The ProSet-Linda Approach to Prototyping Parallel Systems[1]

Wilhelm Hasselbring

Department of Computer Science, University of Dortmund, D-44221 Dortmund, Germany

Parallel programming is conceptually harder to undertake and to understand than sequential programming, because a programmer often has to manage 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. Approaches to prototyping parallel algorithms with very high-level parallel programming languages intend to alleviate the development of parallel algorithms. To make parallel programming easier, early experimentation with alternate algorithm choices or problem decompositions for parallel applications is suggested. This paper presents the ProSet-Linda approach which has been designed for prototyping *parallel* systems.

1 INTRODUCTION

There has been particular attention on parallel programming and processing within the computer science community during the last years. Several motivations for programming parallel applications exist:

1. Decreasing the execution time for an application program.
2. Increasing the fault-tolerance.
3. Exploiting explicitly the inherent parallelism of an application.

Achieving speedup through parallelism is a common motivation for executing an application program on a parallel computer system. Another motivation is achieving fault-tolerance: for critical applications like controlling a nuclear power plant, a single processor may not be reliable enough. Distributed computing systems are potentially more reliable: as the processors are autonomous, a failure in one processor does not affect the correct function of the other processors. Fault-tolerance can, therefore, be increased by replicating functions or data of the application on several processors. If some of the processors crash, the others can continue performing their tasks.

However, the main motivation for integrating explicit parallelism into a *prototyping* language is to provide means for explicitly modeling inherently parallel applications. Consider, for instance, parallel systems such as air-traffic-control and airline-reservation applications, which must respond to many external stimuli and which are therefore inherently parallel. To deal with nondeterminism and to reduce their complexity, such applications are preferably structured as independent parallel processes.

Combining parallel programming with prototyping intends to alleviate parallel programming on the basis of enabling the programmer to practically experiment with ideas for parallel applications on

[1]Preprint of a paper to appear in *The Journal of Systems and Software*.

a high level neglecting low-level considerations of specific parallel architectures in the beginning of program development. Prototyping parallel algorithms intends to bridge the gap between conceptual design of parallel algorithms and practical implementation on specific parallel systems.

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 above-mentioned 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 specific parallel architectures to achieve high speedups.

The prototyping language ProSet-Linda and some introductory examples are presented in Sections 2 and 3, respectively. The implementation of ProSet-Linda is discussed in Section 4. Section 5 discusses some related work and Section 6 summarizes the paper.

2 THE PROTOTYPING LANGUAGE PROSET-LINDA

ProSet-Linda combines the sequential prototyping language ProSet (Doberkat et al., 1992) with the coordination language Linda (Gelernter, 1985; Carriero and Gelernter, 1992) to obtain a parallel programming language as a tool for prototyping parallel algorithms (Hasselbring, 1994b). ProSet is an acronym for PROTOTYPING WITH SETS. The procedural, set-oriented language ProSet is a successor to SETL (Kruchten et al., 1984). The high-level structures that ProSet provides qualify the language for prototyping. Refer to (Kruchten et al., 1984) for a case study using SETL for prototyping.

2.1 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*x: x in [1..10] | 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.

2.2 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 (Halstead, 1985) futures is adapted to set-oriented programming and combined with Linda's (Gelernter, 1985) 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 interrelations 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).

2.2.1 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 (Halstead, 1985). 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.

In summary: parallel execution is achieved only at creation time of a process and maintained through immediately assigning to a variable, storing in a data structure, returning as a result from a procedure, and depositing in tuple space (this is discussed below). Every time one tries to obtain a copy one has to wait for the termination of the corresponding process and obtains the returned value only then. Additionally, the following statement, which spawns a new process, is allowed:

```
|| p();
```

The return value of such a process will be discarded. This may be compared with a `fork` in the UNIX™ operating system. Side effects and write parameters are not allowed for parallel processes in ProSet-Linda. Synchronization and communication is done only via tuple-space operations.

2.2.2 Synchronization and Communication Linda is a coordination language which extends a sequential language by means for synchronization and communication through so-called *tuple spaces* (Gelernter, 1985). 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 (Hasselbring, 1994b).

