M3Set - A Language for Handling of Distributed and Persistent Sets of Objects

László Böszörményi and Karl-Heinz Eder
Institut für Informatik, Universität Klagenfurt
Universitätsstr. 65 -67, A - 9020 Klagenfurt
e-mail: {laszlo, charly}@ifi.uni-klu.ac.at      http://www.ifi.uni-klu.ac.at/staff/stafftab.html

Keywords: Distributed Object Management, Parallel/Distributed Languages
Parallel/Distributed Databases, Object-Oriented Languages, Modula-3

Abstract

We claim that distributed object-oriented systems must provide a higher level of abstraction to their users, than usually provided. Especially, it is necessary to provide application-oriented, intelligent aggregates of objects with transparent distribution of their elements. Beside that, it seems to be not only reasonable, but also relatively easy to connect persistence with distribution. A system, offering distributed and persistent polymorphic sets of objects, on the level of a clean, type safe programming language is introduced. The user of such a system gets distribution and persistence in the same “natural” way, as users of traditional systems get volatile arrays of numbers or classes of objects.

1 Introduction

The basic problem of object-based distributed systems is that the abstraction level of discussion is too low.

We speak generally about “object distribution”, which is not really relevant for most applications. Most applications are not interested on having arbitrarily distributed objects, rather on aggregates of objects that are themselves distributed.

Maybe the most important difference between a desk calculator of the forties and a modern (von Neumann or Zuse) computer is the capability of the latter to process aggregates of data (actually arrays of numbers), instead of processing data individually. In a similar way, the most important difference between a computer network and a true distributed system could be seen (beyond all the usual transparency requirements) in the capability of the latter to process aggregates of distributed objects in a transparent way. Finding the proper aggregate is the only way to free the application programmer from struggling with the location of individual objects. Rather, he/she gets a collection of objects, which is automatically distributed corresponding to the semantics of the application on the one side and to the underlying architecture on the other side.

Claiming pre-defined distributed collections instead of freely distributed objects imposes surely a kind of restriction. But restriction is often exactly the key for making new technology available
to a broad user community. Such an example is structured programming, restricting flexibility in programming. A maybe still better example is SQL, restricting even the power of the access to data - the key for its unrestricted success.

In the area of classical parallel processing (scientific computing) this has always been known. Scientific programming concentrates on parallel arrays of numbers [22, 16]. This approach has its great advantages in semi-automatic parallelization of programs processing large matrices, it has, however, its limits as well, and can surely not serve as a basis for distributed object systems.

We propose for such an aggregate the concept of distributed, persistent and polymorphic sets of objects. This kind of aggregate has the following advantages:

1. Sets are well-understood, relying on sound mathematical theory.
2. Sets are per definition unordered collections, therefore, no assumption must be made on the order of processing its elements. From this follows that they can be processed in parallel most naturally (they are indeed inherently parallel data structures).
3. Data are retrieved from a set based on their values (instead of computed positions), which is the usual view of most applications coping with a large amount of data (e.g. of most database query languages).
4. There are a number of reasons to combine distribution and persistence, being related in a most inherent way: Distribution is an extension of scope in space, persistence is the same in time. Most applications, requiring distribution of objects, also need persistence of the same. Moreover, some technical problems are very similar. For example, objects need an identification which is independent from both space and time. A framework that considers both of these aspects can provide its user a view on object aggregates, which is transparent both to location and duration (i.e. the user neither have to struggle with locating his/her objects nor with saving and reloading them).
5. Polymorphic sets are collections of objects of a given base type or of any subtype of it. With the help of polymorphic sets we can easily build hierarchical subclasses on the extensional level (see [3]).

Based on these considerations we have designed and implemented (some parts are still under development) a system that fulfills the above requirements [2, 10, 3]. It supports the concept of distributed and persistent sets on two basic levels: on the language (M3Set) and on the run-time level (PPOST - a Parallel, Persistent Object STore). PPOST can serve as the basis for a main memory based database system [1, 11].

In the following we concentrate on the language. Query optimization and parallelization is described in a recent paper [10]. Other important aspects, such as the architecture of the PPOST system, some questions of persistence and transactions are discussed in [2, 3]. Further important questions such as efficient automatic index- and distribution management are subjects for further studies.

