-stack is empty and the b-stack contains S, S is moved to the f-stack after constraining.
- If a better solution is found, all elements of the f-stack are moved to the b-stack.
- If a space from the f-stack is committed or cloned, it is eligible to go on the f-stack.

Taking these facts together yields the program shown in Figure 3. The procedure `BABE` takes the f-stack (`Fs`), the b-stack (`Bs`), and the so-far best solution (`BS`).

```
fun {BABE Fs Bs BS}
   case Fs of nil then
      case Bs of nil then BS
      [] B|Br then {Constrain B BS} {BABE [B] Br BS}
      end
   [] F|FR then
      case {Ask F} of failed then {BABE Fr Bs BS}
      [] solved then {BABE nil {Append Fr Bs} [{Access F}]}
      [] alternatives(N) then C={Clone F} in
         {Commit F 1} {Commit C 2#N} {BABE F|C|Fr Bs BS}
      end
   end
end
```

Fig. 3. Branch-and-bound exploration with explicit state.

2.2 Concurrent Programming

Central for the integration of search into a concurrent and distributed setting is that search is encapsulated. Encapsulation means that speculative constraint computations are separated from the concurrent computations that control the search engine. Encapsulation is provided by spaces: failure is kept encapsulated to a space and does not affect other computations.

Active Services Each concurrent search engine is provided as an active and autonomous concurrent service. Each service runs in its own thread and serves messages set to the active service. This simple construction guarantees consistency in a concurrent setting and also allows for straightforward migration from a concurrent architecture to a distributed architecture.

Ports provide message sending for active services. A port maintains an ordered stream of messages ("mailbox"). A `Send`-operation on the port adds a message to the end of the stream. The stream of messages then is incrementally processed as new messages arrive. Ports have been initially conceived in the context of AKL [7].

In our context a search engine is turned into a concurrent worker by attaching a port to the engine. The engine then serves incoming messages. One particular message is `explore`, which recursively continues exploration.

2.3 Distributed Programming

The basic idea of Distributed Oz is to abstract away the network as much as possible. This means that all network operations are invoked implicitly by the system as an incidental result of using particular language operations. Distributed Oz has the same language semantics as Oz by defining a distributed semantics for all language entities. The distributed semantics extends the language semantics to take into account the notion of *site* (or *process*). It defines the network operations invoked when a computation is distributed across multiple sites.

Partial Network Transparency Network transparency means that computations behave the same independent of the site they compute on, and that the possible interconnections between two computations do not depend on whether they execute on the same or on different sites. Network transparency is guaranteed in Distributed Oz for most entities. While network transparency is desirable since it makes distributed programming easy, some entities in Distributed Oz are not distributable. There are two different reasons for an entity to be not distributable.

- The entity is *native* to the site. Examples are external entities such as files, windows, but also native procedures acquired by dynamic linking. Native procedures depend on the platform, the operating system, and also on the process. Particular examples for native procedures in Mozart are most constraints which are implemented in C++ rather than in Oz [9].
- Distribution would be too complex. One class of entities for which distribution is too complex are computation spaces. Furthermore, even a distributed implementation of computation spaces would be of limited use, since a computation space typically contains native propagators.

Resource Access For distributed computations that need to utilize resources of a distributed system, it is important to gain access to site-specific resources. Access is gained by dynamic linking of *functors* that return *modules* in Oz. Dynamic linking resolves a given set of resource-names (which are distributable) associated with a functor and returns the resources (which are site-specific).

A straightforward way to access site-specific resources is accessing them through active services. The service is distributable while its associated thread is stationary and remains at the creating site. Thus all resource accesses are performed locally. Services by this resemble remote procedure call (RPC) or remote method invocation (RMI).

Compute Servers An Oz process can create new sites acting as *compute servers*. Compute server creation takes the Internet address of a computer and starts a new Oz process with the help from operating services for remote execution. The created Oz process can be given a functor for execution. Thus the functor gives access to the remotely spawned computations. Typically, a functor is used to set up the right active services and to get access to native resources.

