Implementing Parallel Algorithms based on Prototype Evaluation and Transformation

W. Hasselbring
University of Dortmund, Computer Science Department, Software Technology,
D-44221 Dortmund, Germany

P. Jodeleit
LVE Verfahrenselektronik GmbH, D-45138 Essen, Germany

M. Kirsch
Mannesmann Autocom GmbH, Information Systems,
D-40547 Düsseldorf, Germany

Abstract

Combining parallel programming with prototyping is aimed at alleviating parallel programming by enabling the programmer to make practical experiments with ideas for parallel algorithms at a high level, neglecting low-level considerations of specific parallel architectures in the beginning of program development. Therefore, prototyping parallel algorithms is aimed at bridging the gap between conceptual design of parallel algorithms and practical implementation on specific parallel systems.

The essential prototyping activities are programming, evaluation and transformation of prototypes. This paper gives a report on some experience with implementing parallel algorithms based on prototype evaluation and transformation employing the ProSet-Linda approach.

1 Introduction

Parallel programming is conceptually harder to undertake and to understand than sequential programming, because a programmer often has to cope with the coexistence and coordination of multiple parallel activities. Prototyping is used to *explore* the essential features of a proposed system through practical experimentation before its actual implementation to make the correct design choices early in the process of software development. Early experimentation with alternate algorithm choices or problem decompositions for parallel applications is suggested to make parallel programming easier.

To be useful, prototypes must be *built* rapidly, and designed in such a way that they can be *modified* rapidly. Therefore, prototypes should be built in very high-level languages to make them rapidly available. Consequently, a prototype is usually not a very efficient program since the language should offer constructs which are semantically on a very high level, and the runtime system has a heavy burden for executing these highly expressive constructs. The primary goal of parallel programming—decreasing the execution time for an application program—is not the first goal with prototyping parallel algorithms. The first goal is to experiment with ideas for parallel algorithms before mapping programs to fit for specific parallel architectures to achieve high speedups.

Prototypes may be classified as throwaway, experimental or evolutionary [11]. A throwaway prototype describes a product designed to be used only to help identify requirements for a new system. Experimental prototyping focuses on the technical implementation of a development goal. In evolutionary prototyping, a series of prototypes is produced that complies with an acceptable behavior, according to the feedback from prototype evaluations. Once the series has converged, the result may be turned into a software product by transformations.

This raises issues of software engineering: once we are satisfied with the prototype, how do we transform it systematically into a production-efficient program? This question is fairly difficult for sequential programs [28], but not satisfactorily solved as yet for prototypes of parallel algorithms. Therefore, such transformations are usually accomplished manually. Before building transformation tools it seems to be reasonable to gain some knowledge about the requirements on such tools through practical experience and to develop a theoretical foundation for such tools. The present paper discusses the systematic manual transformation of ProSet-Linda prototypes into efficient message-passing programs (PVM and MPI) and shared-memory programs on multi-processor workstations.

Various experience with developing sequential software systems using prototyping has been made [14]. This paper presents some experience with implementing *parallel* algorithms for computer vision (Section 2) and for the Salishan/Cowichan problems (Section 3) based on prototype evaluation and transformation. Section 4 discusses related work and Section 5 draws some conclusions. The ProSet-Linda prototyping approach is presented in Appendix A which should be consulted first when this approach is new to the reader.

2 Developing Parallel Algorithms for Computer Vision

The ProSet-Linda approach has been applied to the development of parallel algorithms for high-level three-dimensional computer vision [19]. To give a report on the experience made, first three-dimensional computer vision and interpretation-tree search for model matching within this context is discussed. Parallel interpretation-tree search with ProSet-Linda, the evaluation of the prototypes and the transformation of the most promising prototype into efficient implementations is discussed in the subsequent subsections.

2.1 Three-Dimensional Computer Vision

Computer vision is commonly divided into several levels. With three-dimensional computer vision [10], low-level vision is concerned with processing range data acquired by a laser range scanner to eliminate noise. Medium-level vision is concerned with identifying geometric surfaces. High-level vision tries, for example, to identify the shape and position of data objects using matched model features. In high-level vision, first the *model invocation* process pairs likely model and data features for further consideration. Model matching then uses the candidate matches proposed by the invocation to form consistent groups of matches. Fig. 1 illustrates this process.

2.2 Interpretation-Tree Search for Model Matching

The classical control algorithm for symbolic model matching in computer vision is the *Interpretation-Tree* search algorithm [15]. The algorithm searches a tree of potential data-to-model 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 using a set of consistency constraints according to some (geometric) criterion. The goal of the search algorithm is to maximize the set of consistent data-to-model 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 [15]. However, even with these effective forms of pruning, the algorithm still can have

