Using the ProSet-Linda Prototyping Language for Investigating MIMD Algorithms for Model Matching in 3-D Computer Vision*

W. Hasselbring[1] and R. B. Fisher[2]

[1] Dept. of Computer Science, University of Dortmund
Informatik 10 (Software Technology), D-44221 Dortmund, Germany
Telephone: 49-(231)-755-4712, Fax: 49-(231)-755-2061
email: willi@ls10.informatik.uni-dortmund.de

[2] Dept. of Artificial Intelligence, University of Edinburgh
5 Forrest Hill, Edinburgh EH1 2QL, Scotland, United Kingdom
Telephone: 44-(31)-650-3098, Fax: 44-(31)-650-6899
email: rbf@aifh.ed.ac.uk

Abstract. This paper discusses the development of algorithms for parallel interpretation-tree model matching for 3-D computer vision applications such as object recognition. The algorithms are developed with a prototyping approach using ProSet-Linda. ProSet is a procedural prototyping language based on the theory of finite sets. The coordination language Linda provides a distributed shared memory model, called tuple space, together with some atomic operations on this shared data space. The combination of both languages, viz. ProSet-Linda, is designed for prototyping parallel algorithms.
The classical control algorithm for symbolic data/model matching in computer vision is the *Interpretation Tree* search algorithm. Parallel execution can increase the execution performance of model matching, but also make feasible entirely new ways of solving matching problems. In the present paper, we emphasize the *development* of several parallel algorithms with a prototyping approach. The expected improvements attained by the parallel algorithmic variations for interpretation-tree search are analyzed.

Keywords: model-based vision, object recognition, parallel search, prototyping parallel algorithms.

1 Introduction

Three-dimensional computer vision is commonly divided into several levels. In the research investigated at Edinburgh, low-level vision is concerned with processing range data acquired by a laser range scanner to eliminate noise [10]. Medium-level vision is concerned with identifying geometric surfaces [22]. High-level vision tries, for example, to identify the shape and position of data objects using matched given model features. In the high-level component, first the model

* In: Proc. IRREGULAR '95, Lecture Notes in Computer Science, Springer-Verlag, September 1995, Lyon, France.

invocation process pairs likely model and data features for further consideration [7]. Model matching then uses the candidate matches proposed by the invocation to form consistent groups of matches.

The classical control algorithm for symbolic model matching in computer vision is the *Interpretation Tree* search algorithm [13]. The algorithm searches a tree of potential model-to-data correspondences, such that each node in the tree represents one correspondence and the path of nodes from the current node back to the root of the tree is a set of simultaneous pairings. This model matching algorithm is a specialized form of the general AI tree search technique, where branches are pruned according to a set of consistency constraints according to some (geometric) criterion. The goal of the search algorithm is to maximize the set of consistent model-to-data correspondences in an efficient manner. Finding these correspondences is a key problem in model-based vision, and is usually a preliminary step to object recognition, pose estimation, or visual inspection.

Unfortunately, this algorithm has the potential for combinatorial explosion. To reduce the complexity, techniques for pruning the trees have been developed, thus limiting the number of candidate matches considered [13]. However, even with these effective forms of pruning, the algorithms still can have exponential complexity, making the standard interpretation-tree search algorithms unsuitable for use in scenes with many features.

Parallel execution can increase the execution performance of model matching, but also allow entirely new ways of solving matching problems. As has been observed, it is only from new algorithms that orders of magnitude improvements in the complexity of a problem can be achieved [3]. Thus, rapid prototyping of parallel algorithms may serve as the basis for developing parallel, high-performance applications. **In this paper, we present a methodology for the *development* of parallel high-level vision algorithms using a ProSet-Linda based prototyping approach.**

Parallelism in low- and medium-level computer vision is usually programmed in a *data-parallel* way, for instance based on the computational model of cellular automata [14]. For high-level symbolic computer vision, the data-parallel approach is not appropriate, as symbolic computations have an irregular control flow depending on the actual input data.

Data parallelism is opposed to *control parallelism*, which is achieved through multiple threads of control, operating independently. The data-parallel approach lets programmers replace iteration (repeated execution of the same set of instructions with different data) with parallel execution. It does not address a more general case, however: performing many interrelated but *different* operations at the same time. This ability is essential for developing algorithms for high-level symbolic computer vision. The data-parallel programming model is based on the *single-program/multiple-data* (SPMD) model as opposed to the *multiple-instruction/multiple-data* (MIMD) model.

Developing parallel algorithms is in general considered an awkward undertaking. The goal of the ProSet-Linda approach is to partially overcome this problem by providing a tool for prototyping parallel algorithms [15]. To support prototyping parallel algorithms, a prototyping language should provide simple and powerful facilities for dynamic creation and coordination of parallel processes.