Further Reading An overview on the design of Distributed Oz is [6]. A tutorial account on distributed programming with Mozart is [20]. The distributed semantics of logic variables is reported in [5]; the distributed semantics of objects is discussed in [21]. More information on functors, dynamic linking, and module managers in Mozart can be found in [4].

3 Architecture

The concurrent search engine consists of a single *manager* and several *workers*. The manager initializes the workers, collects solutions, detects termination, and

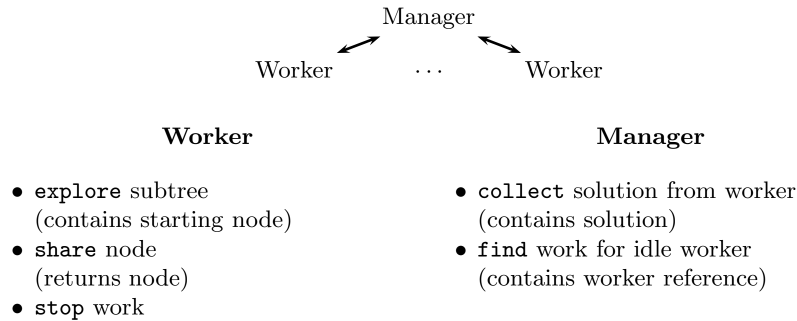

Fig. 4. Architecture of concurrent search engine.

assists in finding work for workers. Workers explore subtrees, share work with other workers, and send solutions to the manager.

3.1 Cooperation

Manager and workers are understood best as concurrent autonomous agents that communicate by exchanging messages. The architecture of the concurrent search engine composed from manager and workers is sketched in Figure 4.

Initialization The concurrent search engine is initialized on behalf of the manager. The manager sends an `explore`-message for the root node of the search tree to a single worker. This single worker then starts working by exploring. A worker that currently explores a subtree is *busy* and *idle* otherwise.

Exploration A worker explores nodes of the search tree. By working it generates new work (new nodes).

Finding Work Suppose that worker W_i is idle. It announces this fact to the manager by sending a `find`-message. The manager then tries to find a busy worker W_b that is willing to share work with W_i. If the manager finds work, it informs W_i by sending an `explore`-message containing the work found. To allow communication back from the manager to W_i, the `find`-message contains a reference to W_i.

The manager maintains a list of possibly busy workers which are not known to be idle since the manager has not received a `find`-message from them. From this list the manager picks a worker W_b and then sends a `share`-message to W_b.

When W_b receives a `share`-message, it first checks whether it has enough work to fulfill the request. A worker receiving a `share`-message can be unable or unwilling to share work. It can be unable, because it is idle. It can be unwilling, because it has so little work such that sharing it might make the worker idle itself (for example, the worker has only a single node left). In case the worker is willing to share work, it removes a node from its own pool of work and sends it

to the manager. When the manager receives the node, it forwards the node to the requesting worker.

If the manager receives that a `share-message` has been unsuccessful, it tries the next busy worker. If all busy workers have been tried, it starts over again by re-sending the initial `find`-message.

Collecting Solutions When a worker finds a solution, it sends a `collect`-message containing the solution to the manager.

Termination Detection The manager detects that exploration is complete, when the list of presumably busy workers becomes empty.

Stopping Search If the search tree needs partial exploration (for example, single-solution search) the manager can stop search by sending a `stop`-message to all workers.

Almost all communication between manager and workers is asynchronous. The only point where synchronization is needed, is when the manager decides whether finding work has been successful. This point is discussed in more detail in Section 3.3.

Important Facts The concurrent search engine does not loose or duplicate work, since nodes are directly exchanged between workers. Provided that the entire tree is explored, the number of exploration steps performed by the concurrent engine is the same as by the standard depth-first engine.

The exploration order is likely to be different from left-most depth-first. The order depends on which nodes are exchanged between workers and is indeterministic. For all-solution search this has the consequence that the order in which the manager collects solutions is indeterministic. For single-solution search this has the consequence that it is indeterministic which solution is found. In addition, it is also indeterministic how many exploration steps are needed. The number can be smaller or greater than the number of exploration steps required by depth-first exploration. The phenomenon to require less steps is also known as super-linear speedup.