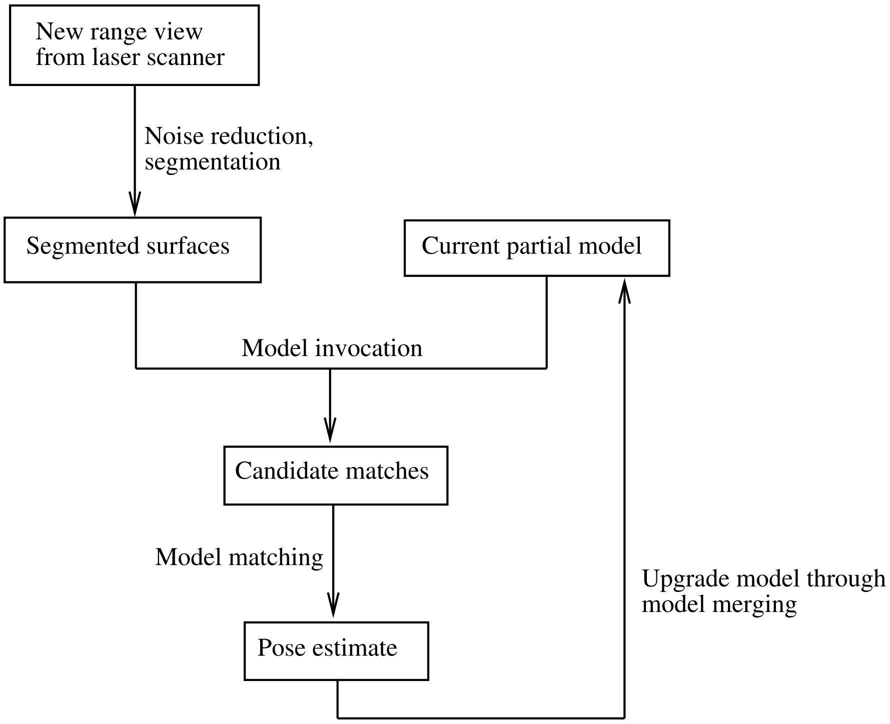

Figure 1: The role of model matching in three-dimensional computer vision for model acquisition from multiple range views.

exponential complexity, making the standard interpretation-tree search algorithm unsuitable for use in scenes with many features.

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 [9]. The tree in Fig. 2 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. The segmented surfaces in Fig. 1 are the data features and the current partial model consists of the model features. Model invocation uses the model and data properties to pair likely model and data features. It produces a sorted list of consistent data-to-model pairs $(data_i, model_i, A_i)$ where A_i is the compatibility measure (plausibility) of the features $data_i$ and $model_i$. The list is initially sorted with larger A_i values at the top. Model matching uses Ω to build the interpretation tree.

2.3 Parallel Interpretation-Tree Search with ProSet-Linda

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 discussed below 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.

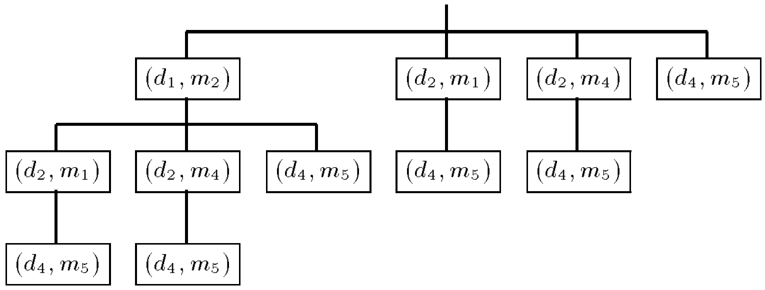

Figure 2: The interpretation tree for $\Omega = [(d_1, m_2), (d_2, m_1), (d_2, m_4), (d_4, m_5)]$. The d_i are data features and the m_i are model features. The root of the interpretation tree has no pairings. Each data feature appears (in order) at most once in a branch. At each node at level k in the tree, therefore, there is a hypothesis with k features matched.

Four parallel variations of non-wildcard interpretation-tree search have been investigated as feasibility studies for efficient parallelization [19]. The evaluation of the prototypes serves to select the most promising way to parallelize the search. Based on this evaluation, the best parallel algorithm has been refined into efficient implementations [24]. Below, only this algorithm is presented, because it has been transformed into efficient implementations. In this paper, only small parts of the code can be presented. Refer to [19] and [24] for more detailed presentations.

The so-called parallel *best-first* search tree algorithm is based on the sequential best-first search tree algorithm which assumes that it is possible to evaluate how well sets of model features match sets of data features (based on the plausibilities from the model invocation and consistency measures as the set sizes grow) [9]. As any real problem is likely to provide some useful heuristic ordering constraints, the potential for speeding up the matching process is large. The best-first search tree algorithm searches for the first *plausible* solution (usually not the optimal solution).

Fig. 3 displays the coarse structure of the master-worker program for this best-first search tree algorithm. Arrows indicate access to the tuple spaces. These access patterns are only shown for one of the identical worker processes. The program uses two tuple spaces: One for the work tasks (WORK) and one for the results (RESULT). The master (the main program) spawns a number of worker processes to do the work. This number is an argument to the main program. The initial task tuples, which represent the nodes at the first level of the interpretation-tree, are deposited at tuple space WORK. 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. Synchronization between the master and the workers is achieved when the first satisfactory match has been deposited by a worker at RESULT.