Section 2 gives a brief introduction to the tool for implementing the parallel variations of the interpretation-tree search algorithm, viz. ProSet-Linda for prototyping parallel algorithms. Section 3 takes a general look at parallel interpretation-tree search. We do not parallelize the standard interpretation-tree algorithm, but the non-wildcard and best-first alternatives in Sects. 4 and 5, respectively. Section 6 evaluates the investigated algorithms. Section 7 discusses the transformation of prototypes into efficient implementations and Sect. 8 draws some conclusions.

2 Prototyping Parallel Algorithms with ProSet-Linda

Before presenting the implementation of the parallel interpretation-tree model matching algorithms, we have a look at ProSet-Linda as the language used for implementation. The procedural, set-oriented language ProSet [5] is a successor to SETL [20]. ProSet is an acronym for PROtotyping with SETs. The high-level structures that ProSet provides qualify the language for prototyping. Refer to [4] for a full account of prototyping with set-oriented languages and to [6, 18] for case studies for prototyping using SETL.

2.1 Basic Concepts

ProSet provides the data types atom, integer, real, string, Boolean, tuple, set, function, and module. ProSet is weakly typed, i.e., the type of an object is in general not known at compile time. Atoms are unique with respect to one machine and across machines. They can only be created and compared for equality. Tuples and sets are compound data structures, the components of which may have different types. Sets are unordered collections while tuples are ordered. There is also the undefined value `om` which indicates undefined situations.

As an example consider the expression `[123, "abc", true, {1.4, 1.5}]` which creates a tuple consisting of an integer, a string, a Boolean, and a set of two reals. This is an example of what is called a *tuple former*. As another example consider the *set forming* expression `{2*x: x in [1..10] | x>5}` which yields the set `{12, 14, 16, 18, 20}`. The quantifiers of predicate calculus are provided (∃, ∀). The control structures have ALGOL as one of its ancestors.

2.2 Parallel Programming

Process communication and synchronization in ProSet-Linda is reduced to concurrent access to a shared data pool, thus relieving the programmer from the burden of having to consider all process inter-relations explicitly. The parallel processes are decoupled in time and space in a simple way: processes do not have to execute at the same time and do not need to know each other's addresses (as it is necessary with message-passing systems). Linda is a coordination language which extends a sequential language by means for synchronization and communication through so-called *tuple spaces* [12]. Synchronization and communication in ProSet-Linda are carried out through several atomic operations: addition, removal, reading, and updates of individual tuples in tuple space. Linda and ProSet both provide tuples; thus, it is quite natural to combine both models to form a tool for prototyping parallel algorithms. The access unit in tuple space is the tuple. Reading access to tuples in tuple space is associative and not based on physical addresses, but rather on their expected content described in *templates*. This method is similar to the selection of entries from a data base. ProSet supports multiple tuple spaces. Several ProSet-Linda library functions are provided for handling multiple tuple spaces dynamically.

ProSet provides three tuple-space operations. The `deposit` operation deposits a tuple into a tuple space:

```
deposit [ "pi", 3.14 ] at TS end deposit;
```

`TS` is the tuple space at which the tuple `[ "pi", 3.14 ]` has to be deposited. The `fetch` operation tries to fetch and remove a tuple from a tuple space

```
fetch ( "name", ? x |(type $(2) = integer) ) at TS end fetch;
```

This template only matches tuples with the string `"name"` in the first field and integer values in the second field. The symbol `$` may be used like an expression as a placeholder for the values of corresponding tuples in tuple space. The expression `$(i)` then selects the `i`th element from these tuples. Indexing starts with `1`. As usual in ProSet, | means *such that*. The optional *l*-values specified in the formals (the variable `x` in our example) are assigned the values of the corresponding tuple fields, provided matching succeeds. Formals are prefixed by question marks. The selected tuple is removed from tuple space. The `meet` operation is the same as `fetch`, but the tuple is not removed and may be changed:

```
meet ( "pi", ? x ) at TS end meet;
```

Changing tuples is done by specifying expressions `into` which specific tuple fields will be changed. Consider

```
meet ( "pi", ? into 2.0*3.14 ) at TS end meet;
```

where the second element of the met tuple is changed into the value of the expression `2.0*3.14`. Tuples which are met in tuple space may be regarded as shared data since they remain in tuple space irrespective of changing them or not.

3 Parallel Interpretation-Tree Search

Parallelism in a tree search algorithm can be obtained by searching the branches of a tree in parallel. A simple approach would be to spawn a new process for each subtree to be evaluated. This approach would not work well since the amount of parallelism is determined by the input data and not by, for instance, the number of available processors.