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 | (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 `$` may be used as a placeholder for the values of corresponding tuples in tuple space. The expression `$(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. It is allowed to specify multiple templates and an `else`-part within such a statement as will be done in the examples of Section 3.

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 can be regarded as shared objects since they remain in tuple space irrespective of changing them or not. With `meet`, in-place updates of specific tuple components are supported. For a detailed discussion of prototyping parallel algorithms with ProSet-Linda refer to (Hasselbring, 1994b).

3 INTRODUCTORY EXAMPLES

As introductory examples, we present the complete parallel solutions to the classical dining philosophers problem and to the traveling salesman problem.

3.1 The Dining Philosophers Problem

The dining philosophers problem is a classical problem in parallel programming which has been posed by (Dijkstra, 1971). It is often used to test the expressivity of new parallel languages.

The ProSet-Linda solution in Figure 1 is derived from the C-Linda version in (Carriero and Gelernter, 1990). 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 enclosed statement which is specified for the selected template is executed. The program works for arbitrary $\texttt{n} > \texttt{1}$.

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 (Carriero and Gelernter, 1990) 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.

```
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 1. Solution for the dining philosophers problem. The function `CreateTS` creates a new tuple space. The templates in `fetch` operations are enclosed in parentheses and not in brackets in order to set the templates apart from tuples.

3.2 The Traveling Salesman Problem

As a second introductory example, we present the complete parallel solution to the traveling salesman problem in which it is desired to find the shortest route that visits each of a given set of cities exactly once. We want to compute an optimal route for some cities in the Ruhrgebiet, an area in Germany named after the river Ruhr. The selected cities with their connections and distances are displayed in Figure 2. The salesman should start in Essen.

The problem can be solved using *branch-and-bound* (Lawler and Wood, 1966). It uses a tree to structure the search space of possible solutions. The root of the tree is the city in which the salesman should start. Each path from the root to a node represents a partial tour for the salesman. Leaf nodes represent either partial tours without connections to not yet visited cities or complete tours. Complete tours visit each city exactly once. Figure 3 displays the search tree for our selection of cities. The complete tours, in which each city is visited, are set off through thick lines. In general, it is not necessary to search the entire tree: a *bounding rule* avoids searching the entire tree. For the traveling salesman problem, the bounding rule is simple. If the length of a partial tour exceeds the length of an already known complete tour, the partial tour will never lead to a solution better than what is already known.

Parallelism in a branch-and-bound algorithm is obtained by searching the tree in parallel. The main program in Figure 4 stores the cities with their connections in the global constant set `DistTable`. This set is a map which maps pairs of cities to their distance. The distances are specified for each direction. The distances between two cities may be different for different directions (e.g. for one-way connections). The set `Nodes` contains the cities involved. The string `Start` indicates the starting point.