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 loop of the worker ensures that these constraints are satisfied:

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

The hypotheses from the model invocation are stored in the tuple Hypotheses. The set of pairs MyPath represents the current partial branch in the tree. The condition ‘Entry(1) > x(1)’ enforces

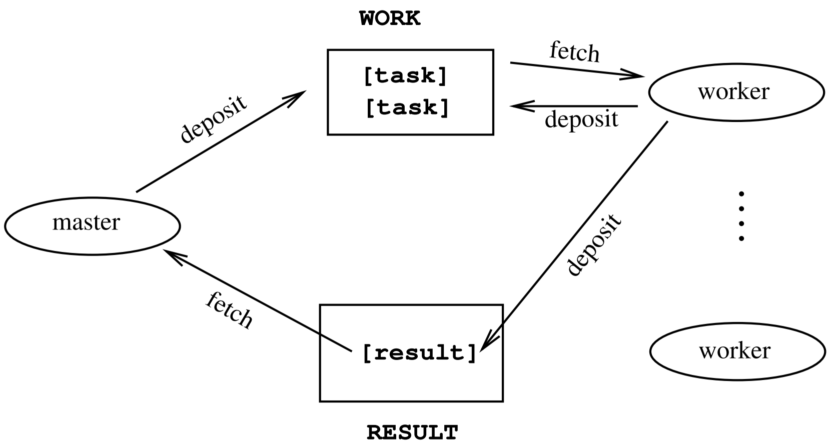

Figure 3: The coarse structure of the master-worker program for the parallel best-first search tree algorithm.

the data feature ordering constraint. Only extensions that satisfy the normal binary constraints are accepted (the Boolean function Consistent checks this). A match is satisfactory when the termination number of matched features has been reached. Extension stops when this threshold is reached. The threshold is a program argument. 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).

The central data structure for this algorithm 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 data-to-model 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, 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**;

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. 'Hypotheses(1)(1)' selects the data feature and 'Hypotheses(1)(2)' selects the model feature from the first hypothesis.

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 refers to the next entry. The integer `0` indicates the end of the queue. Fig. 4 illustrates the structure of this queue.

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. 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 in tuple space, 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:

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

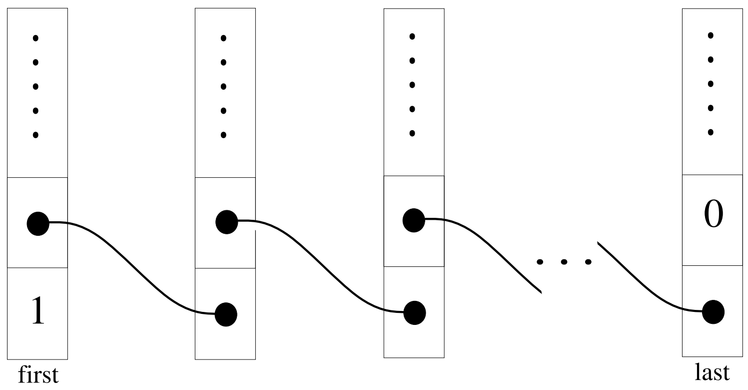

Figure 4: The distributed priority queue for parallel best-first search. The integer values `1` and `0` are used to indicate the top and the end of the queue, respectively. The intermediate entries are *identified* by the contained atoms. We use black circles to represent the atoms. A link between two 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).

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.

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.

The priority queue is stored as a *distributed data structure* [22] 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 before, 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. A variety of other data structures, such as distributed priority sorted heaps or distributed sorted trees, could be used to implement the priority queue.

2.4 Evaluation of the Prototypes

For testing the parallel algorithms, the output from the low- and medium-level components of the IMAGINE2 [10] system for range images of workpieces is used. Some experimental results with the ProSet-Linda prototype for the parallel best-first search algorithm are displayed in Fig. 5. Fig. 5a shows the number of visited nodes in relation to the number of workers and Fig. 5b 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.

The parallel best-first algorithm 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

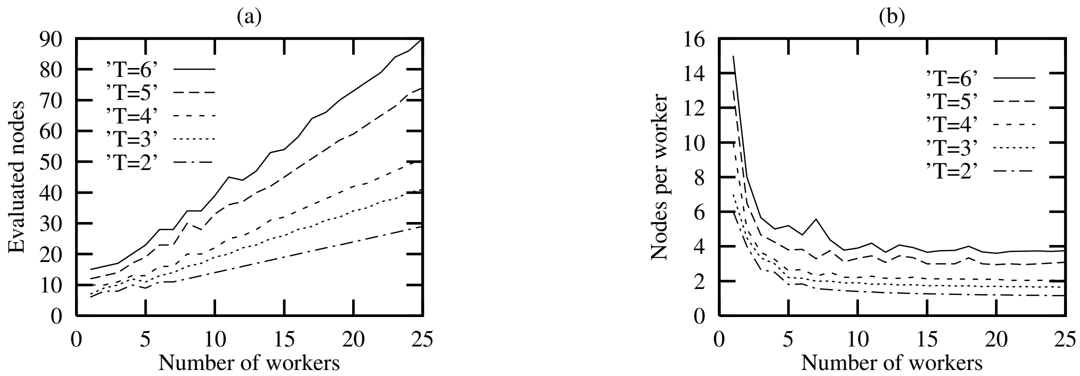

Figure 5: The experimental results for the ProSet-Linda implementation of the parallel best-first search algorithm.

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. Therefore, the experimental evaluation showed that the parallel best-first search algorithm is the most promising way to parallelize model matching for three-dimensional computer vision. Refer to [19] for a more detailed description of the evaluation.

Another main observation to make at this point is: because the sequential variations of interpretation-tree model matching algorithms were presented in a *set-oriented* way [9], it was quite straightforward to implement them and the alternative parallel implementations in ProSet-Linda and then compare them, in only a few weeks. The prototypes for the developed algorithms could be regarded as *executable specifications*.