The programs which will be presented in the following sections are master-worker applications (also called *task farming*). In a master-worker application, the task to be solved is partitioned into independent subtasks. These subtasks are placed into a tuple space, and each process in a pool of identical workers then repeatedly retrieves a subtask description from the tuple space, solves it, and puts the solutions into a tuple space. The master process then collects the results. An advantage of this programming approach is easy load balancing because the number of workers is variable and may be set to the number of available processors.

Similar to sequential tree search, it is in general not necessary to search the entire tree: *bounding rules* avoid searching the entire tree.

4 Parallel Non-wildcard Search Tree Algorithms

As many of the nodes in the standard interpretation tree algorithm arise because of the use of *wildcards*, an alternative search algorithm explores the same search space, but it does not use a wildcard model feature to match otherwise unmatchable data features [8]. The tree in Fig. 1 displays an example non-wildcard interpretation tree. In a sequential algorithm, the tree is searched depth-first following the leftmost branches first (no pruning is shown here to illustrate the shape of the tree). The tuple Ω is the output of model invocation.

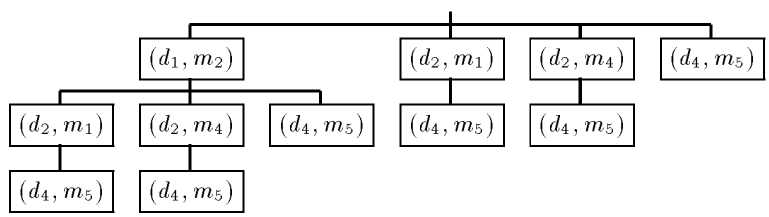

Fig. 1.
An example interpretation tree for $\Omega = [(d_1, m_2), (d_2, m_1), (d_2, m_4), (d_4, m_5)]$. The d_i are data features and the m_i are model features. The root of the interpretation tree has no pairings. Each data feature appears (in order) at most once in a branch. At each node at level k in the tree, therefore, there is a hypothesis with k features matched.

Section 4.1 presents a parallel non-wildcard complete search algorithm which finds all satisfactory matches. A match is satisfactory when the termination number of matched features has been reached.

The sequential non-wildcard search tree algorithm stops when the first satisfactory match has been found [8]. It does not search for *all* solutions. Section 4.2 presents a parallel non-wildcard search tree algorithm which stops when the first satisfactory match has been found.

4.1 Parallel Complete Search Tree Algorithm

This section discusses a parallel master-worker implementation of the non-wildcard search tree algorithm which provides all satisfactory matches.

Model invocation uses model and data properties to pair likely model and data features for further consideration [7]. It produces a sorted list of consistent model-to-data pairs $(model_i, data_i, A_i)$ where A_i is the compatibility measure (plausibility) of the features $model_i$ and $data_i$. The pair list is initially sorted with larger A_i values at the top. The hypotheses from the model invocation are stored as a tuple in the variable `Hypotheses`.

Figure 2 displays the coarse structure of the master-worker program. Arrows indicate access to the tuple spaces. These access patterns are only shown for one of the identical worker processes.

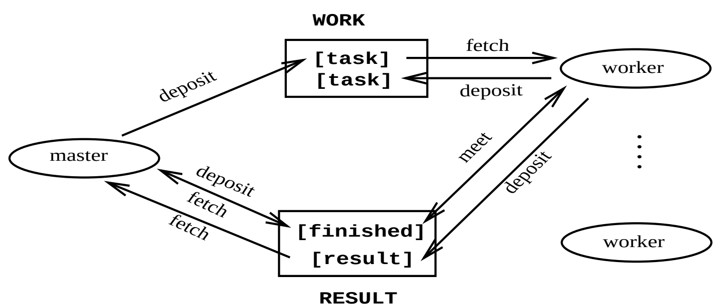

Fig. 2. The coarse structure of the master-worker program.

In this paper, only small parts of the code can be presented. Refer to [16] for the complete description. We use two tuple spaces. One for the work tasks (`WORK`) and one for the results (`RESULT`). The number of worker processes `NumWorker` is an argument to the main program. This could be, for instance, the number of available processors. The termination threshold for satisfactory matches is the next argument to the main program. The master (the main program) spawns `NumWorker` worker processes to do the work and puts the initial task tuples into tuple space `WORK`. These initial tasks represent the nodes at the first level of the interpretation-tree. Then, the master initializes a shared counter for the number of finished workers at tuple space `RESULT` and waits until all workers have done their work (by executing a blocking fetch until the number of finished worker processes equals `NumWorker`). Then the master fetches the possible matches from `RESULT`.