In a recent paper [9] DeWitt et al. introduce a similar approach, called ParSet, based on the SHORE OODBMS and Parallel Sets of Kilian [13]. They show with the help of the OO7 benchmarks that the approach results in excellent traversals. They offer a C++ library, but no direct language support yet (it is planned).
2 M3Set

There is an obvious need to have access to a large amount of persistent data and to process them in parallel whenever possible. Nevertheless, these aspects are generally handled separately. Moreover, persistent and parallel systems are rather heavy weight. Our aim is to provide these features together, in a cost-effective way, in form of a middle-weight system. We introduce the language M3Set - realized as an extension of a clean, object-oriented language, Modula-3 [14, 4] - that considers parallelism and persistence in one, simple conceptual framework, based on sets. Parallelism and persistence become normal, first-class citizens of the programmer’s model.

M3Set has the full power of a general-purpose programming language and incorporates additional features to express declarative queries. The extension of Modula-3 is kept as small and as simple as possible. It is the free choice of the programmer to take advantage of the highly optimized and parallelized query expressions or to use explicit, unoptimized iterations. This leads on the one side to a simple and unified programmer’s model, on the other side to a relatively simple optimizer. Type checking is an inherent part of the language, recognition of common subexpressions can be done statically after type checking. Cost based optimization and parallelization can be done in one step at run-time.

Optimization is based on a very powerful generalized select-expression (allowing all kind of usual queries). As it is an expression, it is rather declarative and can be therefore easily optimized and parallelized - apart from side-effects (see [10]).

The language allows very different implementations. Any combination of persistent and volatile, parallel and sequential, optimized and unoptimized implementations is possible. This is eased by the fact that a great part of the implementation is located outside the compiler with the help of predefined interfaces.

The set operators of M3Set are comparable to already proofed ones [13, 5, 17]. We use these operators also as a basis for optimization and parallelization. The OPAL language of GemStone [7] offers only classes but not types. DBPL [18] - an extension of Modula-2 [21] - is strongly typed, its data model is relational. Parallelization is not considered in the DBPL environment. FAD [19, 20] is object-oriented, but does not support polymorphism. Its compiler optimizes and parallelizes queries written in an expression based language. In [8] parallelism is based on sets of tuples for relational operators and parallelism is hidden from applications. We hide parallelism based on sets of objects. Our data model is object-oriented [6, 15] and the query language is declarative.

2.1 Language definition

Modula-3 [14] is one of the newest members of the Pascal family. It supports the traditional structured programming concepts in an unusually clean way. It is type-safe without compromise. Moreover, it is object-oriented, supports threads, exceptions, multiple interfaces and garbage collection. It provides also sets, restricted to sets of ordinal types (as in Pascal).
The syntax of Modula-3 sets remains actually unchanged.\footnote{The syntax of Modula-3 sets remains actually unchanged.}

The predefined Modula-3 types \texttt{NULL}, \texttt{REFANY}, \texttt{ROOT} and the value \texttt{NIL} remain unchanged.\footnote{The predefined Modula-3 types \texttt{NULL}, \texttt{REFANY}, \texttt{ROOT} and the value \texttt{NIL} remain unchanged.}

The set type

M3Set relaxes the above restriction to sets and introduces the \textit{object-set} type (object-sets will be called simply sets, except we want to emphasize the difference), with the following syntax\footnote{The syntax of Modula-3 sets remains actually unchanged.}:

\[ \text{Objectset} = \text{SET OF Basetype} \]

The value of an object-set is either NIL, or a reference; the relation \texttt{NULL} \texttt{::=} \texttt{Objectset} \texttt{::=} \texttt{REFANY} holds for any object-set type, where \texttt{::=} is the reflexive and transitive Modula-3 operator for subtyping, \texttt{NULL} is a predefined type containing only the value \texttt{NIL}, and \texttt{REFANY} is the predefined supertype of all (traced) references\footnote{The predefined Modula-3 types \texttt{NULL}, \texttt{REFANY}, \texttt{ROOT} and the value \texttt{NIL} remain unchanged.}. For \texttt{Basetype} must hold: \texttt{Basetype} \texttt{::=} \texttt{Obj.T} \texttt{::=} \texttt{ROOT}, where \texttt{ROOT} is the predefined supertype of all objects and \texttt{Obj} is an in M3Set predefined interface, exporting the type \texttt{T}, the supertype of all objects, which are allowed to be a member of an object-set. All \texttt{Obj.T}-objects have a \textit{unique identifier}, and a \textit{Hash} and an \textit{Equal} operator, exported as procedures of the \texttt{Obj} interface (these can be used e.g. by the run-time system to check that an object can occur only once in a set). The identity of an object is independent from space and time, i.e. it identifies an object in a distributed and persistent environment unambiguously. Object-sets are also polymorphic, i.e. they can contain any object that is a subtype of the base type. An element can occur only once in a set.