2.5 Transforming the Most Promising Prototype into Efficient Implementations

As a consequence of the evaluation, the prototype of the parallel best-first algorithm has been transformed into efficient implementations [24]. The ProSet-Linda prototype is first transformed into a C-Linda implementation. With Linda, it is easy to program with different styles, e.g. with distributed data structures, active data structures and message passing [2]. The transformations between C-Linda styles are discussed in [2]. Then, the C-Linda implementation has been transformed into a message-passing style to serve as a preliminary step for a message-passing implementation. In this project, a MPI library (Message Passing Interface) is used [7]. The transformation from message-passing style C-Linda programs into MPI programs is straightforward: only the coordination part is changed.

For an efficient implementation with MPI, the implementation of the priority queue needs to work efficiently on *distributed* memory machines. Parallelizing a priority queue requires the processing of several request in parallel. Classical sequential algorithms use a heap for maintaining a priority queue. Early approaches tried to use parallel priority queue algorithms developed for shared memory machines on distributed memory machines, too, by simulating the required shared memory [31]. This results in imbalanced load of the processors' memories. Load balancing processes are required which produce additional simulation overhead.

Thus, new algorithms for distributed memory were investigated. These algorithms use local sequential priority queues on each processor to maintain a global parallel priority queue [23]. Early attempts in this direction proposed algorithms that do not guarantee the selection of the element with global maximum priority. The semantics of a priority queue changed in a way not acceptable for many applications. Recently published algorithms provide real parallel priority queues for distributed memory without the above mentioned problems [32].

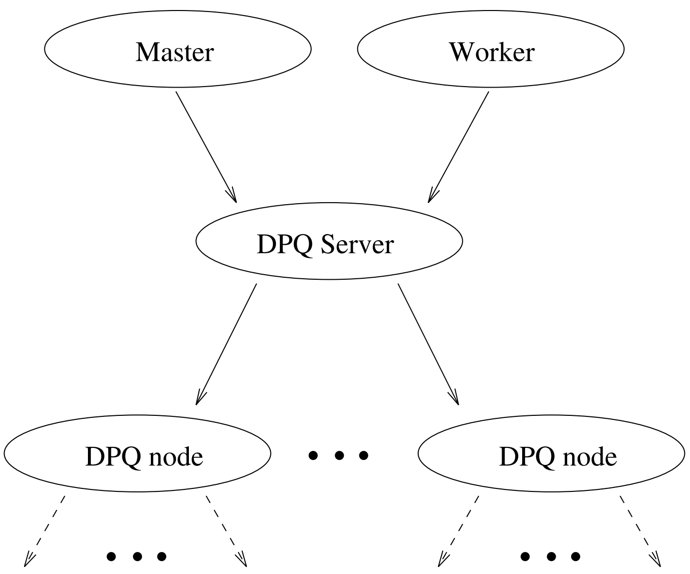

Figure 6: The coarse structure of the distributed priority queue in the low-level C-Linda and MPI implementations.

Our parallel algorithm uses an n-ary global heap consisting of local priority queues residing on each processor [26]. The global heap property is kept across all local priority queues, i.e. in a minimum-heap each node locally maintains a priority queue containing only elements that are smaller than the elements kept at the children nodes. Due to the use of an n-ary heap and a simple load balancing scheme this algorithm is easily scalable to the number of available processors.

The distributed priority queue (DPQ), which has been implemented with C-Linda and MPI, uses a divide-and-conquer model. The DPQ is a distributed heap with a root node and several worker processes connected in a tree structure (see Fig. 6). According to the heap property, all items stored at one particular node have a larger priority than all those stored at any of its children. Each process of the DPQ stores its items in a local heap. Items are inserted or deleted on a particular node in a round-robin scheme which provides a perfectly balanced distribution of the data.

The DPQ is accessed at the DPQ server process which invokes the appropriate actions to maintain the heap property of the whole distributed heap. For every request — push or pop — the DPQ server process determines the host process which actually has to change its number of stored items. Then, the work is propagated down the path to this host. The DPQ server process is able to process the next request as soon as the previous request is handed over to a DPQ node at a lower level of the distributed heap. Due to the distribution scheme, congestion along a path in the distributed heap is avoided by choosing another subtree each time a request is relayed down the heap.

Fig. 7a shows the total elapsed execution times of the C-Linda (message-passing style) and MPI implementations on a network of six SparcStations™ connected by an Ethernet network. The C-Linda program uses the Network-C-Linda system from SCA [33], and the MPI implementation uses CHIMP/MPI (Common High-Level Interface to Message Passing) from the Edinburgh Parallel Computing Centre [1]. The elapsed times include the time required for starting the processes over the network. With both implementations, the number of evaluated nodes of the search tree increases with the number of workers as predicted by the prototype. Fig. 7b shows also that the evaluation times for each node in the interpretation tree decreases when the number of workers is increased: the quality of the result is improved while the total execution times of model matching remains approximately constant, even when the number of workers exceeds the number of available processors.

