HAL
archives-ouvertes.fr

Progress Report on Parallelism in MuPAD

Christian Heckler, Torsten Metzner, Paul Zimmermann

▶ **To cite this version:**

Christian Heckler, Torsten Metzner, Paul Zimmermann. Progress Report on Parallelism in MuPAD. [Research Report] RR-3154, INRIA. 1997. <inria-00073535>

HAL Id: inria-00073535
https://hal.inria.fr/inria-00073535

Submitted on 24 May 2006

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Progress Report on Parallelism in MuPAD

Christian Heckler
Torsten Metzner

Paul Zimmermann

N ° 3154

Avril 1997

THÈME 2

ISSN 0249-6399

Progress Report on Parallelism in MuPAD

Christian Heckler*
Torsten Metzner
Paul Zimmermann†

Thème 2 — Génie logiciel
et calcul symbolique
Projet Euréca

Rapport de recherche n°3154 — Avril 1997 — 14 pages

Abstract: MuPAD is a general purpose computer algebra system with two programming concepts for parallel processing: *micro-parallelism* for shared-memory machines and *macro-parallelism* for distributed architectures. This article describes language instructions for both concepts, the current state of implementation, together with some examples.

Key-words: computer algebra, parallelism, shared-memory machine, distributed computation, Centre Charles Hermite, MuPAD

(Résumé : tsvp)

This work was done inside the Automath project (Paderborn) and the Centre Charles Hermite (Nancy).

* Université de Paderborn, Postfach 1621, D-33095 Paderborn, {chh,tom}@uni-paderborn.de
† Projet Euréca et Centre Charles Hermite (opération Calcul symbolique), BP 101, 54600 Villers-lès-Nancy Cedex, zimmerma@loria.fr

Unité de recherche INRIA Lorraine
Technopôle de Nancy-Brabois, Campus scientifique,
615 rue de Jardin Botanique, BP 101, 54600 VILLERS LÈS NANCY (France)
Téléphone : (33) 83 59 30 30 – Télécopie : (33) 83 27 83 19
Antenne de Metz, technopôle de Metz 2000, 4 rue Marconi, 55070 METZ
Téléphone : (33) 87 20 35 00 – Télécopie : (33) 87 76 39 77

Parallélisme en MuPAD

Résumé : MuPAD est un système de calcul formel général offrant deux sortes de parallélisme : le *micro-parallélisme* pour les machines à mémoire partagée et le *macro-parallélisme* pour les architectures distribuées. Cet article décrit par des exemples la sémantique du langage MuPAD pour ces deux types de parallélisme et l'état actuel de son implantation.

Mots-clé : calcul formel, parallélisme, machine à mémoire partagée, calcul distribué, Centre Charles Hermite, MuPAD

Introduction

Most parallel computer algebra softwares (see [13] for a review) are either extensions of existing systems (ALTS and PACLIB for the SACLIB library, PARSAC-2 for SAC-2, ||MAPLE|| and Sugarbush for Maple, RR for Reduce), or systems specialized for some kinds of operations (PAC for linear algebra), or only tools for distributed computing, without any symbolic manipulation system (DSC for C or Lisp programs). On the contrary, MuPAD is a general purpose system whose language contains parallel instructions since the very beginning (see [6, pages 169–181] and [7, pages 160–161]). However, until now no parallel version was officially released (the last version, MuPAD 1.3, was released for sequential machines in December 1996), due to lack of access to computers with enough processors and efficient threads [16]. The situation changed in 1996, where an access was given to the Power Challenge Array (PCA) from the Centre Charles Hermite at INRIA-Lorraine (Nancy, France). This machine has 4 clusters with up to 18 processors each (see Figure 1), i.e. it is exactly the architecture for which MuPAD was designed for (interesting enough, at the time the MuPAD language was designed, i.e. around 1989, such machines did not exist!). Up to now, prototype shared-memory versions of MuPAD were ported on the following architectures: Sun Sparc [16], Sequent Symmetry [17], Convex SPP 1600-16 and Silicon Graphics Power Challenge.