The subtype relation is also defined for object-sets. If the base type of object-set \texttt{os1} is a subtype of the base type of object-set \texttt{os2} then \texttt{os1} \texttt{::=} \texttt{os2} also holds. This implies that all instances of \texttt{os1} are also instances of \texttt{os2}. For example a student may be a subtype of a person. Then a set of students is a subtype of set of persons. All sets of students are also sets of persons, therefore the former ones can always substitute the latter ones.

As object-sets are references, they must be initiated dynamically, with the built-in Modula-3 function \texttt{NEW}. The syntax of the call of \texttt{NEW} is the same as for dynamic arrays, requiring the type of the dynamic data as the first parameter and its size in the second one. In case of object-sets, the size parameter is optional and its semantics has changed: it is just an estimation of the set size. Consider the examples on Figure \ref{fig:example}.

\begin{verbatim}
TYPE
  Person = Obj.T OBJECT name: TEXT METHODS display():= D END; (*Person::=:Obj.T*)
  Student = Person OBJECT matrNr: CARDINAL END; (*Student::=:Person*)
  PersSet = SET OF Person;
  StudSet = SET OF Student;
VAR
  persons := NEW(PersSet, 10000); (*1. parameter: type, 2. parameter: set-size estimation*)
  students := NEW(StudSet, 1000);

Figure 1. Examples for Set Declarations
\end{verbatim}
Set constructors

A set value can be defined by a set constructor, with the following general form:

Objectset{Objectlist}

Objectlist contains a list of objects. For example the following set constructor defines a set of persons, containing two persons:

PersSet{NEW(Person, name:= “Paul”), NEW(Person, name:= “Peter”)}

Assignment statement

The assignment has reference semantics. After the assignment persons:= students, both persons and students point to the same set. For type compatibility the usual rules of Modula-3 hold [14]. The above assignment is always legal, because students <: persons. After the assignment the variable person (of static type PersSet) has the dynamic type StudSet. The opposite assignment students:= persons is legal only, if the dynamic type of persons is a subtype of StudSet. Otherwise, a so-called narrow-failure is generated at run-time.

Set-inclusion and -exclusion

An object can be included into a set through the set-inclusion statement. Similarly, it can be excluded by the set-exclusion statement. The general form of inclusion and exclusion:

set += object                                          (* object is included into set*)
set -= object                                            (* object is excluded from set*)

Expressions

The set-expressions union, intersection, difference and symmetric difference are defined as usual, with unchanged syntax. Operations between sets in a subtype relation are allowed. The type of the result for set union and symmetric difference is the more general one, for intersection the more specific one of the type of the operands. For set difference the type of the result is that of the first operand. For example:

persons:= persons + PersSet{NEW(Person, name:= “Mary”)};
persons:= persons + students;

In the first statement a new object is included implicitly into the set persons. The second statement creates a union of persons and students. The type of the result is PersonSet.
Selection

The central operation in M3Set is the select-expression. It allows to express actually all kinds of usual queries. It takes a number of sets as input and produces a new set value. The general form is:

Objectset \{ \text{object} \mid \text{id}_1 \leftarrow \text{set}_1 \ [\ , \ \text{id}_2 \leftarrow \text{set}_2 \ [\ , \ \ldots \ , \ \text{id}_n \leftarrow \text{set}_n] \ : \ : \ \text{condition} \}

The type of the result is \text{Objectset}. It contains those objects (\text{object}), that satisfy the \text{condition}, which must be of Boolean type. The type of \text{object} must be a subtype of the base type of \text{Objectset}. Each \text{id}_i represents an element of the corresponding input set \text{set}_i (\leftarrow \text{stays for } \epsilon, \ n \text{ is currently limited by the implementation to 2}). The scope of these identifiers is the select expression, their type is the base type of the corresponding set. All identifiers must be different, while the same set may occur more than once (e.g. for self-join). The condition may principally contain side-effects, because it may contain procedure- or method-calls changing the state-space involved. It is not prohibited to have such conditions, but select-expressions containing such conditions are not considered by the optimization and parallelization (the discussion of side-effects see in [10]). The result of a selection is always an object-set. Consequently, selections can be nested at will, any \text{set}, may be itself a selection (and any other set expression).