At least with the C-Linda implementation, the total execution time decreases as long as the number

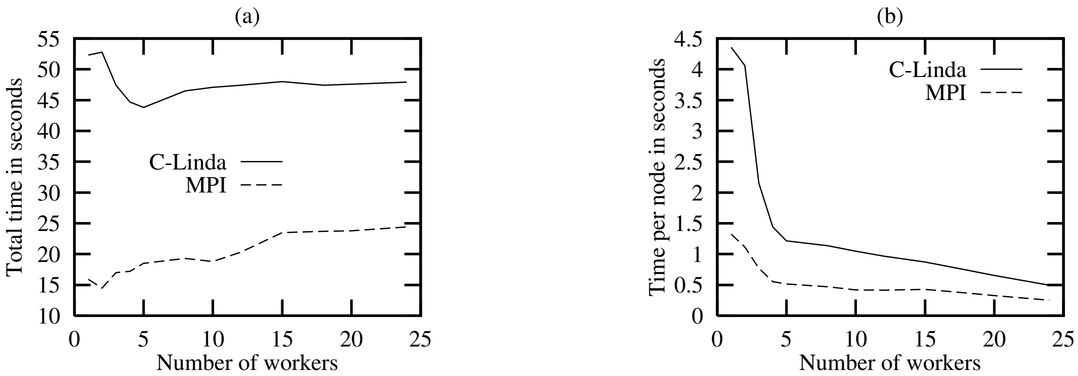

Figure 7: Comparison of the total execution times and the times per node for the transformed C-Linda and the MPI implementations on a network of six SparcStations™. The termination threshold for satisfactory matches for the displayed times is $T = 6$.

of workers does not exceed the number of available processors. Fig. 7a shows that the execution time remains almost constant for fifteen or more workers with both implementations. The results are different for less than fifteen workers because the C-Linda system applies automatic load balancing for the available hosts. CHIMP/MPI assigns processes to hosts in a strict round-robin scheme. Despite the load balancing, the C-Linda implementation is not as fast as the MPI implementation. The communication procedures of MPI are on a lower level than the C-Linda primitives, allowing optimizations to a greater extent. Refer to [24] for a detailed discussion of the tranformed implementations.

3 Developing Parallel Algorithms for the Salishan/Cowichan Problems

The Salishan [8] and the Cowichan [34] problems are suites for assessing the usability of parallel programming systems. They have been implemented with the ProSet-Linda approach [21]. To save space, only the experience with one problem, viz. Hamming's Problem, are reported in the present paper.

Input to Hamming's Problem is an ordered list of prime numbers p_1, …, p_k and an integer n. Hamming's Problem is to produce the ordered stream containing integers lower or equal to n matching the product:

$$p_1^{i_1} \cdot p_2^{i_2} \cdot \ldots \cdot p_k^{i_k}$$

with $i_j \in \mathbb{N}_0$ for $j \in \{1, \ldots, k\}$. These numbers are called *hamming numbers* and the problem is called *extended hamming problem*. The simple version restricts the problem to the primes 2, 3, and 5. Another formulation for the problem is to find all integers between 1 and n with all prime factors occuring in the given list of primes.

The steps for transforming the prototype solutions for this problem into efficient implementations are illustrated in Fig. 8. After the evaluation of prototypes for different parallelizations, the prototypes are transformed within ProSet-Linda itself to arrive at a conceptual level of C-Linda [33]. In the next step, the ProSet-Linda prototype is transformed into a C-Linda implementation. With the step from ProSet-Linda to C-Linda, the transformation of the sequential parts—in particular the data structures—is the first task. The transformation of the coordination part is straightforward.

With the transformation from C-Linda to the Multi-Thread Architecture [29], the tuple space is simply stored in shared memory. The Multi-thread implementation is directly derived from the first C-Linda implementation. The Multi-thread implementation allows parallel execution on multi-processor SparcStations™.

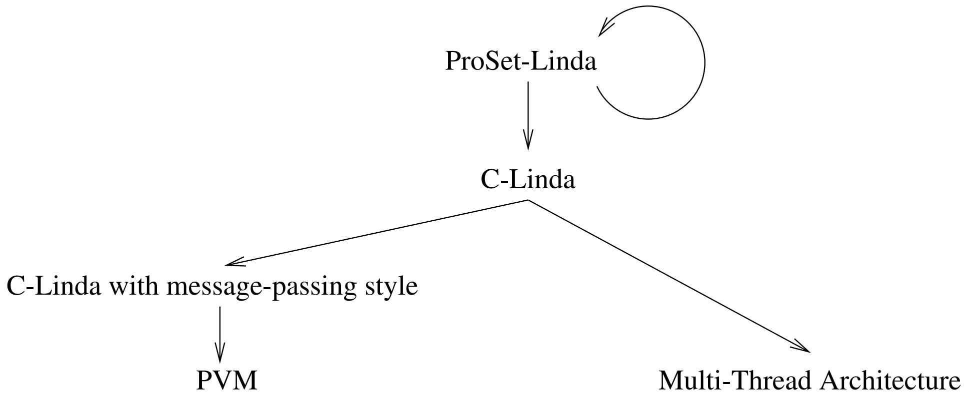

Figure 8: The transformation steps from ProSet-Linda to low-level programming systems which have been applied to the Salishan/Cowichan problems.

For the transformation from C-Linda into PVM, first the C-Linda implementation is transformed into a message-passing style C-Linda program, as discussed in subsection 2.5. Again, the transformation from message-passing style C-Linda programs into PVM programs is straightforward: only the coordination part is changed. PVM (Parallel Virtual Machine) is a popular message-passing library which is available on workstation networks as well as several parallel computer systems [12].