3.2 Worker

A worker is a search engine that is able to share nodes and that can be stopped. Figure 5(a) summarizes which messages a worker receives and sends. The ability to share work requires explicit state representation. A worker knows the manager and maintains a list of nodes that need exploration ("work pool").

Concurrent Control The worker is implemented as an active service. It runs in its own thread and sequentially serves the messages it receives. This simple design is enough to ensure consistency of the workers state in a concurrent setting.

The worker recursively invokes exploration as sketched in Section 2.1 by sending an exploration message to itself. By message sending, exploration and communication with the manager is easily synchronized.

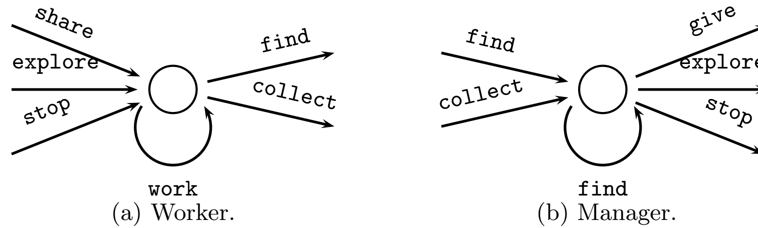

Fig. 5. Summary of messages.

Which Node to Share A promising candidate is the highest node in the search tree, since it is likely that the subtree starting from it is large ("large work granularity"). A large subtree prevents that the requesting worker becomes idle soon and thus prevents excessive communication. Later it becomes clear that sharing the highest node is a particularly good choice for distribution.

3.3 Manager

The manager is implemented as an active service, the messages it sends and receives are summarized in Figure 5(b). The manager knows all workers. They are needed for initialization and for stopping. The manager maintains a list of workers not known to be idle and a list of solutions.

Finding Work Finding work can be a time consuming task since it can take several attempts to seek for a worker that is able to share work. Hence, it is infeasible to block the manager while seeking work.

A design that does not block the manager is as follows. When the manager receives a `find`-message, it spawns a new thread that takes the current list of busy workers as snapshot. Directly after thread creation the manager is again available to serve incoming messages. If no work has been found, the initial `find`-message is send again to the manager and the thread terminates. This is repeated until either work is found or no presumably busy workers are left.

The solution to take a snapshot of the currently busy workers upon message receipt is simple but has the following drawback. The manager might ask workers that are still contained in the snapshot but have already announced that they are idle themselves. This can result in a delay of the manager to find work and thus the initially requesting worker might remain idle for a longer period of time.

3.4 Best-solution Search

The main design issue in best-solution search is how to maintain the so-far best solution. The sequential branch-and-bound engine always knows the so-far best solution (`BS` in Figure 3). This is difficult to achieve in a concurrent setting with

several workers. Maintaining the best solution for each worker would require a large communication and synchronization overhead. Instead we prefer a solution where both manager and workers maintain the so-far best solution as follows:

Manager When the manager receives a new solution through a `collect`-message, it checks whether the solution is really better. If the solution is better, the manager sends it to all workers. This requires a `better`-message that contains the so-far best solution.

Worker When a worker finds a new solution, it stores the solution as so-far best solution and informs the manager by sending a `collect`-message. When a worker receives a `better`-message the worker does *not* check whether the received solution S_1 is better than the worker's so-far best solution S_2. If S_1 is worse, S_1 will be replaced anyway. The manager eventually sends a solution which is at least as good as S_2 (since it receives S_2 from this worker). It might be better in case the manager has received an even better solution from some other worker.

The architecture sketched above entails that a worker might not always know the so-far best solution. This can have the consequence that parts of the search tree are explored that would have been pruned away if the worker would have had exact knowledge. Thus the loose coupling might be paid by some overhead. To this overhead we refer to as *exploration overhead*.

4 Distributed Search Engines

This section discusses how to adopt the concurrent search engine such that its workers are distributed across networked computers.

Search Engine Setup The setup of the search engine uses compute servers. The manager is created first. Then a new Oz process is created for each worker. Typically, each process is created on a different networked computer.