An actual implementation may restrict \text{object} to be either one of the variables \text{id}_i or a call of a globally declared function. Similarly, \text{condition} maybe restricted to a call of a globally declared function. Such a restriction may ease the implementation of these constructs in a distributed environment, as we don't have to care with the distribution of nested contexts. Figure 3. shows the usage of selections, based on the declarations of Figure 2.

```
TYPE
Address = Obj.T OBJECT street: TEXT; nr: CARDINAL END; (*Address <: Obj.T*)
Person = Obj.T OBJECT name: TEXT METHODS addr(): Address END; (*Person <: Obj.T*)
Student = Person OBJECT matrNr: CARDINAL END; /*Student <: Person*/
Pair = Obj.T OBJECT p1, p2: Person END; /*Pair <: Obj.T*/
PersSet = SET OF Person; /*Set of persons*/
StudSet = SET OF Student; /*Set of students*/
PairSet = SET OF Pair; /*Set of pairs of persons*/

PROCEDURE P(par1, par2:Person): Pair = /*Returns a new, initialized pair of persons*/
BEGIN
  RETURN NEW(Pair, p1:= par1, p2:= par2)
END P;

VAR
persons := NEW(PersSet); /*set of persons*/
stud1 := NEW(StudSet); /*set of some students*/
stud2 := NEW(StudSet); /*set of some other students*/

..fill persons, stud1 and stud2 with objects, keep the latter two distinct ...
```

Figure 2. Declarations of Set Types and Variables
Expression (1) simply generates a copy of the set \( \text{persons} \), expression (2) selects all persons without any valid address. The semi-join (3) selects all students of the set \( \text{stud1} \) that have the same address as any of the students from \( \text{stud2} \). The result is a set of type \( \text{StudSet} \), containing already existing student-objects. The join-expression (4) creates all pairs of students from the two input sets, having the same address. As \( \text{Student} <: \text{Person} \) holds, the fields \( p1 \) and \( p2 \) in \( \text{Pair} \) can be initialized by student-objects. The type of the result is \( \text{PairSet} \).

**Cardinality**

The semantics of the built-in Modula-3 function \( \text{NUMBER} \) is extended that it returns the actual number of the elements of a set.

**Universal and existential quantifiers**

\[
\begin{align*}
\text{ALL} & \quad \text{object} <- \text{set} :: \text{condition} \\
\text{ANY} & \quad \text{object} <- \text{set} :: \text{condition}
\end{align*}
\]

Both quantifiers ought to be principally of Boolean type. However, most existential questions are not of the form: "does exist any object that ... ?", but rather of the from "if there exists any object \( x \) that ... then take it and do this and this.". Therefore, the quantifiers return either an object (for the \( \text{condition} \) being true) or NIL (for false) as a result. ANY checks whether any of the objects in the set satisfies the condition. If yes, it picks one of them, otherwise it returns NIL. ALL checks whether all objects in the set satisfy the condition. If yes, it picks one of them, otherwise it returns NIL. For the empty set both quantifiers return false. For example, to display any student with a legal matriculation number, we write:

\[
\begin{align*}
\text{VAR student: Student := ANY st <- students :: st.matrNr # 0;} & \quad (*\text{contains NIL or a student*}) \\
\text{... IF student # NIL THEN student.display() END; ...} & \quad (*\text{prints student if legal*})
\end{align*}
\]

Note that (ANY \( st <- \text{students :: st.matrNr # 0} \) # NIL is a shorthand for \( \text{NUMBER(StudSet\{st [ st <- \text{students :: st.matrNr # 0}\}}) # 0 \). A similar form for ALL also can be given. However, ANY and ALL are not only shorter but generally also faster, because they evaluate only as long as necessary. The following statement inserts the object \( \text{stud} \) in the set \( \text{students} \), if and only if the matriculation number of \( \text{stud} \) is unique:

\[
\text{IF (ANY st <- students :: stud.matrNr = st.matrNr) = NIL THEN students += stud END}
\]
Membership

The syntax of membership is the very same as for basic Modula-3 sets: `element IN set`. The result is of type Boolean. The expression `student IN students` is true, if object `student` is contained in the set `students`, otherwise false.

