Parallelism in MuPAD

Holger Naundorf
Universität Paderborn
33095 Paderborn, Germany
snoo@uni-paderborn.de

July 20, 1995

1 Motivation

When a computeralgebra-system is calculating, the size of the used data can increase dramatically – this is called intermediate data swell. All existing computeralgebra-systems try to cope with this problem by using (partially) unique data representation. That means that logically equal data is stored only once. Besides saving much memory this technique speeds up recognizing equal data as equal – usually a very expensive operation –, because equality can be decided in $O(1)$ if the compared data are stored at the same location. Recognizing the unequality of data can always be managed in $O(1)$ with high probability with the help of signatures [17].
Anyhow there exist many problems which produce data that is too huge for the memory of one machine. Enlarging secondary memory is no solution because swapping decreases efficiency too much.
One canonical idea is to use the memory of many connected machines as one huge memory. To avoid too many communications, every machine stores most of the data that it requires for computation. To limit the altogether used memory, unique data representation between machines should be possible.
Obviously a program must be parallel to really use the distributed memory and the computation power of the used machines. To encourage the user to write parallel programs, parallelism must be easy to use and efficient. When developing parallel programs, the perhaps hardest problems are mapping tasks onto existing processors and managing communication between the tasks.
Ideally MuPAD would do this on its own without bothering the user with details. Unfortunately a system can do this without informations only if the processors are connected tightly. Otherwise the user has to provide some information. But there are many completely different types of information a user can provide and it seems impossible to foresee all the types. Additionally for many types of information there are no optimal algorithms known. Therefore a parallel

paradigm must be provided that enables the user to take complete control over mapping and communication of tasks. In MuPAD there exist two different kinds of parallelism – micro- and macro-parallelism – similar to CLIDE [22], Distributed Eiffel [21] (both based on Clouds [4]) or LADY [19]. An overview over most existing parallel computeralgebra-systems can be found in [16].

2 Micro-Parallelism

2.1 Motivation

Many parallel programs use very fine-grained parallelism – very many very quickly solvable tasks are produced. Using tightly coupled computers it is worth exploiting this parallelism, but the distribution of the tasks on the participating computers must be very fast. Otherwise the speed-up gained by parallelism is lost by the distribution. Therefore the distribution of the tasks must not be implemented in the MuPAD-language but in the much more efficient C-language. Nevertheless this parallelism must be very easy to use to be accepted.
A more detailed description of micro-parallelism can be found in [5, 6].

2.2 Functionality

With the help of micro-parallelism the user can mark statements explicitly as executable in parallel. MuPAD executes statements, that are marked in this way, in parallel on the existing computers.
MuPAD uses dynamical scoping – a new task shares all variables with its creator. The only exception are the variables created locally by the new task. Further discussion about the advantages and disadvantages of dynamical scoping (with respect to lexical scoping) can be found in [18]. The shared variables can be used to communicate with other tasks.

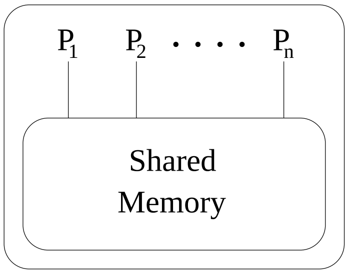

Figure 1: Micro-parallelism

2.2.1 Parallel for-Loops

MuPAD provides two different kinds of parallel for-loops (analogous to the two kinds of sequential for-loops). One version is:

for *i* **from** *start* **to** *end* **parallel**
  private *priv*;
  StmtSeq
end_for

This loop executes *StmtSeq* for every value between *start* and *end*. The variable i and the variables *priv* are local to every task.
As result the loop returns a sequence of the results of the created tasks in the natural order (since MuPAD is in some sense functional, every task returns a result).
The second version of a for-loop is:

for *i* **in** *A* **parallel**
  private *priv*;
  StmtSeq
end_for

This loop executes *StmtSeq* in parallel for every operand of A. Again the variable i and the variables *priv* are local to every task.
As result the loop returns a sequence of the results of the created tasks in the order according to the order of the operands of A.

2.2.2 Parallel and sequential Blocks

The statement

parbegin
  private *priv*;
  $stmt_1$
  $\vdots$
  $stmt_n$
end_for

executes the statements $stmt_1$ to $stmt_n$ in parallel – every task has the local variables *priv* – and returns a sequence of the results of the statements in their natural order.
The statement

seqbegin
  $stmt_1$

$\vdots$

$stmt_n$

```
end_for
```

executes the statements $stmt_1$ to $stmt_n$ sequentially.