This program is a 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.

The master (the main program in Figure 4) first deposits the *current* minimal distance together with the corresponding route into the tuple space `RESULT`. This minimal distance is initially the sum over all distances in `DistTable` (an upper limit), and the corresponding route is an empty tuple. Then, the master deposits the initial routes into tuple space `WORK`, and spawns `NumWorker` worker processes in active tuples to compute the search tree in parallel. This number is an argument to the main program. These workers execute in an infinite loop, in which tasks are fetched from tuple space `WORK`, and results are computed and added at tuple space `RESULT`.

After spawning the workers, the master waits until all workers have done their work, and then the master fetches the optimal distance together with the corresponding route from tuple space `RESULT`. Here, Multilisp's future concept is applied to synchronize the master with the workers: the workers are spawned as components of *active* tuples (Hasselbring, 1994b). Since only passive tuples can match a template, the master waits for the termination of the workers, and only then fetches the results. The workers need not terminate in a specific order because each one resolves into the passive tuple `[0]` in tuple space `RESULT`. Tuple spaces are multisets.

Each worker (Figure 5) first checks whether there are more task tuples in tuple space `WORK`, and terminates when there is no more work to do. Then each worker checks whether its partial route (then stored in `MyRoute`) exceeds the length of an already known complete route: then the worker discards this partial route (according to the bounding rule) and continues to fetch another task tuple. If the length of the partial route does not exceed the length of an already known complete route, the

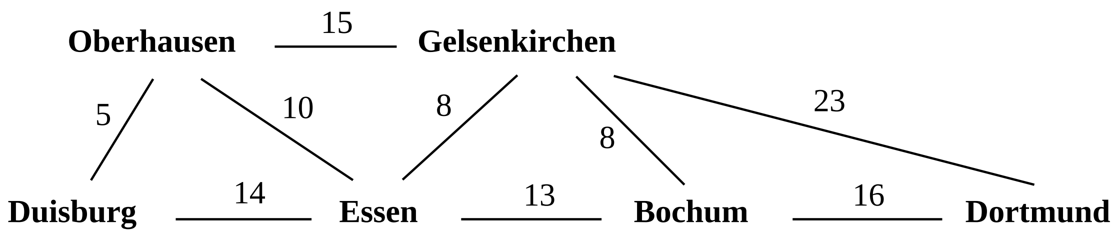

Figure 2. Some cities in the Ruhrgebiet with their connections and distances.

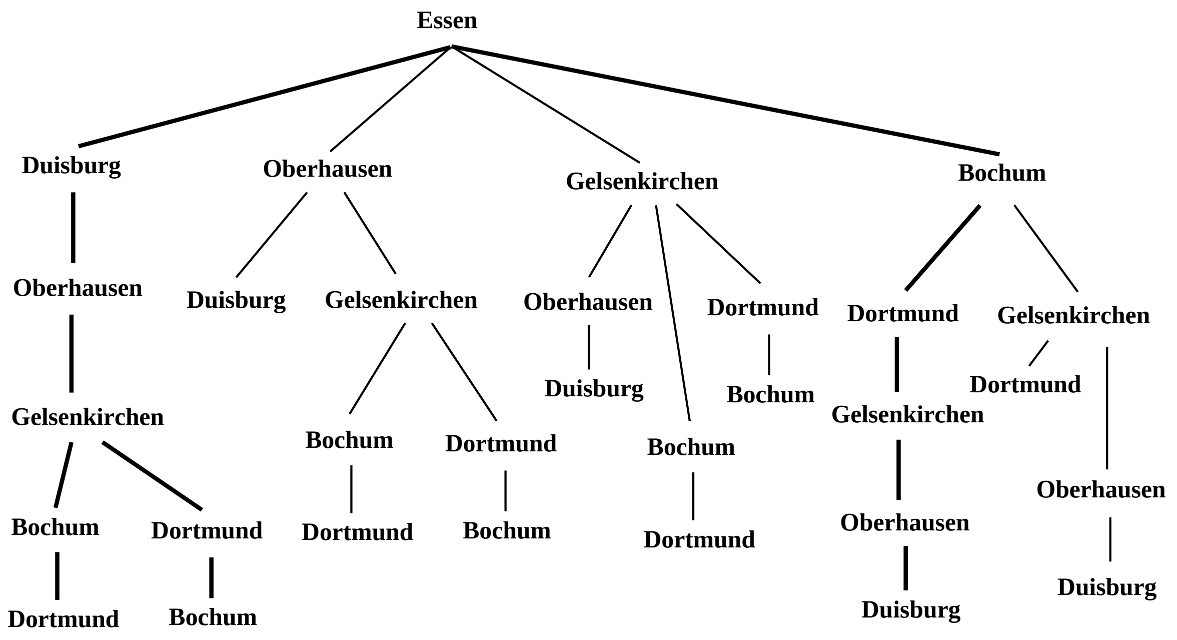

Figure 3. The search tree for our selection of cities in the Ruhrgebiet.

```
program tsp;
  visible constant DistTable :=
           {[["Duisburg","Essen"], 14], [["Essen","Duisburg"], 14],
            [["Duisburg","Oberhausen"], 5], [["Oberhausen","Duisburg"], 5],
            [["Oberhausen","Essen"], 10], [["Essen","Oberhausen"], 10],
            [["Oberhausen","Gelsenkirchen"], 15],
            [["Gelsenkirchen","Oberhausen"], 15],
            [["Essen","Gelsenkirchen"], 8], [["Gelsenkirchen","Essen"], 8],
            [["Essen","Bochum"], 13], [["Bochum","Essen"], 13],
            [["Bochum","Dortmund"], 16], [["Dortmund","Bochum"], 16],
            [["Bochum","Gelsenkirchen"], 8], [["Gelsenkirchen","Bochum"], 8],
            [["Dortmund","Gelsenkirchen"], 23],
            [["Gelsenkirchen","Dortmund"], 23]},
        Nodes := domain (domain DistTable) + range (domain DistTable);
  constant WORK := CreateTS(),     -- For the workers and the work tasks
           RESULT := CreateTS(),  -- For the actual minimal route and distance
           NumWorker := argv(2);  -- Program argument: number of workers
begin
  Start := "Essen";
  -- The minimal distance is initially the sum over all distances:
  Max := +/ [x(2): x in DistTable];
  deposit [ [], Max ] at RESULT end deposit; -- Initialize the result