This was a sketch of the implementation of the master process (the main program). Now let us look at the worker procedure. Each worker executes in a loop in which it first checks whether there are more task tuples in tuple space `WORK`, and terminates when there is no more work to do. Before termination, the shared counter for the number of finished workers in `RESULT` is incremented to indicate the termination to the master.

Each extension of a branch in the interpretation-tree is formed by appending new entries from Ω, subject to the constraints that (1) each data feature appears at most once on a path through the tree and (2) the data features are used in order (with gaps allowed). The condition in the following **for** loop of the worker ensures that these constraints are satisfied:

```
for Entry in Hypotheses | (forall x in MyPath | (Entry(1) > x(1))) do
  if Consistent (MyPath, Entry) then
    deposit [MyPath + {Entry}]  at TargetTS end deposit;
  end if;
end for;
```

The set of pairs `MyPath` represents the current partial branch in the tree. The condition `Entry(1) > x(1)` enforces the data feature ordering constraint. Only extensions that satisfy the normal binary constraints are accepted (`Consistent` checks this). Extension stops when the termination threshold of matches is reached. Beforehand, `TargetTS` has been set to indicate whether we have a new incomplete work task (`TargetTS` is `WORK`) or a new satisfactory result (`TargetTS` is `RESULT`).

4.2 Parallel First-Stop Search Tree Algorithm

This section briefly describes a parallel non-wildcard search tree algorithm which stops the program when the first satisfactory match has been found. This algorithm is quite similar to the sequential non-wildcard search tree algorithm, but the tree is searched in a non-deterministic order and *not* depth-first following the leftmost branches first.

Synchronization between the master and the workers is achieved when the first satisfactory match has been found. Provided that there exist at least one consistent match, the master need not wait until all tasks are evaluated as is the case with the parallel non-wildcard complete search tree algorithm of Sect. 4.1. The master waits for the first satisfactory match to be deposited by a worker at `RESULT`. For a detailed discussion of this program refer to [16].

5 Parallel Best Search Tree Algorithms

The best-first search tree algorithm [8] assumes that it is possible to evaluate how well sets of model features match sets of data features (based on the plausibilities from the invocation and consistency measures as the set sizes grow). As any real problem is likely to provide some useful heuristic ordering constraints, the potential for speeding up the matching process is large.

In contrast to the non-wildcard algorithms (see Sect. 4), with the best-first algorithms we are interested in both the cost of a path to a solution (i.e. we wish to minimize the time to finding a solution) as well as the quality of the solution. Both algorithms use the same tree structure (see Fig. 1), but the portion explored may be different.

Section 5.1 presents a parallel search tree algorithm which provides the optimal match where each data feature is mapped to a model feature when considering plausibilities for the data/model feature pairs. The sequential best-first search tree algorithm searches for the first plausible solution (usually not the optimal solution). Section 5.2 presents a parallel best-first search tree algorithm.

5.1 Parallel Optimum Search Tree Algorithm

One parallel search tree algorithm finds the *optimal* match where a satisfactory number of data features is mapped to model features when considering plausibilities for the data/model feature pairs.

In addition to putting the initial task tuples into `WORK`, and initializing a shared counter for the number of finished workers at `WORK`, the master initializes an empty result set with plausibility 0.0 at tuple space `RESULT` (the current optimum). After spawning the workers, the master waits until all workers have done their work, and then fetches the optimal match from `RESULT`.

The algorithm assumes that the plausibility evaluation is monotonically decreasing as the path length increases. First the worker computes the plausibility of its own path of matches and reads the plausibility of an already known satisfactory match from `RESULT`, and compares it with the plausibility of its own (not yet satisfactory) match. If its own plausibility is lower than the plausibility of an already known satisfactory match, the worker continues to fetch another task tuple (according to the bounding rule). Otherwise, the worker checks whether the length of its partial match is already a satisfactory match but one. If so, it changes the optimal match in `RESULT` to its own evaluated match (extended to a satisfactory match). This algorithm is essentially a *branch-and-bound* algorithm [19]. For a detailed discussion of the task evaluation refer to [16].

5.2 Parallel Best-First Search Tree Algorithm

An alternative parallel best-first search tree algorithm terminates at the first satisfactory match. The central data structure is a distributed priority queue of entries of the following form, sorted by the estimated evaluation of the next potential extension:

$$(S_i = \{pair_{i_1}, pair_{i_2}, \ldots pair_{i_n}\}, g(S_i), m, f(S_i \cup \{pair_m\}))$$

where S_i is a set of n mutually compatible model-to-data pairs (a partial branch in the tree), $g(S_i)$ is the *actual* evaluation of S_i, m indicates that $pair_m$ is the next extension of S_i to be considered, and $f(S_i \cup \{pair_m\})$ is the *estimated* evaluation of that extension. The priority queue is sorted with larger $f()$ values at the top.