Each newly created process is given a functor that creates and returns the worker service. It is important that the functor can be given first-class since the worker requires access to the manager service.

Distributing Nodes Since spaces are not distributable, workers cannot exchange work by communicating spaces directly. Scripts are not distributable, since they typically contain references to native propagators. However, a functor that on application returns the script *is* distributable. This means that the root space can be recomputed via the script from the given script-functor by all workers.

Given the root space, work can then be communicated by communicating paths in the search tree that describe how to recompute nodes:

$$\text{node} \quad \longleftrightarrow \quad \text{root} + \text{path}$$

When a worker acquires new work, the acquired node is recomputed. This causes overhead, to which we refer to as *recomputation overhead*. The higher the node in the search tree, the smaller the recomputation overhead. For this reason, sharing the topmost node is a good choice. Since all nodes are subject to sharing, a worker must always maintain the path to recompute a node.

Recomputable Spaces In the following we employ *recomputable spaces* (r-spaces for short) as convenient abstractions for distributed search engines. An r-space supports all space operations. Additionally, an r-space provides an *export* operation that returns the path for recomputation. Search engines that employ r-spaces rather than "normal" spaces are otherwise identical, since r-spaces provide the same programming interface.

The key feature of an r-space is that commit-operations are executed on demand which is beneficial for two reasons. Firstly, not the entire search tree might be explored during single solution search. Secondly, a node might be handed out to some other worker and thus the commit operations might be wasted.

An r-space encapsulates the following three components:

Sliding Space It is initialized to a clone of the root space.
Pending Path A list of pending commit-operations.
Done Path A list of already performed commit-operations.

The sliding space always satisfies the invariant that it corresponds to a space that has been recomputed from the root space and the done path.

Initialization Creation of an r-space takes a path P as input. The sliding space is initialized to a clone of the root space. The pending path is initialized to the path P. The done path is initialized to the empty path.
Commit A commit to the i-th alternative adds i to the pending path.
Update Updating an r-space performs all commit-operations on the pending path. Then the pending path is added to the done path and is reset.
Ask, Clone, Access Ask, clone, and access update the r-space first and then perform the corresponding operation on the sliding space.
Export Export returns the concatenation of done and pending path.

An r-space is extended straightforwardly to support best-solution search by storing a list of operations rather than a simple path. This list of operations contains elements of the form `commit(`i`)` and `constrain(`x`)`, where i is the number of an alternative and x is a solution. This presupposes that solutions are distributable.

5 Evaluation

The examples used for evaluation are all common benchmark problems: Alpha, 10-S-Queens, Photo, and MT 10. The examples vary in the following aspects:

Search Space and Search Cost All but MT 10 have a rather small search space where every exploration step is cheap (that is, takes little runtime).

Strategy For Alpha and 10-S-Queens all-solution search is used. For Photo and MT 10 best-solution search is used.

Number of Solutions 10-S-Queens has a large number of solutions (every tenth node is a solution). This makes the example interesting, because each solution is forwarded to the manager. This assesses whether communication is a bottleneck and whether the manager is able to process messages quickly.

The choice of examples addresses the question of how good parallel search engines can be for borderline examples. MT 10, in contrast, can be considered as a well-suited example as it comes to size and cost.

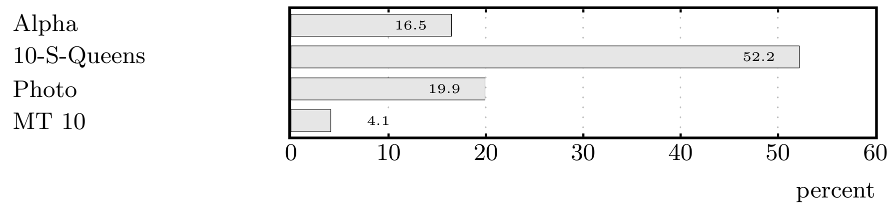

Fig. 6. Total overhead of distributed search engine.

Total Overhead Figure 6 shows the total overhead of a distributed search engine for the examples. The overhead is taken as the additional runtime needed by a distributed search engine with a single worker, where both worker and manager execute on the same computer compared to a sequential search engine. Information on the used software and hardware platforms can be found in Section A.