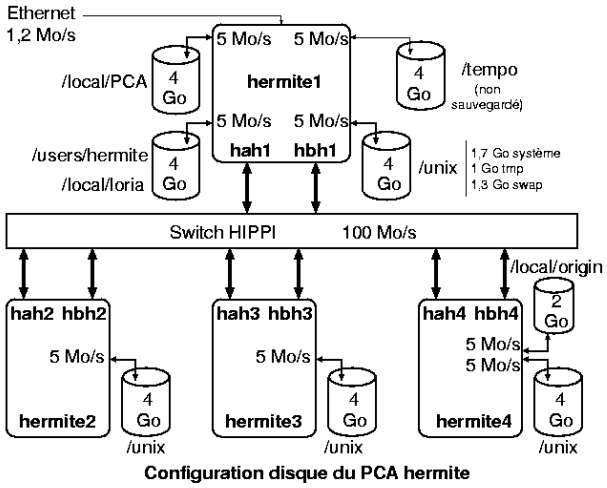

Figure 1: The Power Challenge Array architecture

The paper is organized as follows. Section 1 describes the current state of the shared-memory version of MuPAD, and shows a real example with the PCA prototype version. Section 2 describes the language constructs of the distributed version, and gives some hints how it will be implemented.

1 Micro-parallelism

In this section, we first describe the syntax of MuPAD parallel statements for shared-memory machines, then how these statements are implemented, and give some timings for polynomial factorization.

1.1 Language instructions

Two kinds of parallel constructs are available on shared-memory computers: parallel `for`-loops and parallel blocks. Parallel `for`-loops are exactly like sequential loops, except the keyword `do` is substituted by `parallel`:

for i **from** start **to** end **parallel**
  private j,k;
  stmts
end_for

The **private** keyword enables the user to declare auxiliary variables, which are local to each processor. Parallel blocks are defined as follows:

parbegin
  private j,k;
  $stmt_1$
  ⋮
  $stmt_n$
end_par

where the statements $stmt_1$ to $stmt_n$ are executed in parallel. One of these statements may itself consist of sequential instructions, in which case the **seqbegin** construct is needed:

seqbegin
  $stmt_1$
  ⋮
  $stmt_n$
end_seq

These three constructs (**parbegin**, **seqbegin** and parallel **for**-loop) are the only parallel constructs[1] available in the micro-parallel (shared-memory) version of MuPAD. They are

[1] These constructs are also available in the sequential version, where parallel tasks are processed in a random order. This enables the user to test parallel programs on a sequential machine.

enough to write some nice parallel programs, as we will see through polynomial factorization over finite fields.

Besides these constructs, since MuPAD version 1.3, *semaphores* and *locks* are available. A semaphore can be used to guard a resource with n instances. `sem:=Semaphore::sinit(n)` creates a semaphore `sem` and initializes it with n instances. `Semaphore::P(sem)` locks one instance of the desired resource, i.e. decrements the semaphore `sem`. The task blocks if the desired resource is not available, i.e. waits until the semaphore is greater than zero. `Semaphore::V(sem)` unlocks one instance of the desired resource, i.e. increments the semaphore. `Semaphore::N(sem)` returns the number of instances guarded by the semaphore `sem`.

A lock is a binary semaphore, the simplest type of a semaphore. `l:=Lock::linit()` initializes a lock `l`, `Lock::lock(l)` locks it, and `Lock::unlock(l)` unlocks it. For example a lock can be used to prevent multiple tasks from executing simultaneously a given section of a program (*critical section*), e.g. an access to a shared variable; the section begins with a lock operation and ends with an unlock operation.

1.2 Implementation

The implementation of the micro-parallelism concept is based on the memory management system MAMMUT [15]. MAMMUT offers a virtual shared memory machine, i.e. the programmer sees a set of parallel running MuPAD kernels sharing a common memory. In the shared memory, the stacks of the shared MuPAD variables and the *problem heap* of parallel executable MuPAD tasks are stored.

On Unix platforms the current version of MAMMUT works as follows. At the beginning the main MuPAD kernel process allocates shared memory using the Unix system call `mmap`. Shared variables are realized with pointers into that shared memory. After initializing some shared variables, parallel processes (also executing the MuPAD kernel) are created with a `fork` system call.[2] Thereby the number of processes is defined in the initialization file `.MMMinit`. The scheduling of the processes on the available processors is managed by the operating system.