In addition to putting the initial task tuples into tuple space `WORK`, and initializing a shared counter for the number of finished workers, the master initializes the top of the priority queue at tuple space `WORK` with components $(\{\}, 1.0, 1, A_1)$:

```
deposit [ 1, 0, {}, 1.0, 1, Hypotheses(1)(3) ] at WORK end deposit;
```

Each entry of the priority queue is stored as a tuple in `WORK`. The first component indicates the *pointer* to the corresponding entry. The integer `1` indicates the top of the queue. The second component *points* to the next entry. The integer `0` indicates the end of the queue. Figure 3 illustrates the structure of this queue.

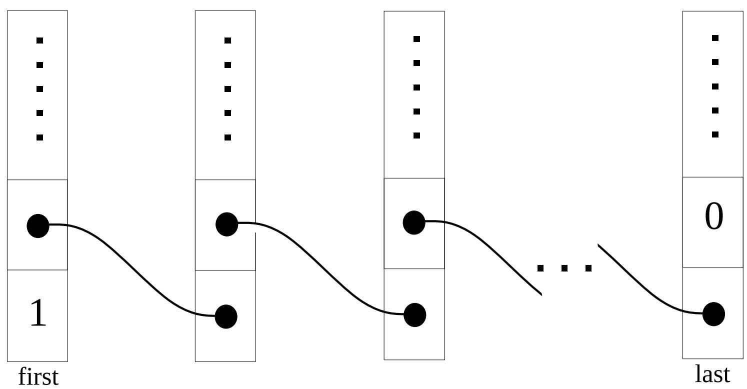

Fig. 3. The distributed priority queue for parallel best-first search.
The integer values `1` and `0` are used to indicate the top and the end of the queue, respectively. The intermediate entries are *identified* by the contained atoms. We use black circles to represent the atoms. A link between two atoms means that these two atoms are equal. Note, that the atoms are not the *addresses* of the respective entries, but rather the *identification* of the entries (distributed pointers which are independent of memory addresses to allow access from different processors).

The expression `Hypotheses(1)(3)` selects the plausibility for the highest rated hypothesis from the model invocation (this is A_1). The hypotheses are initially sorted by the model invocation.

Again, each worker executes in a loop and first pops the top of the priority queue $(S_i, g(S_i), m, f(S_i \cup \{pair_m\}))$ at tuple space `WORK`:

```
fetch ( 1, ? second, ? S_i, ? g, ? m, ? f ) at WORK end fetch;
if second /= 0 then
  -- The second entry becomes the first one through a changing meet:
  meet ( ? into 1, ?, ?, ?, ?, ? | $(1)=second ) at WORK end meet;
end if;
```

After popping the top of the priority queue, other worker processes can work in parallel on the tail of the queue to allow parallel access to the distributed queue, provided that there exists a tail.

If not rejected by consistency checks, early termination or non-existence of further hypotheses, the worker generates the next descendant of the successful extension:

$$(S_i \cup \{pair_m\}, g(S_i \cup \{pair_m\}), m+1, f(S_i \cup \{pair_m\} \cup \{pair_{m+1}\}))$$

plus the next descendant of the parent:

$$(S_i, g(S_i), m+1, f(S_i \cup \{pair_m\}))$$

to be inserted into priority queue.

Insertion of the next descendant of the successful extension is done by the `Insert` function, which enters the new node into the appropriate priority position, provided that the priority queue contained more than one entry:

```
if second /= 0 then
  Insert (1, S_i + {Hypotheses(m)}, g, m+1, f);
else
  next := newat(); -- A new 'distributed pointer'
  deposit [1, next, S_i + {Hypotheses(m)}, g, m+1, f] at WORK end deposit;
end if;
```

If the priority queue contained only the popped entry (the variable `second` indicates this), we directly deposit the new entry as the top of the priority queue. The next entry then obtains the atom `next` as its identity. ProSet's built-in function `newat` returns a new atom. The next descendant of the parent is inserted accordingly.

The algorithm needs two evaluation functions, $f()$ for the estimated new state evaluation and $g()$ for the actual state evaluation. The $f()$ evaluation function gives longer branches higher evaluations to direct the workers to search the tree depth-first. As with the first-stop algorithm (see Sect. 4.2), the master waits for the first satisfactory match to be deposited by a worker at `RESULT`.