2.3 Examples

Using two simple examples – calculation of Fibonacci-numbers and multiplying matrices – it will be demonstrated how to parallelize programs using micro-parallelism and what effects this has on runtime. Especially the program for calculating the Fibonacci-numbers is very bad but this example was taken for its striking simplicity.
The runtime was measured on a Sequent Symmetry with 12 processors of type 80386, each running with 16MHz. For sequential programs the time used by the processor was measured while for parallel programs the real time was measured. In the following tables runtime is measured in seconds. The speed-up is calculated with respect to the sequential version.

2.3.1 Fibonacci-program

Program Programs implementing the definition of Fibonacci-numbers are:

```
f1:= proc(n)
     begin
        if n < 2 then
           1
         else
           f1(n-1) + f1(n-2)
        end_if
     end_proc:

f2:= proc(n)
     begin
        if n < 2 then
           1
         else
           _plus((parbegin
                    f2(n-1) ;
                    f2(n-2)
                 end_par))
        end_if
     end_proc:
```

```
f3:= proc(n)
     begin
        if n < 12 then
           f1(n)
         else
           _plus((parbegin
                     f3(n-1) ;
                     f3(n-2)
                  end_par))
        end_if
     end_proc:
```

`f1` is the (very slow but) canonical implementation for calculating the Fibonacci-numbers. Obviously the summands in `f1` can be calculated independently and hence in parallel. In `f2` the summands are calculated in parallel by enclosing the calculation in `parbegin end_par`.
The tasks created by `f2` are very short – the parallelism is very fine grained. It is well known that too fine-grained parallelism can increase runtime. Therefore in `f3` less tasks are created with half of the created tasks having a longer runtime. At the moment there do not exist library-functions, which support such transformations – the user has to do this without any help. This is a weakness that must be removed as soon as possible.

Timings Even when running sequential programs on the parallel MuPAD-version there is some more work the parallel version has to perform. Consequently the parallel version is slower, when a sequential program is executed. In table 1 the timings for the sequential program `f1` on the sequential and parallel version are listed. The difference of performance is obvious and the parallel version has to be improved.
Table 2 shows the timings and speed-up for the parallel Fibonacci-programs `f2` and `f3` with several numbers of processors and several arguments. The timings of `f2` are much higher than the timings of `f1` and `f3`. It seems, that the time for creating and starting a task is much higher than the runtime of a task. Interestingly with increasing the number of processors the runtime of `f2` does not decrease much. Probably the processors disturb each other when creating and taking tasks and provoke idle times. Table 3 shows the number of tasks `f2` and `f3` create.

2.3.2 Multiplication of Matrices

Programs Programs for multiplying matrices (represented by tables) are:

```
mult := proc(n)
          local e, i, j, l, r;
```

```
        begin
           for i from 1 to n do
              for j from 1 to m do
                 e := 0;
                 for l from 1 to k do
                    e := e + m1[i, l]*m2[l, j]
                 end_for;
                 r[i, j] := e
              end_for
           end_for;
           r
        end_proc:

ms := proc(n)
      begin
         mult(20, n, 20, table(), table())
      end_proc:

pmult:= proc(n)
           local i, r;
        begin
           for i from 1 to n parallel
              private j ;
              for j from 1 to m parallel
                 private l, e;
                 e := 0;
                 for l from 1 to k do
                    e := e + m1[i, l]*m2[l, j]
                 end_for;
                 r[i, j] := e
              end_for
           end_for;
           r
        end_proc:

mp := proc(n)
      begin
         pmult(20, n, 20, table(), table())
      end_proc:
```

Obviously the parallel version can be derived from the sequential version easily.

Timings This time the runtime of the sequential program on the sequential and the parallel version of MuPAD are almost equal (table 4) and hence the

parallel version performs very good. The author does not know the reason for the changing behavior of the parallel version.
As could be expected, multiplying very small matrices is not worth using parallelism. But even for relatively small matrices the parallel programs exhibits good speed-up. The bad speed-up using 10 processors is probably due to other users allocating processors such that MuPAD was not really able to use 10 processors. E.g. one time *mp(40)* was calculated in 26s yielding a speed-up of 6.19.

3 Macro-Parallelism

3.1 Motivation

Micro-parallelism is very easy to use. But unfortunately it is neither possible to implement a shared memory efficiently nor to compute a good mapping of tasks with those informations provided by micro-parallelism automatically on a distributed computer with slow communication or high latency.
However the user can often provide informations like:

Parallelism of tasks: If a created task can itself be parallelized, this task should be solved on a parallel computer. If the task is purely sequential it should be solved on a fast computer, that need not be parallel.