The numbers suggest that for examples with small search space and small search cost the overhead is less than 20%. This is due to the additional costs for maintaining r-spaces and message-sending to an active service. For large examples (MT 10) the overhead can be neglected. The overhead of around 50% for 10-S-Queens is due to frequent communication between worker and manager. Compared to the small search cost, this overhead is quite tolerable.

Speedup Figure 7 shows the speedup that is obtained for the examples with a varying number of workers. All examples offer substantial speedup. For three workers all examples yield at least a speedup of two, and for six workers the speedup exceeds three. The speedup for MT 10 with six workers is larger than 5.

For all combinations of workers and examples but Alpha with two workers the coefficient of deviation is less than 7% and for 70% of all combinations less than 5% (in particular, for all combinations of MT 10 less than 4%). For Alpha with two workers the coefficient of deviation is less than 11%. This allows to conclude that speedup is stable across different runs and that indeterminism introduced by communication shows little effect on the runtime. Moreover, this clarifies that both minimal and maximal speedup are close to the average speedup.

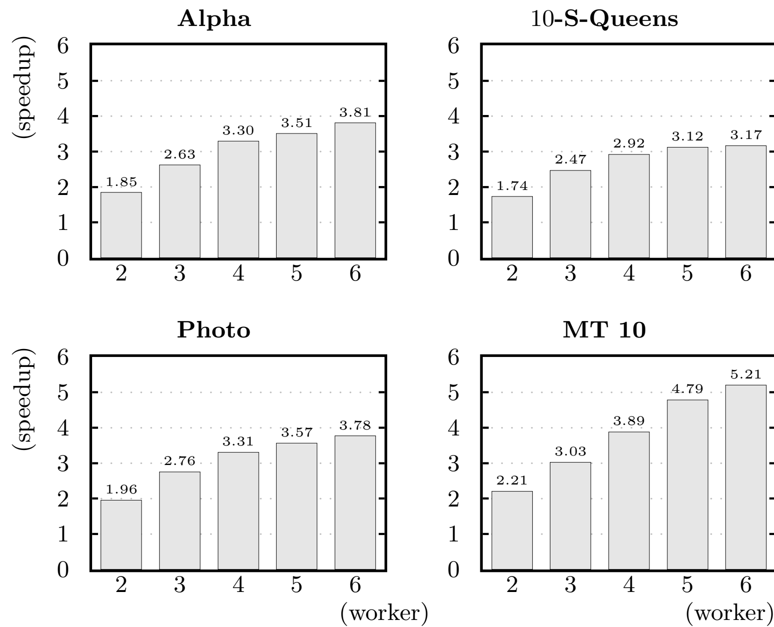

Fig. 7. Speedup.

Work Granularity Figure 8 shows the average work granularity which is amazingly coarse. Work granularity is the arithmetic mean of the sizes of subtrees explored by a worker in relation to the size of the entire tree. For all combinations of examples and workers the granularity remains close to ten percent. This means that the simple scheme for sharing work is sufficient.

Manager Load A possible drawback of a single manager is the potential of a performance bottleneck. If the single manager is not able to keep up with processing `find`-messages, workers might be idle even though other workers have work to share. Figure 9 shows the load of the manager, where a load of 50% means that the manager is idle during half of the entire runtime.

For all examples the manager has a load of less than 50%. For the more realistic examples Photo and MT 10 the load is less than 20%. This provides evidence that the manager will be able to efficiently serve messages for more than six workers. There are two reasons why the load is quite low. Firstly, work granularity is coarse as argued above. Coarse granularity means that workers infrequently communicate with the manager to find work. Secondly, each incoming request to find work is handled by a new thread. Hence, the manager is immediately ready to serve further incoming messages.

Recomputation Overhead Figure 10 shows the recomputation overhead. The numbers suggest that the overhead for recomputation is always less than 10%.