Foreach Statement

There is an additional loop statement to Modula-3 in M3Set, the FOREACH loop, which applies a statement sequence on each member of a set, with the following general form:

FOREACH e <- set [SORTEDBY Filter] DO Statementsequence END

The FOREACH statement is executed as follows. First, the set-expression (`set`) is evaluated. It may be any right-hand or left-hand expression. In the first case a temporary (nameless) set is created anyway, thus the statement iterates on this extension. In the second case (the set expression evaluates to a writable designator, e.g. a variable) the statement iterates on the original extension. In any case, the statement sequence must not change the extension, i.e. it must not insert or remove any element into or from the set. If it does, a predefined exception is generated at run-time. After evaluating `set`, the statements of the statement-sequence are applied on each `e ∈ set`. The scope of `e` is the FOREACH statement itself.

The parallel execution of a FOREACH statement is still subject of further study. At the time being it is fully in the responsibility of the programmer to start Modula-3 threads for each partition of a distributed set (see in section Distribution, how to access individual partitions of a distributed set). The addition of a remote fork and remote join operation is also subject of further study. They can ease the parallel execution of aggregate functions, such as the addition of a certain attribute value of all elements of an object set.

Filter is a function specifying an order on the elements of a set, with the following predefined signature (filters must be marked by the `<FILTER *> pragma`):

```
Name(e1, e2: Basetype): [-1 .. 1]             (*-1 for e1 < e2, 0 for e1 = e2, 1 for e1 > e2*)
```

`Basetype <: Obj.T` must hold. If SORTEDBY is given then the iteration will be executed corresponding to the filter function. The optimizer is supposed to recognize if an index is available to the filter and take use of it when available. Figure 4. shows an example of giving out a list of students, ordered by the matriculation numbers.

---

3 Pragmas are not elements of the language, rather directives to the compiler.
PROCEDURE FiltMNr(st1, st2: Student): [-1 .. 1] = (*defines order among students*)
BEGIN
WITH m1 = st1.matrNr, m2 = st2.matrNr DO
  IF m1 < m2 THEN RETURN -1 ELSIF m1 = m2 THEN RETURN 0 ELSE RETURN 1 END
END (*WITH m1, m2*)
END FiltMNr;
...
FOREACH st <- students SORTEDBY FiltMNr DO st.display() END

Figure 4. Iteration with the Foreach Statement

2.2 Distribution

To express distribution we use a concept of set partitions. A set can be partitioned into a number of disjunct subsets, whose union equals to the original set. Set partitions may be mapped onto computing nodes. A computing node has an own address space, and is able to execute statements in parallel (truly or virtually) to other nodes. Methods of objects located in different nodes can be executed in parallel.

Distribution is expressed with the help of the ON-clause. Beside the set of nodes we also may specify the distribution strategy, which is controlled by user-supplied functions:

ON Partitions [HASH Map | RANGE Map, Filter]

Partitions must be an ordinal type, typically a subrange of CARDINAL, i.e. a range of logical numbers. The actual mapping of the logical numbers onto physical machine addresses are hidden by the implementation. Map and Filter are user-defined functions with a predefined signature (they must be marked with the corresponding pragmas, <*MAP*> resp. <*FILTER*>). If neither HASH nor RANGE is specified in the ON-clause, then round-robin distribution is used (for a discussion of distribution strategies see [8]). In the case of hashed or range distribution the user-defined map function is called on every element to be inserted, to decide which node is to be chosen. Note that the map function must be idempotent, it must return for the same object always the same partition number. Otherwise, the system may fail to retrieve an object.