The priority queue is stored as a *distributed data structure* [17] in tuple space `WORK`. Distributed data structures may be examined and manipulated by multiple processes simultaneously. In this case, multiple processes can work independently on different partitions of the queue. The individual entries are *linked* together by means of ProSet's atoms. ProSet does not support *pointers* as they are known in Modula, C or similar procedural languages. As mentioned in Section 2.1, atoms are unique with respect to one machine and across machines (they contain the host and process identification, creation time, and an integer counter). Atoms can only be created and compared for equality. We use them as *distributed pointers* which are independent of the processor's memory addresses. Note, that multiple processes can work independently on different partitions of the queue. The insertion procedure of new entries into the distributed priority queue `Insert` is presented in [16]. A variety of other data structures, such as distributed priority sorted heaps or distributed sorted trees, could be used to implement the priority queue.

6 Evaluation

The parallel complete search tree algorithm. The parallel non-wildcard complete search tree algorithm provides all satisfactory matches. If we neglect pruning of inconsistent branches, the number of evaluated nodes is proportional to H^T where H is the number of hypotheses from the model invocation and T is the termination threshold. The time to evaluate these nodes with the sequential algorithm is proportional to H^T, whereas the time to evaluate these nodes with the parallel algorithm is proportional to $\frac{H^T}{W}$ where W is the number of worker processes, since the worker processes evaluate the branches of the tree in parallel. However, the actual amount of parallelism may be restricted by the branching factor of the tree and contention caused by competing access to the tuple spaces. In principle, the situation for the above calculation does not change when considering pruning of inconsistent branches

With the non-wildcard algorithm, the second and third levels of the search tree represent matches that use several non-wildcard pairings. The binary constraints eliminate almost all false pairings quickly [8]. The trade-off is that the branching factor of the non-wildcard tree is H instead of the number of data features as with the standard interpretation-tree algorithm, but the depth of the tree for any false sets of matches is usually very shallow. Therefore, the parallel non-wildcard complete search algorithm allows a high amount of parallelism because of the large branching factor of the tree.

The parallel first-stop search algorithm. The parallel first-stop search algorithm is quite similar to the sequential non-wildcard search tree algorithm. The tree is not searched depth-first following the leftmost branches first, but in parallel in a non-deterministic order.

For the sequential algorithm, the time to find the first match is highly data dependent. If, for instance, the left-most branch represents a satisfactory match, the sequential algorithm will probably be faster than the parallel algorithm, because the parallel algorithm will probably not follow the left-most branch first. The parallel algorithm *may* find a satisfactory match earlier, but this is not definite since the evaluation order is non-deterministic. However, the mean time to finding a solution is improved in proportion to the number of workers. Since the non-wildcard algorithms do not consider any valuation for the data/model feature pairs, nothing can *guide* the workers to follow the most promising branches.

This raises the question whether it pays to parallelize the tree search when we are only interested in obtaining *any* satisfactory match. If there are many possible solutions, then this parallel algorithm is unlikely to make dramatic improvements; however, if there are few solutions, then the speedup should be nearly linear in the number of workers. The situation changes to some extent when we are interested in obtaining a *good* satisfactory match.

The parallel optimum search algorithm. This algorithm searches in parallel the same tree as a similar sequential branch-and-bound algorithm would search. The bounding rules apply to parallel search in the same way as they apply to sequential search (branch-and-bound). Therefore, the parallel optimum search algorithm can be compared to a similar sequential branch-and-bound algorithm in much the same way as the parallel non-wildcard complete search tree algorithm of Sect. 4.1 can be compared to a sequential non-wildcard algorithm which provides all satisfactory matches (see above).

The parallel best-first algorithm. For testing this algorithm, the output from the low- and medium-level components of the IMAGINE2 system [9] for the range image of a workpiece is used. Some experimental results for the parallel best-first search algorithm are displayed in Fig. 4. Figure 4a shows the number of visited nodes in relation to the number of workers and Fig. 4b shows the number of visited nodes per worker in relation to the number of workers. T is the termination threshold for satisfactory matches. The zigzag line is due to non-determinism, but the tendency is obvious. The number of visited nodes per worker converges to approximately $\frac{T}{2}$ as the number of workers increases. Therefore, the addition of worker processes increases the search space.