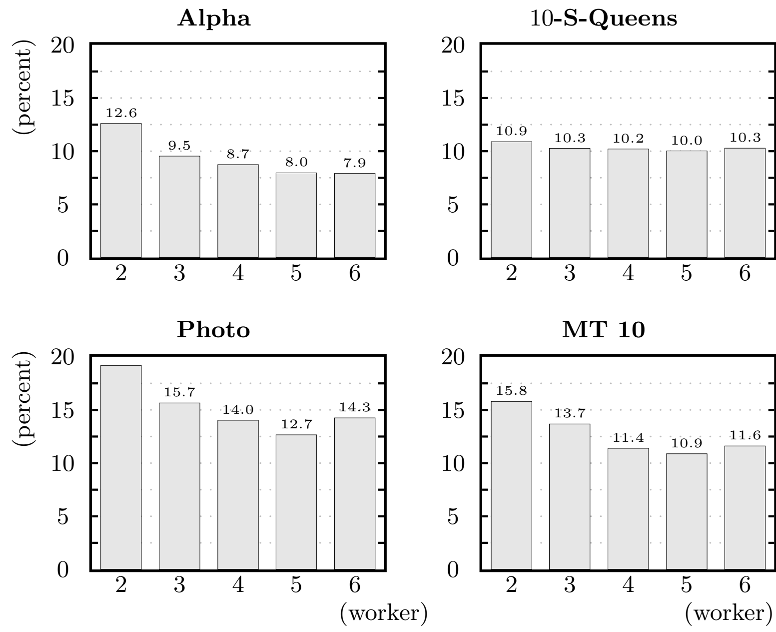

Fig. 8. Work granularity.

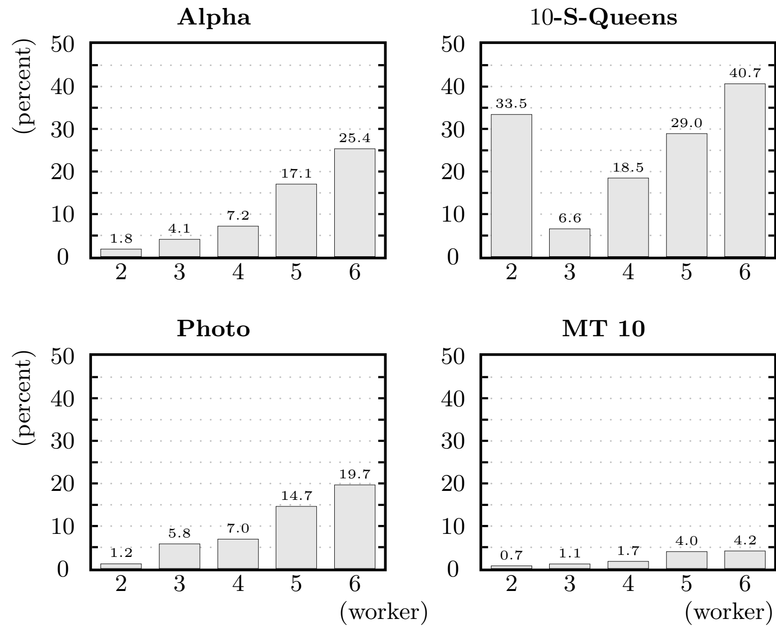

Fig. 9. Manager load.