Needed data: If a task needs much data it should be solved on a computer with much main memory to avoid swapping.

Runtime: If a task has a short runtime it is not worth solving it on a distant computer.

Urgency of results: If the result of a task is very urgent, it is necessary to solve it on a fast computer. Less urgent tasks can be delayed in order to solve this task quickly.

Dependencies of tasks: If tasks depend on each other and communicate often (e.g. pipelining) these tasks should be solved by computers with fast communication.

It is hard to design an easy parallel paradigm, that enables the user to give all these informations (and possibly more). Moreover, for many informations and hardware restrictions there is no optimal mapping algorithm known. Hence the user must have the possibility to implement his/her own mapping algorithm.
Therefore in MuPAD exists macro-parallelism – a comfortable message-passing paradigm. It is important to notice that the user will usually not use this paradigm directly but will use comfortable functions implemented in the library. Only if there do not exist adequate library functions the user will use the macro-parallelism directly or will implement his/her own library functions.

	f1(16)	f1(20)	f1(22)	f1(24)
Sequential	13/1	91/1	239/1	625/1
Parallel	17/0.76	117/0.77	317/0.75	837/0.74

Table 1: Timings/Speed-up of the sequential Fibonacci-program `f1` on the sequential and parallel MuPAD-version

	f3(16)	f3(20)	f3(22)	f3(24)	f2(16)	f2(20)
2	10/1.30	66/1.38	178/1.34		18/0.72	422/0.22
4	6/2.16	36/2.52				
6	4/3.25	21/3.50			10/1.30	350/0.26
8	4/3.25	21/4.33				
10	3/4.33	20/4.55	48/4.98	122/5.12	10/1.30	335/0.27

Table 2: Timings/Speed-up of of the parallel Fibonacci-functions

	f3(16)	f3(20)	f3(22)	f3(24)	f2(16)	f2(20)	f2(22)	f2(24)
Tasks	24	176	464	1218	3192	21890	57312	150048

Table 3: Number of tasks created by the parallel Fibonacci-programs `f2` and `f3`

	ms(1)	ms(2)	ms(5)	ms(10)	ms(20)	ms(40)
Sequential	4/1	6/1	13/1	28/1	59/1	161/1
Parallel	6/0.66	8/0.75	16/0.81	31/0.90	64/0.92	167/0.96

Table 4: Timings/Speed-up of the sequential program `ms` on sequential and parallel MuPAD

	mp(1)	mp(2)	mp(5)	mp(10)	mp(20)	mp(40)
2	4/1	6/1	11/1.18	19/1.47	37/1.59	90/1.79
5	3/1.33	4/1.50	5/2.60	9/3.11	18/3.27	43/3.74
10	5/0.8	4/1.50	5/2.60	6/4.66	14/4.21	34/4.73

Table 5: Timings/Speed-up of the parallel program `mp` with different numbers of processors

This technique can be used, because macro-parallelism is considered for coarse-grained parallelism. Therefore the time spent for the distribution of the tasks is not too important.

3.2 Functionality

MuPAD consists of an arbitrary number of clusters. Each cluster itself can consist of an arbitrary number of computers and can be programmed with the help of micro-parallelism. The clusters can communicate with each other with comfortable message passing constructs:

Queues: Every cluster possesses a queue for every name. By calling *writequeue(name, cluster, data)* every cluster can write the data *data* into the queue of cluster *cluster* with name *name*.

By executing *readqueue(name)* a cluster can read data out of its queue with the name *name*. If no value is in the queue this function returns the value *null()*. In contrast *readqueue(name, Block)* blocks until a value is written into the queue.

Pipes: Between every pair of clusters for every name there exists a pipe. This pipe can be used to exchange data. By executing *writepipe(name, cluster, data)* the cluster writes the data *data* into the pipe to cluster *cluster* with the name *name*.

By executing *readpipe(name, cluster)* a cluster reads data out of the pipe from the cluster *cluster* with the name *name*. If no value is in the pipe this function returns the value *null()*. In contrast *readpipe(name, cluster, Block)* blocks until a value is written into the pipe.

Net-Variables: For some purposes, e.g. broadcasting, pure message passing constructs are very awkward. Therefore in macro-parallelism there exist net-variables, that represent a shared memory between all clusters.