In order to do some synchronisation between the processes a lock mechanism is necessary. For example the access to the stacks of the MuPAD variables in the shared memory must be coordinated. MAMMUT offers the possibility of locking a MAMMUT object, by setting its *lock bit*. The operation of testing and setting the lock bit must be protected using the operating systems lock (for example using the system functions `ussetlock()` and `usunsetlock()` on SGI machines), in order to prevent two processes from setting the lock bit of the same object simultaneously.

As mentioned above, MuPAD tasks that can be done in parallel are stored in a *problem heap* in the shared memory. When a process has nothing to compute it takes a task out of

[2] *Threads* could also be used in order to implement parallel running MuPAD kernels. However, when implementing the first parallel version no implementation of threads was available.

the heap. If there is no task in the problem heap the process goes to sleep and wakes up by a signal when new tasks come into the problem heap.

The garbage collection of MAMMUT is based on the principle of reference counting. Each object has a counter for the number of pointers pointing to it. If that counter is equal to zero the memory of the object can be deallocated. This method enables an *unique data representation*. Besides saving much memory this technique speeds up recognizing equal data, because equality can be decided by one pointer comparison. However, the frequently executed operation of changing and reading out the reference counter must be protected by a lock, which may be time consuming on some machines. This is the reason why the prototype parallel version for the Sun Sparc multiprocessor architecture gives no significant speedup [16]. Therefore, it is planned to develop a *parallel* mostly copying algorithm. The mostly copying garbage collection developed by Bartlett [4, 5] (see also [1]) is a variant of the classical *stop-and-copy* algorithm, where all accessible objects on the heap are copied into a new space. The advantages of this strategy are that it results in memory compaction and its running time is proportional to the amount of accessible memory. However, at the beginning an initial set of pointers to objects in the heap (the *root set*) must be known. Using the mostly copying strategy only a set of all possible pointers (the program stack and the registers) must be known, not the exact roots.

It is planned to demonstrate the first official shared-memory port for Sparc and Power Challenge at the CeBIT in Hannover in the middle of March. These parallel ports of MuPAD 1.3 will then be available on the MuPAD ftp mirrors around the world.

1.3 An example: polynomial factorization

Factorization of polynomials plays a central rôle in computer algebra, and in particular with coefficients in a finite field, since the factorization of polynomials with integer coefficients — either with the algorithm of Zassenhaus or that of Lenstra-Lenstra-Lovácz — begins with a factorization over a finite field [9]. For factoring polynomials of large degree n over finite fields, the best algorithm is currently that of Shoup [19], since it saves a factor of about $O(\sqrt{n})$ in both time and space with respect to Berlekamp's *big prime* algorithm [9].

One of the key operations in Shoup's algorithm is polynomial multiplication: indeed, the first step of the algorithm — computing $x^p \bmod f$ where f is the input polynomial, which requires $O(\log p)$ multiplications of polynomials of the size of f — takes a constant part of the total time. More precisely, computing $x^p \bmod f$ requires $\log p$ squarings and about $\frac{1}{2}\log p$ multiplications for an *average* p. Therefore we will focus on the squaring operation. The following MuPAD function computes the square of a polynomial a — with any kind of coefficients — using a parallel **for**-loop.

```
square := proc(a) local d,i,x,n;
begin
  d:=degree(a); x:=op(a,[2,1]);
  poly(_plus((for n from 0 to 2*d parallel
```

n	seq.	1 pr.	2 pr.	4 pr.	8 pr.	16 pr.	32 pr.
500	60s	124s	121s	91s	73s	49s	34s
1000	672s	851s	534s	445s	297s	191s	141s

Figure 2: Time needed for squaring a polynomial of degree n with coefficients of n bits.

```
    private i,s;
    s:=(if n mod 2=0 then coeff(a,n/2)^2 else 0 end_if);
    for i from max(0,n-d) to min(d,iquo(n-1,2)) do
      s:=s+2*coeff(a,i)*coeff(a,n-i)
    end_for;
    s*x^n
  end_for)),op(a,2..3))
end_proc:
```

Each parallel task computes one term $s_n x^n$ of a^2; the parallel **for** loop returns the sequence of these terms, which is then transformed into a sum using `_plus`, and into a polynomial with `poly`.