  -- Deposit the initial routes into tuple space WORK:
  for Entry in DistTable | Entry(1)(1) = Start do
    deposit [ Entry(1), Entry(2)] at WORK end deposit;
  end for;

  for i in [1..NumWorker] do -- Spawn the worker processes in active tuples:
    deposit [ || Worker (WORK, RESULT)] at RESULT end deposit;
  end for;

  for i in [1..NumWorker] do -- Wait for the workers to finish
    fetch ( 0 ) at RESULT end fetch;
  end for;
  fetch ( ? route, ? distance ) at RESULT end fetch; -- The work has been done

  if route = [] then
    put("There exists no route for the traveling salesman!");
  else
    put("Tour de Ruhr = ", route);
    put("Distance = ", distance);
  end if;
end tsp;
```

Figure 4. Solution for the traveling salesman problem: main program as master process. The unary operator `domain` yields the domain of a map (a set of pairs). Accordingly, `range` yields the range of a map. For sets, `+` is the set union. The unary operator `+/` yields the sum over all elements in a compound data structure (a tuple in our example). The function `CreateTS` creates a new tuple space.

```
procedure Worker (MyWORK, MyRESULT);
begin
 loop
  fetch (? MyRoute, ? MyDistance) at MyWORK
   else return 0; -- Terminate and return 0 into the comprising tuple
                  -- (become passive)
  end fetch;

  meet ( ?, ? Distance ) at MyRESULT end meet; -- to check whether we can continue
  if Distance <= MyDistance then
    continue; -- There exists already a shorter or equal long route:
              -- we prune this subtree according to the bounding rule
  end if;

  if #MyRoute >= #Nodes then
    -- We have a complete route. Change the minimum to our route if it is
    -- still the shortest one:
    meet ( ? into MyRoute, ? into MyDistance | $(2) > MyDistance  )
     xor ( ?, ?                              | $(2) <= MyDistance )
      at MyRESULT
    end meet;
  else
    -- Deposit a new task for each route which is a connection of MyRoute
    -- with a node that is not in MyRoute:
    for Entry in DistTable | (Entry(1)(1) = MyRoute(#MyRoute) and
                              Entry(1)(2) notin MyRoute) do
      deposit [MyRoute with Entry(1)(2), MyDistance + Entry(2)]
        at MyWORK
      end deposit;
    end for;
  end if;
 end loop;
end Worker;
```

Figure 5. Solution for the traveling salesman problem: procedure for the worker processes. The unary operator # returns the number of elements in a compound data structure. The binary operator `with` adds an element to a compound data structure.

worker checks whether its partial route is already a complete route. If the partial route is already a complete route, the worker changes the minimal route in tuple space `RESULT` to the given route, provided that the given route is still the shortest one. If the partial route is not a complete route, the worker deposits new task tuples into tuple space `WORK` for each route which is a connection of the given route with a node that is not in the given route. There has to be a connection defined in `DistTable` between the last node in the given route and the next node that is not in the given route to constitute a new extended route.

Figure 6 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. The program in Figures 4 and 5 prints out:

```
Tour de Ruhr = ["Essen", "Duisburg", "Oberhausen", "Gelsenkirchen",
                "Bochum", "Dortmund"]
Distance = 58
```

For simplicity we assume that there exists at least one complete route that visits each of a given set of cities exactly once. If such a complete route does not exist, the program prints the message "`There`

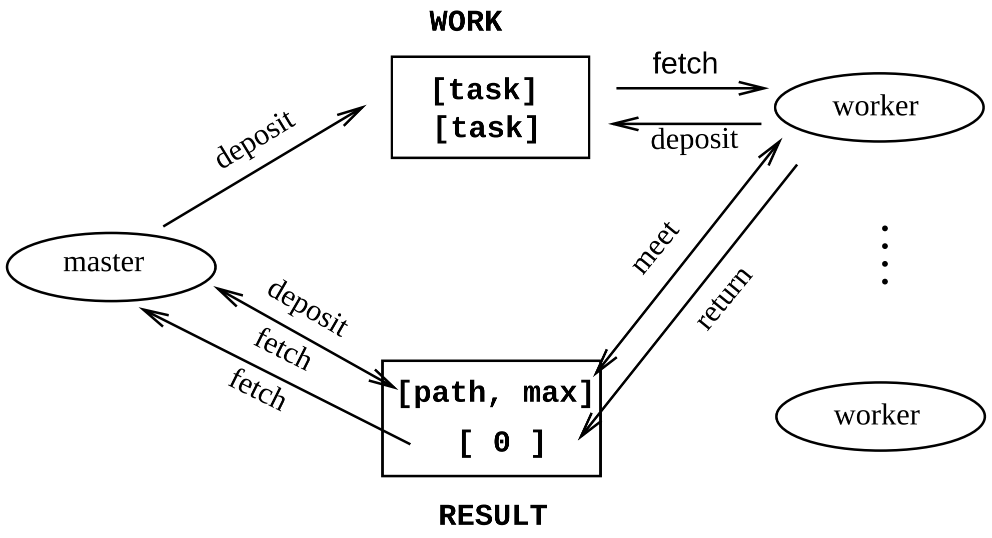

Figure 6. The coarse structure of the master-worker program for the traveling salesman problem.

`exists no route for the traveling salesman!`”. Often it is assumed in solutions for the traveling salesman problem that there exists a connection between each pair of cities. Our program does not have this assumption, and also solves problems where the distances between two cities may depend on the direction.

4 IMPLEMENTATION OF PROSET-LINDA

This section briefly discusses the implementation of ProSet-Linda itself. We implement ProSet-Linda in a somewhat unconventional way: the informal specification is followed by a formal specification, which serves as the basis for a prototype implementation before the production-level implementation is undertaken (Hasselbring, 1994b). Applying formal methods early in the design stage of software systems can increase the designer's productivity by clarifying issues and eliminating errors in the design. A formal development process is more expensive in terms of time and education, but much cheaper in terms of maintenance. There may be bugs, but they are less likely to be at the conceptual level.

The formal specification of the semantics of ProSet-Linda has been defined by means of the formal specification language Object-Z and a prototype for a subset has been implemented from the formal specification with ProSet itself (Hasselbring, 1994a; Hasselbring, 1994b). This prototype allowed immediate validation of the specification by execution. The prototype enabled us to avoid the large time lag between specification of a system and its validation in the traditional model of software production using the life cycle approach (Ghezzi et al., 1991).

In the first C implementation of ProSet-Linda, the SunOS™ Lightweight Processes Library (Sun Microsystems Inc, 1990) is used to implement process creation and synchronization. This Lightweight Processes Library only allows quasi-parallel execution on single processor workstations. The C implementation is transformed from the ProSet prototype implementation. The next implementation was developed under Solaris™ using the Multi-thread Architecture (Powell et al., 1991). This implementation also allows real parallel execution on multi-processor SparcStations™. On these multi-processor SparcStations™ the tuple spaces are stored in shared memory.

The latest implementation has been performed on a local area network (Waltenberg, 1996). On distributed memory architectures, a general problem for implementations of Linda is to provide a map from the virtual shared memory model to physical distributed memory architectures. Therefore, efficient and reliable implementations of Linda on physical distributed memory architectures are in general a great challenge for the implementor. Implementation techniques for physical distributed memory architectures range from those where the tuple space is replicated on each node to those where each tuple resides on exactly one node. The implementation techniques may be classified as follows:

1. Central store with server process
2. Replication of the entire tuple space at each node
3. Distribution of the tuple space over the net with unique copies of each tuple
4. Mixture of these techniques

A central store may very quickly become both a computational and a communicational bottleneck. A distribution of tuple spaces over the nodes in a parallel system in one form or another is the most promising implementation technique on distributed memory architectures for Linda's tuple spaces. This is due to several reasons. First, memory is saved and second, the overhead for guaranteeing the consistency of the replicated tuple spaces is absent. Furthermore, any Linda implementation that can scale to large machines *must* distribute tuple space, so as to avoid node contention. This distributes the cost of handling tuple operations across all nodes in the system. The remaining problem is *how* to distribute the tuple space. Multiple tuple spaces, as they are supported in ProSet-Linda, provide a direct approach for distributing the tuple spaces on a distributed memory architecture. A distributed hashing mechanism has been implemented for ProSet-Linda's tuple spaces (Waltenberg, 1996). This new implementation allows improved performance predictions compared to the former implementations.