3.3 Implementation

3.3.1 Real Shared Memory Machine

Macro-parallelism is already implemented on real shared memory machines (and on those which provide a virtual shared memory). But these machines are coupled tightly and hence are not that interesting for macro-parallelism.

3.4 Message Passing

Metzner [15] implements macro-parallelism on distributed computers with the help of message passing.

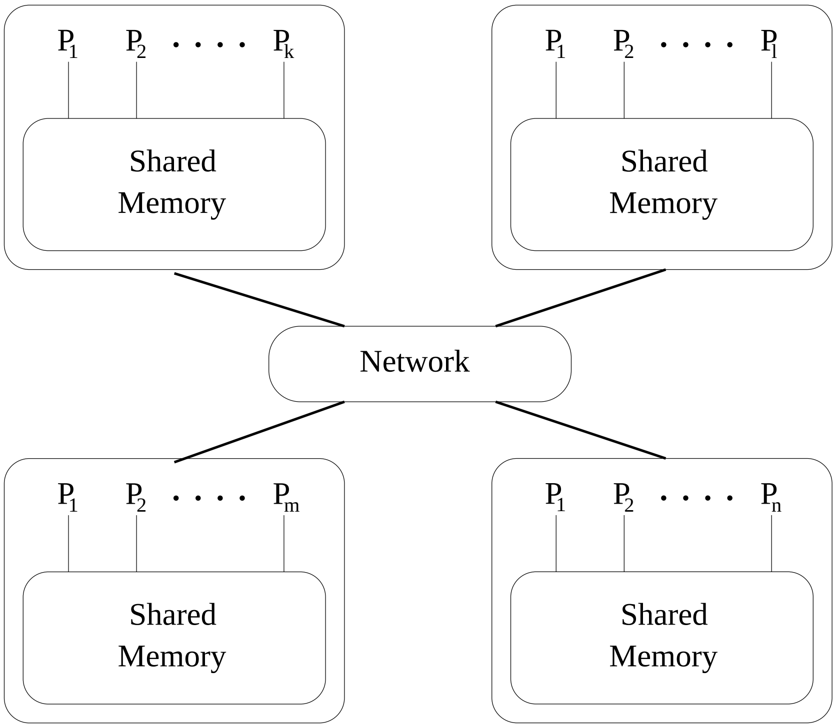

Figure 2: Macro-parallelism

A cluster, that sends data to another cluster, sends a real copy of this data. The advantage is that this is relatively easy to implement and to optimize. Other more complicated implementations can be compared to this implementation. This technique has the disadvantage, that no unique data representation between clusters is possible.

3.5 Virtual Shared Memory

Another (more complicated) method is to implement macro-parallelism on top of a special virtual shared memory. This has following advantages:

1. Unique data representation between clusters is possible.
2. If one cluster does not have enough memory to store all data that it needs, parts of this data can be stored in the memory of other computers. By this calculations can be continued that otherwise would have been stopped.

An efficient implementation of a virtual shared memory machine on loosely coupled computers is and probably will not be possible. Therefore it seems

impossible to implement macro-parallelism with the help of a virtual shared memory machine efficiently.
But macro-parallelism provides some information, that can be used for an efficient implementation.
After data is initialized it is never changed again (net-variables are an exception). This simplifies caching, because there are no coherence problems.
When producing data, that can be read by another cluster, it is known, which cluster will read this data (net variables are an exception again). The sending cluster can tell the virtual shared memory, which cluster will read this data and the data can be sent to this cluster (presending) and cached. This is similar to an implementation of Linda described in [24].
For every queue and for every pipe there exist only one cluster that can read out of it. Therefore when reading no synchronization is necessary.
The caching strategies used by the virtual shared memory implementation must be very good. A simple least recently used strategy will not work well, because it is more important to cache data from distant clusters than from near clusters.

4 Comparison with other Paradigms

4.1 Future Construct

A well known parallel paradigm is the future construct [11, 25, 10, 23]. It is mainly used in Lisp-dialects. A call *(future X)* returns immediately without having evaluated X. It returns a future – a value, that represents the value of X as long as X is not really evaluated. Moreover, a parallel task is started, that evaluates X. When X is evaluated, the result substitutes its future. If a task needs the real value of a future, this task is suspended until the value of the future is evaluated.
A very interesting extension of this concept is the concept of sponsoring [10, 20]. Every task has the ability to sponsor other tasks. As consequence the sponsored tasks get more resources. By this the urgent tasks get much computational power while the less urgent tasks get less computational power. Every task can change its sponsoring policy at every time.
The future construct can be used easily. There does not exist a possibility to influence mapping of tasks. Therefore optimizing parallelism in distributed system with slow communication is impossible.
The future construct can be implemented by using macro-parallelism and the domains concept of MuPAD. The implementation of sponsoring is not possible in an easy way.