There remains an important question: How and where should we apply the ON-clause? Should we connect distribution on the set type? This had the advantage that distribution were known at compilation time and the compiler may prepare partitioning easily. The main disadvantage were in this case that all instances of a given type ought to have the same distribution. Therefore, we prefer to place the ON-clause to the instantiation of object-set variables. This has, however, a disadvantage as well. It may easily lead to contradictions if we want to map partitions on physical nodes (what we certainly do). An object may be member of several sets at the same time. Let’s suppose for example that students and employee are sets of persons, located on two disjunct sets of nodes. If we insert an object in both sets (e.g. a student with a special contract) then this object cannot be located in both sets physically. In [9] the concept of primary and secondary sets are suggested to solve this problem. An object is physically contained in its primary set. It may be contained in many secondary sets, where a reference to the instance in the primary set is stored.
We adapt this idea saying that the primary set of an object is the distributed set where it is included \textit{first}. When an object is created, it is located on the node, where it has been created. It has no primary set, until inserted explicitly into a distributed set, which thus becomes its primary set. On the inclusion into its primary set, the object maybe relocated, corresponding to the specified distribution strategy. Later, the object maybe included into several persistent or temporary sets - all serving as secondary sets and containing only references to the object’s data. The location of an object remains determined by its primary set, even if the primary set is deleted. If an object is excluded from its primary set then it looses its primary set (it still remains at the same location). At the next inclusion into a distributed set, this set becomes its new primary set, and the object maybe relocated.

In Figure 5, \textit{stud} is distributed over 3 nodes in a round-robin manner. The first object will be inserted into node _1_, the second one into node _2_ and so on. The set \textit{emp} is also distributed over 3 nodes. “Big bosses” are inserted on node _1_, “small bosses” on node _2_, all others on node _3_.

\begin{verbatim}
TYPE
  Processors = [1 .. 3]; (*The actual pool of processors contains node_1, node_2, and node_3*)
  Student = Person OBJECT matrNr: CARDINAL END; (*Student is subtype of Person*)
  Position = {BigBoss, SmallBoss, NoBoss}; (*"Standard" Modula-3 enumeration*)
  Employee = Person OBJECT position: Position; salary: REAL END; (*Employee is subtype of Person*)
  StudSet = SET OF Student; (*Set of students*)
  EmpSet = SET OF Employee; (*Set of employee*)
<* MAP *> PROCEDURE Map(emp: Employee): Processors =
  (*Returns for every employee a processor number in the pool.*)
  BEGIN
    CASE emp.position OF
    | Position.BigBoss => RETURN 1;
    | Position.SmallBoss => RETURN 2;
    | Position.NoBoss => RETURN 3;
    END; (*CASE*)
  END Map;

  VAR
    stud:= NEW(StudSet, ON Processors); (*stud has round-robin distribution*)
    emp:= NEW(EmpSet, ON Processors HASH Map); (*emp has hash distribution*)
  BEGIN
    FOR i:= 1 TO 100 DO stud += NEW(Student, matrNr:= i) END;
    (*students are distributed in a round-robin fashion*)
    FOR i:= 1 TO 200 DO emp += NEW(Employee, position:= Position.NoBoss) END;
    (*These 200 employee all come on node_3*)
  END;

Figure 5. Declaration of Distributed Sets of Objects
\end{verbatim}

Individual set-partitions can be accessed by a special indexing, using the index-brackets [* and *]. For example, \textit{students[*i*]} refers to the partition, of the set \textit{students}. The value of the index must be a member of the type describing the actual partitioning, otherwise a compilation or runtime error is generated. Explicit indexing of partitions can be used for “manual” parallelization. Let’s suppose we want to add the salaries of all employee in the set \textit{emp}. Of course, we would like
to make the additions on all nodes in parallel. Unfortunately, the computation of such an aggregate function cannot be expressed with the select expression (which always returns a set and never a scalar value). We may, however, declare a procedure that is forked for all partitions and computes the sum for all elements inside a partition. The application (the main thread) must join at the end of the computation the individual threads, gather the partial results and build the final sum. The example in Figure 6. (relying on the declarations of Figure 5.) shows the major parts of such a "manually" parallelized program. We import the module RemoteThread, which provides remote forking. For parameter passing it uses closure objects, a technique often used in the Modula-3 environment [14, 4].

```
FROM RemoteThread IMPORT T, Closure, Fork, Join;                  (*Implements remote fork and join*)

TYPE
  Cl = Closure OBJECT
      sum: REAL := 0.0;                                             (*stores the sum for a given partition*)
      part: EmpSet;                                        (*represents the data located in a given partition*)
  OVERRIDES
      apply:= StartPart                                           (*threads execute the actual code of apply*)
  END; (*Cl*)

PROCEDURE SumPart(partition: EmpSet): REAL =                        (*computes the sum for partition*)
  VAR sum: REAL := 0.0;
  BEGIN
    FOREACH elem <- partition DO sum:= sum + elem.salary END;
  RETURN sum
  END SumPart;

PROCEDURE StartPart(cl: Cl): REFANY =  (*encapsulates SumPart with a signature expected by Thread*)
  BEGIN
    cl.sum:= SumPart(cl.part); RETURN cl
  END StartPart;

PROCEDURE Start() =                                                      (*starts the threads for each partition*)
  VAR
    cls: ARRAY Processors OF Cl;  (*closure for each partition*)
    threads: ARRAY Processors OF T; (*thread identifier for each partition*)
    sum: REAL := 0.0;                                             (*stores the sum of the salaries of all employee*)
  BEGIN
    FOR proc:= FIRST(Processors) TO LAST(Processors) DO
      cls[proc]:= NEW(Cl, part:= emp[*proc*]);                               (*create closure for each partition*)
      threads[proc]:= Fork(cls[proc]);                                               (*start thread for each partition*)
    END; (*FOR proc*)
    (* ... join the threads e.g. with a barrier ... *)
    FOR proc:= FIRST(Processors) TO LAST(Processors) DO
      sum:= sum + cls[proc].sum                                                              (*compute the final sum*)
    END; (*FOR proc*)
  END Start;
```