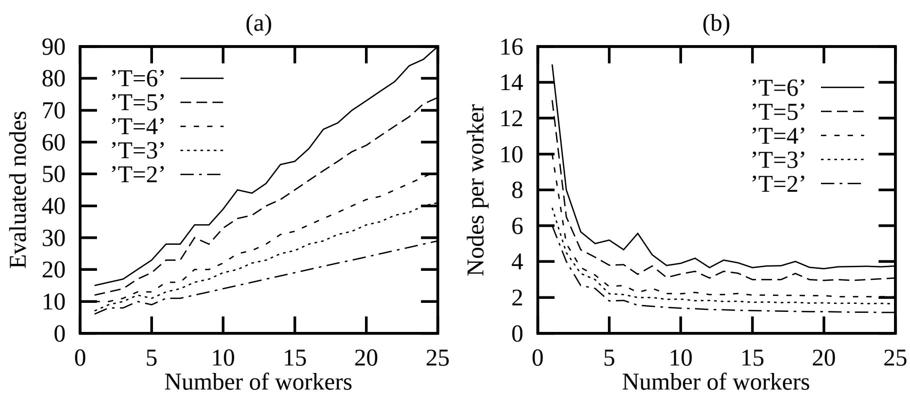

Fig. 4. The experimental results for the parallel best-first search algorithm.

The parallel best-first algorithm appears to be a good compromise between the parallel optimum search algorithm and the sequential best-first algorithm. It is not necessarily much faster than the sequential best-first algorithm, but can produce better results within the same or even a shorter time. The $f()$ function for the estimated new state evaluations directs the workers to search the tree depth-first, which increases the probability of finding a satisfactory match earlier. The workers are *guided* by the plausibilities to follow the most promising branches.

7 Transforming the Prototype Implementation into an Efficient Implementation

As a consequence of our evaluation, in a successor project the prototype of the parallel best-first algorithm will be transformed into an efficient implementation. In a first step, the ProSet-Linda prototype is transformed into a C-Linda implementation [21]. With Linda, it is easy to program with different styles, e.g. with distributed data structures, active data structures and message passing [2]. The C-Linda implementation will be programmed in a message-passing style to serve as a preliminary step for a message-passing implementation with an MPI library (Message Passing Interface [11]). We then use the CHIMP/MPI (Common High-Level Interface to Message Passing) [1] implementation at the Edinburgh Parallel Computing Centre, which is an efficient implementation of the MPI standard. CHIMP/MPI supports Sun, Silicon Graphics, IBM, and DEC workstations as well as Sequent Symmetry and Meiko Transputer/i860/SPARC Computing Surfaces. This CHIMP/MPI implementation will be used to compare performance of the algorithm on different parallel computing systems.

Note that this transformation is done by hand and not supported by a compiler or other tools. We do not believe that such transformations can be done fully automatically, but some kind of tool support is conceivable. Before building transformation tools we would like to gain some insights into the requirements on such tools through practical experience. Another project is underway to implement the Cowichan Problems [23] — a set of problems for assessing the usability of parallel programming systems — with the same development process to obtain a greater range of practical experiences with our approach.

8 Conclusions

We discussed the development of algorithms for parallel interpretation-tree model matching for 3-D computer vision applications with ProSet-Linda. Prototypes for four parallel algorithms have been developed and evaluated.

The evaluation showed that not all algorithmic variations are good candidates for parallelization. An application area for prototyping is to carry out *feasibility studies*. If we had implemented the algorithms directly with a production language, for example C with extensions for message passing, the implementation effort would have been higher. However, the exact savings in time cannot be presented: This would require a similar project without prototyping for comparison.

The other main observation to make at this point is: because the sequential variations of interpretation-tree model matching algorithm were presented in a *set-oriented* way [8], it was quite straightforward to implement them and the alternative parallel implementations in ProSet and then compare them, in only a few weeks. The four presented programs are complete executable prototypes for the developed algorithms. They could be regarded as *executable specifications*.

One of the goals of the successor project will be to compare performance of the algorithm with that predicted by the prototype. This is what prototyping is about: experimenting with ideas for algorithms and evaluating them to make the right decisions for the next steps in the development. Purely theoretic evaluations are often not possible in practice. The main contribution of this paper are the presented techniques for parallelization of interpretation-tree model matching and the evaluation of these techniques. It is also a case study for prototyping of parallel algorithms.

Acknowledgments

This work has been supported by the TRACS program funded by the Human Capital and Mobility program of the European Commission (contract number ERB-CHGE-CT92-0005), and the Universities of Dortmund and Edinburgh.

The authors would like to thank Andrew Fitzgibbon for the help with the IMAGINE2 system, Philippe Fillatreau and Josef Hebenstreit for the discussions on object recognition, Peter Maccallum and Neil MacDonald for support with the Edinburgh Parallel Computing Centre facilities, and Henri Bal and Peter Maccallum for the comments on drafts of this paper.