The experimental results for the transformed low-level implementations are displayed in Fig. 9. The C-Linda and PVM implementations are measured on a network of six SparcStation 10 (single processor) and the Multi-thread implementation is measured on a SparcStation 10/512 with two processors. The execution times for the PVM and Multi-thread implementations decrease even when more worker processes are spawned than processors are available. These good speedups are achieved because process creation is cheap with the PVM and Multi-thread implementations and because the work is better distributed among the processes. The speedups through improved load balancing are eliminated with the C-Linda implementation by the high costs for process creation. For a detailed discussion of these transformations and evaluations refer to [21].

4 Related Work

Various approaches to prototyping parallel algorithms with very high-level parallel programming languages intend to alleviate the development of parallel algorithms in quite different ways. Some transformations of parallel prototypes into efficient implementations are discussed in the literature. The transformation of sequential programs is discussed in [28].

In [20], high-level parallel algorithm specifications are refined within PSETL, which is a data-parallel extension to SETL. High-level PSETL code is successively transformed manually into lower-level architecture-specific PSETL code.

The Crystal [4] approach starts from a high-level functional problem specification, through a sequence of optimizations tuned for particular parallel machines, leading to the generation of efficient target code with explicit communication and synchronization. This approach to automation is to design a compiler that classifies source programs according to the communication primitives and their cost on the target machine and that maps the data structures to distributed memory, and then generates parallel code with explicit communication commands. Regarding those classes of problems for which the default mapping strategies of the compiler are inadequate, Crystal provides special language constructs for incorporating domain specific knowledge by the programmer and directing the compiler in its mapping.

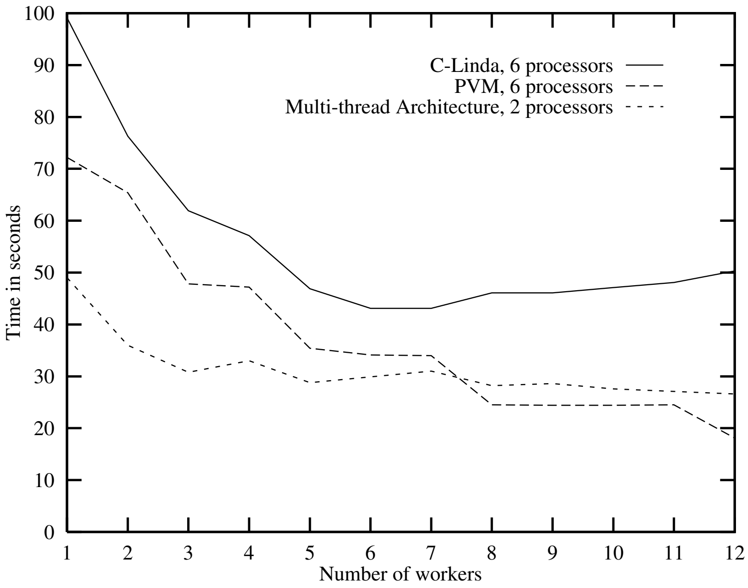

Figure 9: The experimental results for the transformed low-level implementations for Hamming's Problem.

Transformation techniques have been developed for the Proteus prototyping language [30]. Proteus' semi-automatic refinement system is based on algebraic specification techniques and category theory to transform prototypes to implementations on specific architectures. For the time being, these transformations are restricted to the data-parallel constructs of Proteus [30]. [27] discusses the transformation of data-parallel Proteus programs to low-level systems such as High Performance Fortran (HPF) and message-passing libraries. In addition to the data-parallel constructs, Proteus provides some constructs for control parallelism, but transformations of control-parallel prototypes are not discussed.

The automatic or semi-automatic transformation of *control-parallel* prototypes into efficient low-level programs is still an unsolved problem and subject to further research.

5 Conclusions

To build a parallel system, you should start with executable prototypes to study the feasibility of ideas for parallelization (neglect the execution performance in the first instance). Powerful tools are needed to make prototyping of parallel algorithms and systems feasible.

This paper reports on the experience with the development of parallel algorithms for computer vision applications and for the Salishan/Cowichan problems employing the ProSet-Linda approach. Prototypes for several parallel algorithms have been developed, evaluated and transformed. The evaluation showed that not all algorithmic variations are good candidates for efficient 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, because the effort to practically evaluate ideas for parallelization with low-level languages is higher than it is with a prototyping language.

Acknowledgements

This work has been partially supported by the TRACS program funded by the Human Capital and Mobility program of the European Commission.

The authors would like to thank Bob Fisher for the cooperation on the investigation of parallel algorithms for model matching in three-dimensional computer vision and Greg Wilson for the help with the Cowichan problems.

References

[1] R. Alasdair, A. Bruce, J.G. Mills, and A.G. Smith. CHIMP/MPI User Guide. Technical report, Edinburgh Parallel Computing Centre, Edinburgh, UK, June 1994.

[2] N. Carriero and D. Gelernter. *How to write parallel programs.* MIT Press, 1990.

[3] N. Carriero and D. Gelernter. Coordination languages and their significance. *Communications of the ACM*, 35(2):96–107, February 1992.

[4] M. Chen, Y. Choo, and J. Li. Crystal: Theory and pragmatics of generating efficient parallel code. In B.K. Szymanski, editor, *Parallel Functional Languages and Compilers*, Frontier Series, pages 255–308. ACM Press, 1991.