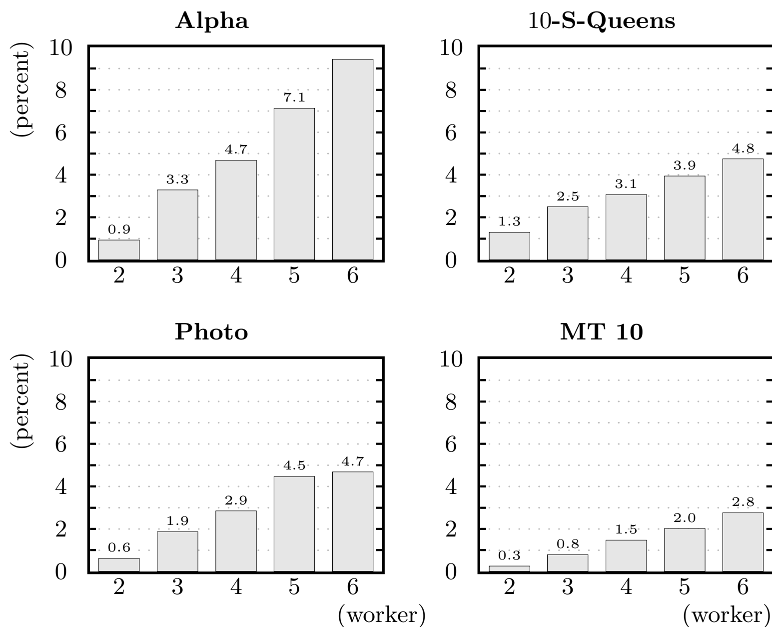

Fig. 10. Recomputation overhead.

This means that the price paid for distributing work across the network is low. For the general performance of recomputation in Oz, see [17].

Exploration Overhead Exploration overhead occurs for branch-and-bound search and is due to the differing order in which solutions are found (see Section 3.4). Figure 11 shows the exploration overhead for Photo and MT 10. The exploration overhead is almost exclusively the cause for the speedup loss.

Exploration overhead is a consequence of performing branch-and-bound in parallel and is independent of our implementation of the search engines. A different approach to parallel best-solution search is presented by Prestwich and Mudambi in [13]. They use a technique called *cost-parallelism*, where several searches for a solution with different cost bounds are performed in parallel. This technique is shown to perform better than parallel branch-and-bound search.

A Hardware and Software Platforms

The performance figures use a collection of standard personal computers running RedHat Linux 6.2 (Kernel 2.2.14) connected by a 100 MBit Ethernet. Three different types of machines are used:

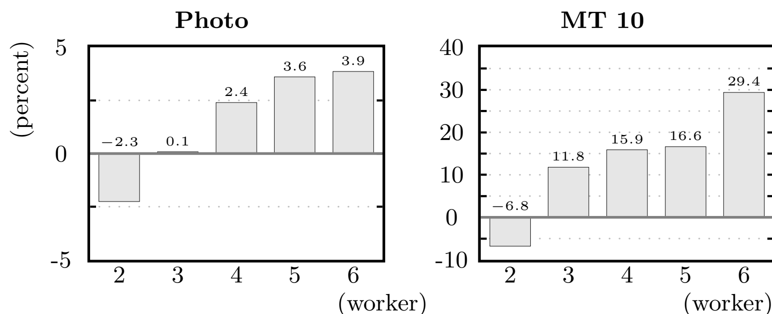

Fig. 11. Exploration overhead.

Type	Processors	Memory
a	2×400 MHz Pentium II	256 MB
b	2×400 MHz Pentium II	512 MB
c	1×466 MHz Celeron	256 MB

The combination of computers for varying number of workers is as follows, where the manager has always been run on computer a: 1:(a), 2:(a,b), 3:(a,b,c), 4:(a,b,2×c), 5:(a,b,3×c), and 6:(a,b,4×c).

All times have been taken as wall time (that is, absolute clock time), where all machines were unloaded. As system Mozart 1.1.1 has been used. All numbers presented are the arithmetic mean of 25 runs.

Acknowledgements I am grateful to Thorsten Brunklaus, Tobias Müller, and the referees for helpful comments on the paper.

References

1. Jacques Chassin de Kergommeaux and Philippe Codognet. Parallel logic programming systems. *ACM Computing Surveys*, 26(3):295–336, September 1994.
2. W. F. Clocksin. Principles of the DelPhi parallel inference machine. *The Computer Journal*, 30(5):386–392, 1987.
3. W. F. Clocksin and H. Alshawi. A method for efficiently executing horn clause programs using multiple processors. *New Generation Computing*, 5:361–376, 1988.
4. Denys Duchier, Leif Kornstaedt, Christian Schulte, and Gert Smolka. A higher-order module discipline with separate compilation, dynamic linking, and pickling. Technical report, Programming Systems Lab, Universität des Saarlandes, `www.ps.uni-sb.de/papers`, 1998. Draft.
5. Seif Haridi, Peter Van Roy, Per Brand, Michael Mehl, Ralf Scheidhauer, and Gert Smolka. Efficient logic variables for distributed computing. *ACM Transactions on Programming Languages and Systems*, 21(3):569–626, May 1999.
6. Seif Haridi, Peter Van Roy, Per Brand, and Christian Schulte. Programming languages for distributed applications. *New Generation Computing*, 16(3):223–261, 1998.