Figure 2 shows the time needed for squaring a dense polynomial of degree n over a finite field F_p with p a prime of n bits, for $n = 500$ and $n = 1000$, on the Origin 2000 (32 processors R10000) of the Centre Charles Hermite. The timings for 1 to 32 processors were obtained with the prototype parallel version of MuPAD for Silicon Graphics machines, whereas the timings in the seq. column correspond to the standard sequential version, available on all MuPAD ftp sites.

Two comments can be made about these timings. Firstly, the one-processor parallel version runs much slower than the sequential one, especially for $n = 500$ (the cost for the lock operations may explain that). Secondly, the speed-up does not grow linearly with the number of processors. This may be due to a cache phenomena, since the parallel squaring function performs more concurrent memory accesses. Nevertheless, with 32 processors, a factor of about 5 can be saved for squaring degree 1000 polynomials, whence one can expect a similar speedup for polynomial factorization.

2 Macro-parallelism

Figure 3 explains the concepts of micro-parallelism and macro-parallelism in MuPAD. Each cluster is made of several processors working on a shared memory (micro-parallelism). The clusters can communicate by a kind of message-passing (macro-parallelism).

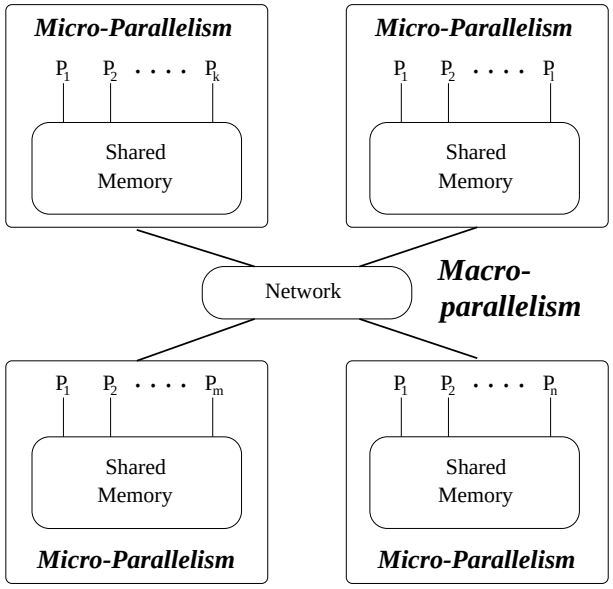

Figure 3: Micro-parallelism and macro-parallelism.

2.1 Language constructs

Due to the fact that we have concentrated our efforts on the (shared-memory) parallel version and other aspects of MuPAD there is no distributed (in the sense of message-passing) MuPAD version available so far.

From the viewpoint of macro-parallelism MuPAD consists of a set of independent clusters which are numbered from 1 to n. Each cluster in turn is formed of a number of running MuPAD kernels and micro-parallelism is (or can be) used inside every cluster. The MuPAD language provides to the user a special concept to program in this distributed environment. However the current user interface is not very pleasant to use. The interface consists of net variables as well as queues and pipes between different clusters.

The *net variables* are common to all clusters, so they can be considered as shared memory between the different clusters (i.e. global variables):

global(a)
global(a, b)

In the first form the net variable `a` is read out. Here `a` is considered to be a variable name so it is not evaluated. The result of the read operation is then evaluated. In the second form the value of `b` is assigned to `a`. Here again `a` is not evaluated and `b` evaluates as usual. During the evaluation of `b`, no other task can change the net variable `a`. Assigning a value to a net variable is therefore an atomic operation.

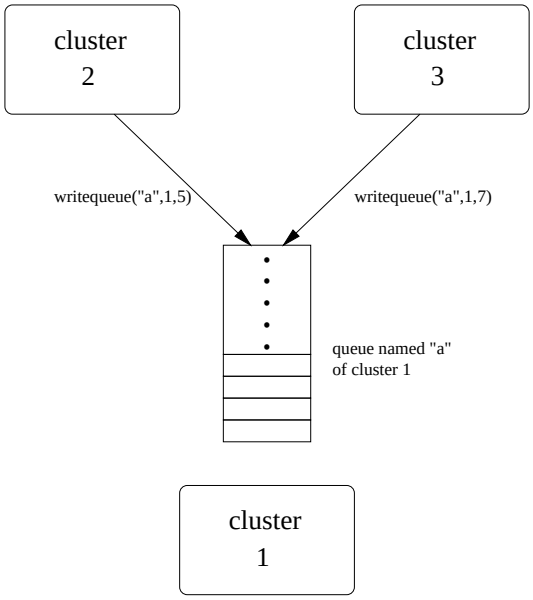

Figure 4: Queues

Each cluster can have any number of queues, and a cluster can write in any queue (see figure 4):

writequeue(a, i, value)

Here `value` is the expression which will be written to the queue `a` of the cluster `i`.