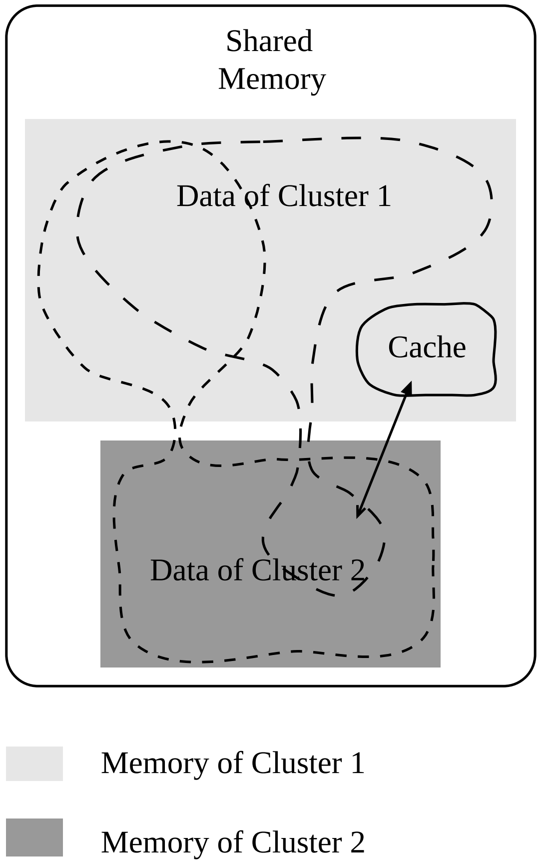

Figure 3: Macro-parallelism implemented with a shared memory

4.2 Tuple-Space

Tuple-space [7, 8, 9] is a simple paradigm to parallelize sequential languages. A tuple-space is a kind of a shared memory. There do not exist addresses in this memory. Instead data with specified properties can be read out. This paradigm is broadly used and accepted [1] and is used by the computeralgebra system Sugarbush [2, 3].

Tuple-space is one form of a virtual shared memory. Though it is very comfortable it is hard to implement it efficiently on a distributed computer with slow communication or high latency. [24] introduces a mapping function for data, which is definable by the user, which enables a much more efficient implementation of tuple-space.

Tuple-space can be implemented with macro-parallelism easily.

4.3 Process-Graph

In [12] an extension of a functional language similar to Miranda is introduced. This extension enables the programmer to describe the process-graph of an application. The compiler can use these informations to calculate a good mapping. This paradigm is not very easy to use and can be implemented with macro-parallelism easily.

4.4 Reflection

A very promising approach to controlling parallelism is reflection [14, 13]. Reflection means that a program can reason about and change its own behavior on a meta-level. By this many informations for parallelization usually given in a program can be provided on a meta-level and hence the informations about how can be parallelized can be separated from the informations about the actual mapping of tasks. The concept of reflection in best suited to object-based systems and though MuPAD has object-oriented features it is basically a functional system. Additionally there are still severe problems implementing reflection efficiently.

References

[1] Linda-like systems and their implementation. Technical Report 91-13, Edinburgh Parallel Computing Centre, June 1991.

[2] B. W. Char. Progress report on a system for general-purpose parallel symbolic algebraic computation. In T. T. Sasaki, editor, *Proc. of the ISSAC*. ACM Press, 1990.

[3] B. W. Char. A user's guide to sugarbush - parallel maple through linda. Preprint, September 1993.

[4] Dasgupta et. al. The design and implementation of clouds distributed operating systems. *USENIX Computing Systems Journal*, 3(1), 1990.

[5] Benno Fuchssteiner, Waldemar Wiwianka, Klaus Gottheil, Andreas Kemper, Oliver Kluge, Karsten Morisse, Holger Naundorf, Gudrun Oevel, and Thorsten Schulze. *MuPAD Benutzerhandbuch*. Birkhäuser Verlag, Basel, 1993.

[6] Benno Fuchssteiner, Waldemar Wiwianka, Klaus Gottheil, Andreas Kemper, Oliver Kluge, Karsten Morisse, Holger Naundorf, Gudrun Oevel, and Thorsten Schulze. *MuPAD User Manual (to appear)*. Birkhäuser Verlag, Basel, 1995.