[5] E.W. Dijkstra. Hierarchical ordering of sequential processes. *Acta Informatica*, 1:115–138, 1971.

[6] 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. IEEE Computer Society Press.

[7] J.J. Dongarra, S.W. Otto, M. Snir, and D. Walker. A message passing standard for MPP and workstations. *Communications of the ACM*, 39(7):84–90, July 1996.

[8] J.T. Feo. *A Comparative Study of Parallel Programming Languages: The Salishan Problems.* North Holland, 1992.

[9] 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.

[10] 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.

[11] C. Floyd. A systematic look at prototyping. In R. Budde, K. Kuhlenkamp, L. Mathiassen, and H. Züllighoven, editors, *Approaches to Prototyping*, pages 1–18. Springer-Verlag, 1984.

[12] A. Geist, A. Beguelin, J.J. Dongarra, W. Jiang, R. Manchek, and V. Sunderam. *PVM: Parallel Virtual Machine A Users' Guide and Tutorial for Networked Parallel Computing.* The MIT Press, 1994.

[13] D. Gelernter. Generative communication in Linda. *ACM Transactions on Programming Languages and Systems*, 7(1):80–112, January 1985.

[14] V.S. Gordon and L.M. Bieman. Rapid prototyping: Lessons learned. *IEEE Software*, 12(1):85–95, January 1995.

[15] W.E.L. Grimson. *Object Recognition By Computer: The Role of Geometric Constraints.* MIT Press, 1990.

[16] R.H. Halstead. Multilisp: A language for concurrent symbolic computation. *ACM Transactions on Programming Languages and Systems*, 7(4):501–538, October 1985.

[17] W. Hasselbring. *Prototyping Parallel Algorithms in a Set-Oriented Language.* PhD thesis, Department of Computer Science, University of Dortmund, 1994. (Published by Verlag Dr. Kovač, Hamburg).

[18] W. Hasselbring. The ProSet-Linda approach to prototyping parallel systems. *The Journal of Systems and Software*, 1997. (to appear).

[19] W. Hasselbring and R.B. Fisher. Using the ProSet-Linda Prototyping Language for Investigating MIMD Algorithms for Model Matching in 3-D Computer Vision. In A. Ferreira and J. Rolim, editors, *Parallel Algorithms for Irregularly Structured Problems*, volume 980 of *Lecture Notes in Computer Science*, pages 301–315. Springer-Verlag, September 1995.

[20] S. Flynn Hummel, S. Talla, and J. Brennan. The refinement of high-level parallel algorithm specifications. In *Proc. Working Conference on Programming Models for Massively Parallel Computers (PMMP '95)*, Berlin, Germany, October 1995. IEEE Computer Society Press.

[21] P. Jodeleit. Implementing the Salishan and Cowichan problems based on prototype evaluation and transformation. Master's thesis, Department of Computer Science, University of Dortmund, May 1996. (in German).

[22] 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.

[23] R.M. Karp and Y. Zhang. Randomized parallel algorithms for backtrack search and branch-and-bound computation. *Journal of the ACM*, 40(3):765–789, 1993.

[24] M. Kirsch. Implementing parallel model matching algorithms for 3-D computer vision with Linda and message passing based on prototype evaluation and transformation. Master's thesis, Department of Computer Science, University of Dortmund, April 1996.

[25] P. Kruchten, E. Schonberg, and J.T. Schwartz. Software prototyping using the SETL programming language. *IEEE Software*, 1(4):66–75, October 1984.

[26] B. Mans. Portable distributed priority queues with MPI. In *Proc. International Conference on High Performance Computing*, pages 16–21. New Delhi, India, December 1995.

[27] L. Nyland, J. Prins, A. Goldberg, P. Mills, J. Reif, and R. Wagner. A refinement methodology for developing data-parallel applications. In *Proc. EuroPar'96*, Lecture Notes in Computer Science, Lyon, France, August 1996. Springer-Verlag.

[28] H.A. Partsch. *Specification and Transformation of Programs.* Springer-Verlag, 1990.

[29] M.L. Powell, S.R. Kleinman, S. Barton, D. Shah, D. Stein, and M. Weeks. SunOS Multi-thread Architecture. In *Proc. USENIX Winter '91 Technical Conference*, Dallas, TX, 1991.

[30] J. Prins and D. Palmer. Transforming high-level data-parallel programs into vector operations. In *Proc. Fourth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 119–128, San Diego, CA, May 1993.

[31] A. Ranade, S. Cheng, E. Deprit, J. Jones, and S. Shih. Parallism and locality in priority queues. In *Sixth IEEE Symposium on Parallel and Distributed Processing*, pages 490–496, October 1994.

[32] P. Sanders. Fast priority queues for parallel branch-and-bound. In A. Ferreira and J. Rolim, editors, *Parallel Algorithms for Irregularly Structured Problems*, volume 980 of *Lecture Notes in Computer Science*, pages 379–393. Springer-Verlag, September 1995.

[33] Scientific Computing Associates, New Haven, CT. *C-Linda User's Guide & Reference Manual*, 1992.

[34] 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.