References

1. R. Alasdair, A. Bruce, J.G. Mills, and A.G. Smith. CHIMP/MPI User Guide. Technical Report EPCC-KTP-CHIMP-V2-USER 1.2, Edinburgh Parallel Computing Centre, Edinburgh, UK, June 1994.
2. N. Carriero and D. Gelernter. *How to write parallel programs.* MIT Press, 1990.
3. J. Cocke. The search for performance in scientific processors. *Communications of the ACM*, 31(3):249–253, 1988.
4. E.-E. Doberkat and D. Fox. *Software Prototyping mit SETL.* Leitfäden und Monographien der Informatik. Teubner-Verlag, 1989.
5. E.-E. Doberkat, W. Franke, U. Gutenbeil, W. Hasselbring, U. Lammers, and C. Pahl. ProSet — A Language for Prototyping with Sets. In N. Kanopoulos, editor, *Proc. Third International Workshop on Rapid System Prototyping*, pages 235–248, Research Triangle Park, NC, June 1992.
6. K.A. Faigin, S.A. Weatherford, J.P. Hoeflinger, D.A. Padua, and P.M. Petersen. The Polaris Internal Representation. *International Journal of Parallel Programming*, 22(5):553–586, 1994.
7. R.B. Fisher. Model invocation for three dimensional scene understanding. In J. McDermott, editor, *Proc. 10th International Joint Conference on Artificial Intelligence*, pages 805–807. Morgan Kaufmann, 1987.
8. R.B. Fisher. Best-first and ten other variations of the interpretation-tree model matching algorithm. DAI Research Paper No. 717, Dept. of Artificial Intelligence, University of Edinburgh, Edinburgh, UK, September 1994.
9. R.B. Fisher, A.W. Fitzgibbon, M. Waite, E. Trucco, and M.J.L Orr. Recognition of complex 3-D objects from range data. In S. Impedovo, editor, *Proc. 7th International Conference on Image Analysis and Processing*, pages 509–606, Monopoli, Bari, Italy, September 1993.

10. R.B. Fisher, D.K. Naidu, and D. Singhal. Rejection of spurious reflections in structured illumination range finders. In *Proc. 2nd Conference on Optical 3D Measurement Techniques*, Zurich, Switzerland, October 1993.
11. Message Passing Interface Forum. MPI: A Message-Passing Interface Standard. Technical Report CS-94-230, University of Tennessee, Computer Science Department, Knoxville, TN, May 1994. (published in the International Journal of Supercomputing Applications, Volume 8, Number 3/4, 1994).
12. D. Gelernter. Generative communication in Linda. *ACM Transactions on Programming Languages and Systems*, 7(1):80–112, January 1985.
13. W.E.L. Grimson. *Object Recognition By Computer: The Role of Geometric Constraints.* MIT Press, 1990.
14. W. Hasselbring. CELIP: A cellular language for image processing. *Parallel Computing*, 14(5):99–109, May 1990.
15. W. Hasselbring. Prototyping parallel algorithms with ProSet-Linda. In J. Volkert, editor, *Parallel Computation (Proc. Second International ACPC Conference)*, volume 734 of *Lecture Notes in Computer Science*, pages 135–150, Gmunden, Austria, October 1993. Springer-Verlag.
16. W. Hasselbring and R.B. Fisher. Investigating parallel interpretation-tree model matching algorithms with ProSet-Linda. DAI Research Paper No. 722, University of Edinburgh, Dept. of Artificial Intelligence, Edinburgh, UK, December 1994. (also available as Software-Technik Memo Nr. 77, University of Dortmund).
17. M.F. Kaashoek, H.E. Bal, and A.S. Tanenbaum. Experience with the distributed data structure paradigm in Linda. In *USENIX/SERC Workshop on Experiences with Building Distributed and Multiprocessor Systems*, pages 175–191, Ft. Lauderdale, FL, October 1989.
18. P. Kruchten, E. Schonberg, and J.T. Schwartz. Software prototyping using the SETL programming language. *IEEE Software*, 1(4):66–75, October 1984.
19. E.L. Lawler and D.E. Wood. Branch-and-bound methods: a survey. *Operations Research*, 14(4):699–719, July 1966.
20. J.T. Schwartz, R.B.K. Dewar, E. Dubinsky, and E. Schonberg. *Programming with Sets – An Introduction to SETL.* Springer-Verlag, 1986.
21. Scientific Computing Associates, New Haven, CT. *C-Linda User's Guide & Reference Manual*, 1992.
22. E. Trucco and R.B. Fisher. Computing surface-based representations from range images. In *Proc. IEEE International Symposium on Intelligent Control (ISIC-92)*, pages 275–280, Glasgow, UK, August 1992.
23. G.V. Wilson. Assessing the usability of parallel programming systems: The Cowichan problems. In *Proc. IFIP WG10.3 Working Conference on Programming Environments for Massively Parallel Distributed Systems*, Monte Verita, Ascona, Switzerland, April 1994. Birkhäuser Verlag AG.

This article was processed using the LaTeX macro package with LLNCS style