7. Sverker Janson, Johan Montelius, and Seif Haridi. Ports for objects. In *Research Directions in Concurrent Object-Oriented Programming*. The MIT Press, Cambridge, MA, USA, 1993.
8. Vipin Kumar and V. Nageshwara Rao. Parallel depth first search. Part II. Analysis. *International Journal of Parallel Programming*, 16(6):501–519, 1987.
9. Tobias Müller and Jörg Würtz. Embedding propagators in a concurrent constraint language. *The Journal of Functional and Logic Programming*, 1999(1):Article 8, April 1999. Special Issue. Available at: `mitpress.mit.edu/JFLP/`.
10. Mozart Consortium. The Mozart programming system, 1999. Available from `www.mozart-oz.org`.
11. Shyam Mudambi and Joachim Schimpf. Parallel CLP on heterogeneous networks. In Pascal Van Hentenryck, editor, *Proceedings of the Eleventh International Conference on Logic Programming*, pages 124–141. The MIT Press, Santa Margherita Ligure, Italy, 1994.
12. Laurent Perron. Search procedures and parallelism in constraint programming. In Joxan Jaffar, editor, *Proceedings of the Fifth International Conference on Principles and Practice of Constraint Programming*, volume 1713 of *Lecture Notes in Computer Science*, pages 346–360, Alexandra, VA, USA, October 1999. Springer-Verlag.
13. Steven Prestwich and Shyam Mudambi. Improved branch and bound in constraint logic programming. In Ugo Montanari and Francesca Rossi, editors, *Proceedings of the First International Conference on Principles and Practice of Constraint Programming*, volume 976 of *Lecture Notes in Computer Science*, pages 533–548, Cassis, France, September 1995. Springer-Verlag.
14. V. Nageshwara Rao and Vipin Kumar. Parallel depth first search. Part I. Implementation. *International Journal of Parallel Programming*, 16(6):479–499, 1987.
15. Christian Schulte. Oz Explorer: A visual constraint programming tool. In Lee Naish, editor, *Proceedings of the Fourteenth International Conference on Logic Programming*, pages 286–300, Leuven, Belgium, July 1997. The MIT Press.
16. Christian Schulte. Programming constraint inference engines. In Gert Smolka, editor, *Proceedings of the Third International Conference on Principles and Practice of Constraint Programming*, volume 1330 of *Lecture Notes in Computer Science*, pages 519–533, Schloß Hagenberg, Linz, Austria, October 1997. Springer-Verlag.
17. Christian Schulte. Comparing trailing and copying for constraint programming. In Danny De Schreye, editor, *Proceedings of the 1999 International Conference on Logic Programming*, pages 275–289, Las Cruces, NM, USA, November 1999. The MIT Press.
18. Christian Schulte. Programming deep concurrent constraint combinators. In Enrico Pontelli and Vítor Santos Costa, editors, *Practical Aspects of Declarative Languages, Second International Workshop, PADL 2000*, volume 1753 of *Lecture Notes in Computer Science*, pages 215–229, Boston, MA, USA, January 2000. Springer-Verlag.
19. Gert Smolka. The Oz programming model. In Jan van Leeuwen, editor, *Computer Science Today*, volume 1000 of *Lecture Notes in Computer Science*, pages 324–343. Springer-Verlag, Berlin, 1995.
20. Peter Van Roy, Seif Haridi, and Per Brand. *Distributed Programming in Mozart - A Tutorial Introduction*. The Mozart Consortium, `www.mozart-oz.org`, 1999.
21. Peter Van Roy, Seif Haridi, Per Brand, Gert Smolka, Michael Mehl, and Ralf Scheidhauer. Mobile objects in Distributed Oz. *ACM Transactions on Programming Languages and Systems*, 19(5):804–851, September 1997.