A Prototyping Parallel Algorithms with ProSet-Linda

ProSet-Linda combines the sequential prototyping language ProSet [6] with the coordination language Linda [3] to obtain a parallel programming language as a tool for prototyping parallel algorithms [17, 18]. The procedural, set-oriented language ProSet [6] is a successor to SETL [25]. ProSet is an acronym for PROtotyping with SETs. The high-level structures that ProSet provides qualify the language for prototyping. Refer to [25] for a case study using SETL for prototyping.

A.1 Basic Concepts

ProSet provides the data types atom, integer, real, string, Boolean, tuple, set, function, and module. As a prototyping language, 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 $\{2 * \mathrm{x} : \mathrm{x} \in [1..10] | \mathrm{x} > 5\}$ which yields the set $\{12, 14, 16, 18, 20\}$. The quantifiers of predicate calculus are provided ($\exists$, $\forall$). The control structures have ALGOL as one of its ancestors.

A.2 Parallel Programming

To support prototyping of parallel algorithms, a prototyping language must provide simple and powerful means for dynamic creation and coordination of parallel processes. In ProSet-Linda, the concept for process creation via Multilisp's futures [16] is adapted to set-oriented programming and combined with Linda's [13] concept for synchronization and communication. 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 (this is necessary with synchronous point-to-point message passing).

A.2.1 Process Creation

Process creation in ProSet-Linda is provided through the unary operator ||, which may be applied to a function call. A new process will be spawned to compute the value of this expression concurrently with the spawning process similar to *futures* in Multilisp [16]. If this *process creator* || is applied to an expression that is assigned to a variable, the spawning process continues execution without waiting for the termination of the newly spawned process. At any time the *value* of this variable is needed, the requesting process will be suspended until the future *resolves* (the corresponding process terminates) thus allowing concurrency between the *computation* and the *use* of a value. Consider the following statement sequence to see an example:

```
x := || p();    -- Statement 1
...             -- Some computations without access to x
y := x + 1;     -- Statement 2
```

After statement 1 is executed in the above example, process p() runs in parallel with the spawning process. Statement 2 will be suspended until p() terminates. If p() resolves before statement 2 has started execution, then the resulting value will be assigned immediately.

Side effects and write parameters are not allowed for parallel processes in ProSet-Linda. Synchronization and communication is done only via tuple-space operations.

A.2.2 Synchronization and Communication

Linda is a coordination language which extends a sequential language by means for synchronization and communication through so-called *tuple spaces* [13]. Synchronization and communication in ProSet-Linda are carried out through several atomic operations on tuple spaces: 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-Linda supports multiple tuple spaces. Several library functions are provided for handling multiple tuple spaces dynamically.

ProSet-Linda 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) **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 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 for values **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. With **meet**, in-place updates of specific tuple components are supported.

A.3 An Introductory Example: the Dining Philosophers Problem

As an introductory example, we present the complete parallel solution to the dining philosophers problem. The dining philosophers problem is a classical problem in parallel programming which has been posed by Dijkstra [5]. It is often used to test the expressivity of new parallel languages.

The ProSet-Linda solution in Figure 10 is derived from the C-Linda version in [2]. In the C-Linda version, the philosophers first fetch their left and then their right chopsticks. In the ProSet-Linda version, this order is not specified. This is accomplished by the use of multiple templates for one **fetch** statement. The **fetch** statement suspends until a matching tuple is available. Then, the

```
program DiningPhilosophers;
  visible constant   n := 5, -- Number of philosophers
                     TS := CreateTS (); -- New tuple space
begin
  for i in [ 0 ... n-1 ] do
    -- Deposit chopsticks and room tickets at the tuple space:
    deposit [ "chopstick", i ] at TS end deposit;
    if i ≠ n-1 then -- One ticket less than the number of philosophers
      deposit [ "room ticket" ] at TS end deposit;
      || phil(i); -- Spawn the next philosopher
    end if;
  end for;
  phil(n-1); -- The main program becomes the last philosopher

  procedure phil (i);
  begin
    loop
      think ();
      fetch ( "room ticket" ) at TS end fetch;
      -- Fetch left and right chopstick in arbitrary order:
      fetch ( "chopstick", i ) ⇒
              -- Left chopstick fetched, fetch the right one:
              fetch ( "chopstick", (i+1) mod n ) at TS end fetch;
        xor ( "chopstick", (i+1) mod n ) ⇒
              -- Right chopstick fetched, fetch the left one:
              fetch ( "chopstick", i ) at TS end fetch;
        at TS
      end fetch;
      eat ();
      -- Return the fetched chopsticks and the room ticket:
      deposit [ "chopstick", i ] at TS end deposit;
      deposit [ "chopstick", (i+1) mod n ] at TS end deposit;
      deposit [ "room ticket" ] at TS end deposit;
    end loop;
  end phil;
end DiningPhilosophers;
```

Figure 10: Solution for the dining philosophers problem. The function **CreateTS** creates a new tuple space. The templates are enclosed in parentheses and not in brackets in order to set the templates apart from tuples

enclosed statement which is specified for the selected template is executed. The program works for arbitrary $n > 1$.

To prevent deadlock, only four philosophers (or one less than the total number of philosophers) are allowed into the room at any time to guarantee to be at least one philosopher who is able to make use of both, his left and his right chopstick. In [2] this is demonstrated with the *pigeonhole principle*: in every distribution of the n chopsticks among the $n-1$ philosophers with table tickets, there must be at least one philosopher who gets two chopsticks. For a detailed discussion of prototyping parallel algorithms with ProSet-Linda refer to [17].