5 RELATED WORK

Some approaches to prototyping parallel algorithms with very high-level parallel programming languages intend to alleviate the development of parallel algorithms in quite different ways. Because of the problems with the low-level programming models for message passing, many models which emphasize some kind of *shared data* have been developed that intend to deliver a higher level of abstraction to alleviate parallel programming. These high-level programming models appear to be good candidates for prototyping parallel algorithms. Such high-level languages are set-based data-parallel approaches and specific extensions to logical, functional, and object-oriented languages as well as coordination languages which extend sequential languages. Examples are:

- Proteus (Goldberg et al., 1994) is a data-parallel variation of SETL (Kruchten et al., 1984) that supports control and data parallelism for prototyping parallel algorithms. A semi-automatic refinement system has been developed for the Proteus language which 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.
- The Crystal (Chen et al., 1991) 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.

- PROLOG programs express two distinct forms of implicit parallelism: AND-parallelism is the simultaneous reduction of several different subgoals in a goal and OR-parallelism is the simultaneous evaluation of several clauses for the same goal. Parallel variations of logic programming languages appear to be candidates for prototyping parallel algorithms (Huntbach and Ringwood, 1995).
- RAPIDE (Luckham et al., 1993) is a parallel object-oriented language specifically designed for prototyping parallel systems that combines the partially ordered event set (poset) computation model with an object-oriented type system for the sequential components.
- With composition and coordination languages, parallel systems are described in terms of processes that comprise a system together with the communication and control interconnections between these processes. A *coordination language* provides means for process creation and inter-process communication which may be combined with sequential *computation languages* to constitute a *parallel programming language* (Carriero and Gelernter, 1992). Compositional C++ (Chandy and Kesselman, 1993) is such a language designed for prototyping. ProSet-Linda belongs to this category, too.

6 SUMMARY

To build a parallel system, you should start with executable prototypes to validate the requirements and study the feasibility (neglect the execution performance in the first instance). Powerful tools are needed to make prototyping of parallel algorithms and systems feasible. Our goal is to make parallel program design easier through prototyping parallel algorithms. The high level of ProSet-Linda's constructs for parallel programming enables us to rapidly develop prototypes of parallel programs and to experiment with parallel algorithms.

The idea of prototyping is being adopted in software engineering for different purposes: prototypes are used *exploratively* to arrive at a feasible specification, *experimentally* to check different approaches, and *evolutionary* to build a system incrementally. The order of development steps in the traditional life cycle model is mapped here into successive development cycles. Note that a prototype is a model, and that this model taken as a program has to be executable so that at least part of the functionality of the desired end product may be demonstrated on a computer. Prototyping has been developed as an answer to deficiencies in the traditional life cycle model, the waterfall model (Ghezzi et al., 1991), where each phase is completed before the next phase is started; but it should not be considered as an alternative to this model. It is rather optimally useful when it complements the life cycle model. It is plausible that prototyping may be used during the early phases of software development. In (Hasselbring and Kröber, 1998), the combination of our prototyping approach with an object-oriented modeling method is discussed.