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

[8] D. Gelernter. Getting the job done. *BYTE*, November 1988.

[9] D. Gelernter, N. Carriero, S. Chandran, and S. Chang. Parallel programming in linda. In D. Degroot, editor, *1985 International Conference on Parallel Processing*, pages 255–263, 1985.

[10] Robert H. Halstead and jr. New ideas in parallel lisp: Language design, implementation, and programming tools. In G. Goos and J. Hartmanis, editors, *Parallel Lisp: Languages and Systems*, volume 441 of *Lecture Notes in Computer Science*, pages 2–57. Springer-Verlag, June 1989.

[11] Morry Katz and Daniel Weise. Continuing into the future: On the interaction of futures and first class continuations. In G. Goos and J. Hartmanis, editors, *Parallel Lisp: Languages and Systems*, volume 441 of *Lecture Notes in Computer Science*, pages 101–102. Springer-Verlag, June 1989.

[12] Paul H. J. Kelly. *Functional programming for loosly-coupled multiprocessors.* Pitman Publishing, 128 Long Acre, London, UK, 1989.

[13] Satoshi Matsuoka. *Language Features for Re-use and Extensibility in Concurrent Object-Oriented Programming Languages.* Doktorarbeit, Department of Information Science, The University of Tokyo, 7-3-1 Hongo, Bunkyo-ku, Tokyo 113, Japan, April 1993.

[14] Satoshi Matsuoka, Takuo Watanabe, Yuuji Ichisugi, and Akinori Yonezawa. Object-oriented concurrent reflective architecture. In M. Tokoro, O. Nierstasz, and P. Wegner, editors, *Object-Based Concurrent Computing, ECOOP '91 Workshop*, volume 612 of *Lecture Notes in Computer Science*, pages 211–226. Springer-Verlag, July 1991.

[15] Torsten Metzner. Computeralgebra auf hetoregenen verteilten systemen (to appear). Diplomarbeit, Universität-Gesamthochschule Paderborn, 1995.

[16] Karsten Morisse. Parallele computeralgebra-systeme. *mathPAD*, 4(3):13–19, December 1994.

[17] Holger Naundorf. *MAMMUT: Memory Allocation ManageMent UniT (Entwurf und Implementierung einer effizienten parallelen Speicherverwaltung für symbolische Manipulation).* Diplomarbeit, Universität Paderborn, Germany, 1992.

[18] Holger Naundorf. *Ein denotationales Domain für parallele objekt-basierte Systeme (to appear).* Phd thesis, Universität Paderborn, Germany, 1995.

[19] J. Nehmer. Strukturierungskonzepte der sprache lady zur beschreibung verteilter betriebssysteme. In Karl Hubert Gl"asser, editor, *Verteilte Systeme: Workshop in der GMD am 4.11.1981*, volume 136 of *Berichte der GMD*, pages 99–121. R. Oldenbourg Verlag, 1981.

[20] Randy B. Osborne. Speculative computation in multilisp. In G. Goos and J. Hartmanis, editors, *Parallel Lisp: Languages and Systems*, volume 441 of *Lecture Notes in Computer Science*, pages 103–137. Springer-Verlag, June 1989.

[21] Przemyslaw Pardyak and Brian N. Bershad. A group structuring mechanism for a distributed object-oriented language. In *Proceedings of the 14th International Conference on Distributed Computing Systems*, 1994.

[22] Mark P. Pearson and Partha Dasgupta. Clide: A distributed, symbolic programming system based on large-grained persistent objects. ftp://helios.cc.gatech.edu/pub/papers/clide.ps.Z.

[23] Steffen Seitz. Algebraic computing on a local net. In R. E. Zippel, editor, *Computer Algebra and Parallelism*, volume 584 of *Lecture Notes in Computer Science*, pages 19–31. Springer-Verlag, May 1990.

[24] Greg Wilson. Improving the performance of generative communication systems by using application-specific mapping functions. In Greg Wilson, editor, *Linda-Like Systems and Their Implementation*, technical report 91-13 11, pages 129–142. Edingburgh Parallel Computing Centre, June 1991.

[25] Taiichi Yuasa. Premature return – another interpretation of the future construct –. In A. Yonezawa and T. Ito, editors, *Concurrency: Theory, Language, and Architecture*, volume 491 of *Lecture Notes in Computer Science*, pages 210–214, Berlin, September 1989. Springer-Verlag.