Figure 6. Explicit Parallelization of an Iteration
2.3 Optimization and Parallelization

The reader might have noticed that there is a given redundancy in the language proposal. The select-expression can be regarded as entirely redundant, as all select-operations can be expressed with the help of loops. The two statements in Figure 7. are semantically equivalent (let’s suppose result is declared as a set of persons).

result:= PersSet{ p | p <- persons :: p.addr() # NIL };
FOREACH p <- persons DO
   IF p.addr() # NIL THEN result+= p END
END (*FOREACH p*)

Figure 7. Implicit and Explicit Iteration over a Set

The difference is that in the select-expression the loop is implicit, expressed in a declarative style, while in the FOREACH statement it is explicit. It is obvious (and well-known) that the first form is much easier to optimize and parallelize. Therefore, select-expressions will be extensively optimized and parallelized, in contrast to FOREACH statements, which will be compiled in a simple, straight-forward manner.

The optimizer gets all the necessary information at run-time, to transform a precompiled sequence of operations in order to achieve optimal (or at least good) performance. The optimizer knows the actual size of the involved sets, it knows the selection cardinalities of indexes and it knows the kinds of distribution. It selects the most efficient algorithm as a function of costs, sizes and distribution. It uses the selection cardinalities to execute the most restrictive condition at first. It can handle a cache of optimized expressions, similarly to the Volcano system [12]. Some details of parallelization and index-based optimization are discussed in [10].

2.4 Persistence

Persistence is bounded to object-set variables. A persistent set is declared in the form:

PERSISTENT Name VAR set: Objectset

The declaration must appear in an interface or in the global scope of a module. Name is of type TEXT and defines the external name (typically a path name of the file system). At program initialization the persistent variable is connected to the external name. It is initialized to the old external content - if the latter is available. It is not prohibited to connect more than one variables with the same external name - this feature can be used for shared access. A predefined exception is generated if the old content is not compatible to the type of the set variable. An automatic

---

4 TEXT is a standard type in Modula-3.
conversion for this case is subject for further study. An object is persistent if and only if it is contained in or can be achieved from a persistent set.

3 Conclusions

A language extension has been presented that considers parallelism and persistence in a unified and simple conceptual frame, based on sets. The language - M3Set - has the unrestricted power of a general-purpose programming language. Automatic parallelization and optimization are restricted to the declarative select-expression. Thus, we buy for a relatively low effort (compared to nested-loop parallelization of typical parallel programming languages) high power. This seems to be a good deal. Common subexpressions can be optimized by the compiler statically. Cost based optimizations can be done at run-time, dynamically. The entire parallelization and optimization process is integral part of the language system. M3Set can be implemented in most different ways, starting at low-cost volatile sets processed sequentially, ending with persistent sets processed in parallel and optimized. A relatively simple and very fast implementation is based on PPOST, a main memory resident, persistent object store [3].

Acknowledgement

The authors thank to the anonymous referees for their valuable comments.

References