This paper presents the prototyping language ProSet-Linda with some introductory examples. Applications experience with parallelizing high-level computer vision algorithms (Hasselbring and Fisher, 1995), cooperative planning of independent agents (Doberkat et al., 1996) and the requirements analysis for a hospital communication server (Hasselbring and Kröber, 1998) are discussed elsewhere.

REFERENCES

Carriero, N., and Gelernter, D., *How to write parallel programs*, The MIT Press, Cambridge, Massachusetts, 1990.

Carriero, N., and Gelernter, D., Coordination languages and their significance, *Communications of the ACM* 35, 96–107 (1992).

Chandy, K., and Kesselman, C., CC++: A declarative concurrent object-oriented programming notation, in *Research Directions in Concurrent Object-Oriented Programming* (G. Agha, P. Wegner, and A. Yonezawa, eds.), The MIT Press, Cambridge, Massachusetts, 1993, pp. 281–313.

Chen, M., Choo, Y., and Li, J., Crystal: Theory and pragmatics of generating efficient parallel code, in *Parallel Functional Languages and Compilers* (B. Szymanski, ed.), ACM Press, New York, N.Y., 1991, pp. 255–308.

Dijkstra, E., Hierarchical ordering of sequential processes, *Acta Informatica* 1, 115–138 (1971).

Doberkat, E.-E., Franke, W., Gutenbeil, U., Hasselbring, W., Lammers, U., and Pahl, C., ProSet — A Language for Prototyping with Sets, in *Proc. Third International Workshop on Rapid System Prototyping* (N. Kanopoulos, ed.), IEEE Computer Society Press, Piscataway, N.J., 1992, pp. 235–248.

Doberkat, E.-E., Hasselbring, W., and Pahl, C., Investigating strategies for cooperative planning of independent agents through prototype evaluation, in *Proc. First International Conference on Coordination Languages and Models (COORDINATION '96)* (P. Ciancarini and C. Hankin, eds.), volume 1061 of *Lecture Notes in Computer Science*, Springer-Verlag, Berlin, 1996, pp. 416–419.

Gelernter, D., Generative communication in Linda, *ACM Transactions on Programming Languages and Systems* 7, 80–112 (1985).

Ghezzi, C., Jazayeri, M., and Mandrioli, D., *Fundamentals of Software Engineering*, Prentice-Hall, Englewood Cliffs, N.J., 1991.

Goldberg, A., Mills, P., Nyland, L., Prins, J., Reif, J., and Riely, J., Specification and development of parallel algorithms with the Proteus system, in *DIMACS: Specification of Parallel Algorithms* (G. Blelloch, K. Chandy, and S. Jagannathan, eds.), AMS Press, 1994.

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

Hasselbring, W., Animation of Object-Z specifications with a set-oriented prototyping language, in *Z User Workshop (Proc. Eighth Z User Meeting)* (J. Bowen and J. Hall, eds.), Workshops in Computing, Springer-Verlag, London, 1994, pp. 337–356.

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

Hasselbring, W., and Fisher, R., Using the PROSET-Linda Prototyping Language for Investigating MIMD Algorithms for Model Matching in 3-D Computer Vision, in *Parallel Algorithms for Irregularly Structured Problems* (A. Ferreira and J. Rolim, eds.), volume 980 of *Lecture Notes in Computer Science*, Springer-Verlag, Berlin, 1995, pp. 301–315.

Hasselbring, W., and Kröber, A., Combining OMT with a prototyping approach, *The Journal of Systems and Software*, to appear (1998).

Huntbach, M., and Ringwood, G., Programming in concurrent logic languages, *IEEE Software* 12, 71–82 (1995).

Kruchten, P., Schonberg, E., and Schwartz, J., Software prototyping using the SETL programming language, *IEEE Software* 1, 66–75 (1984).

Lawler, E., and Wood, D., Branch-and-bound methods: a survey, *Operations Research* 14, 699–719 (1966).

Luckham, D., Vera, J., Bryan, D., Augustin, L., and Belz, F., Partial orderings of event sets and their application to prototyping concurrent timed systems, *The Journal of Systems and Software* 21, 253–265 (1993).

Powell, M., Kleinman, S., Barton, S., Shah, D., Stein, D., and Weeks, M., SunOS Multi-thread Architecture, in *Proc. USENIX Winter '91 Technical Conference*, 1991.

Sun Microsystems Inc, *Programming Utilities & Libraries*, System Manuals, 1990.

Waltenberg, K., Design and implementation of a distributed runtime system for ProSet-Linda (in German), Master's thesis, Department of Computer Science, University of Dortmund, 1996.