A cluster can only read the first element of its queue. There exists a blocking and a non-blocking read action on a queue:

readqueue(a)
readqueue(a, Block)

Here `a` is the name of the queue the cluster will read from. If the queue is empty the function call `readqueue()` will return a MuPAD object of type DOM_NULL. In the second case the function call `readqueue()` will block until there is a value available in the queue.

There exists one special queue named `"work"` in every cluster. After the system is initialized every cluster checks its queue with the name `"work"` periodically when it has no work to do. If the queue contains an expression, this one is read and evaluated. An exception is the cluster 1, which never checks its `"work"` queue, but communicates with the user and gets tasks in this way.

With queues it is possible to send an expression to every cluster, but it is not guaranteed that only one cluster writes in the queue of another cluster. So the cluster that reads from its queue cannot be sure that the value is sent from a specific cluster it wants to receive an expression from. So there exists another data structure named *pipe*, which is similar to queues (see figure 5). For two clusters i and j there exists exactly one pipe of name `a` that runs from cluster i to cluster j. A pipe `a` running from cluster i to cluster j and a pipe `a` running from cluster j to cluster i are two different pipes:

writepipe(a, j, value)

Here the expression `value` is written in the pipe `a` of cluster `j`. If this instruction is performed by cluster `i`, it creates a pipe `a` which runs from cluster `i` to cluster `j`.

readpipe(a, i)
readpipe(a, i, Block)

With the first `readpipe()` call, a cluster reads an expression from the pipe `a` of cluster `i`. The second call is the blocking variant of the function call `readpipe()`.

The function `topology` enables the user or programmer to know the topology of the distributed system: `topology()` returns the number of clusters in the system; `topology(0)` returns the number of MuPAD kernels, i.e. the sum of the MuPAD kernels of individual clusters; `topology(n)`, where `n` is a positive integer between `1` and `topology()`, returns the number of MuPAD kernels in cluster `n`; finally `topology(Cluster)` returns the index of the current cluster.

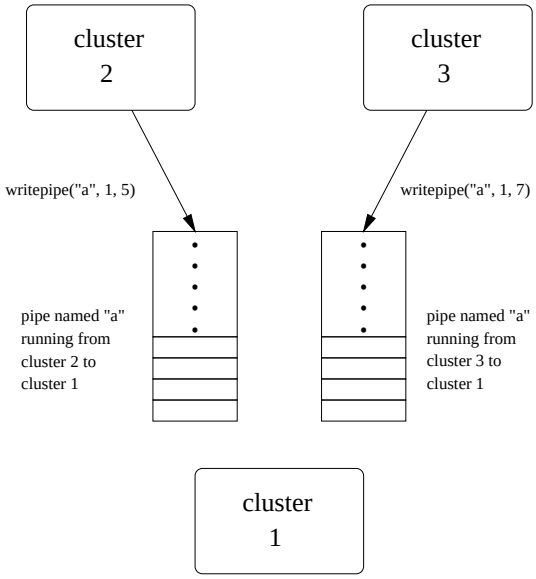

Figure 5: Pipes

2.2 Implementation

To implement the macro-parallelism it is necessary to transfer every MuPAD object in an efficient way. For this reason a binary encoding was chosen, following [3, 11]. Note that every MuPAD procedure is a regular MuPAD object. So this work was done for every MuPAD object. Internally a MuPAD object is represented by a directed acyclic graph (in some very special cases it is a cyclic structure), so common subobjects sharing should be also used. This is done in the obvious way. Every object is marked with a label. When a subobject occurs more than once in a MuPAD object only the first occurrence will be transferred to another cluster; if the subobject occurs a second time the label is sent as a reference. Here we use similar techniques like MP [11] and OpenMath [21] but in contrast to them it is also possible to send cyclic structures. As a result, we got an efficient encoding from a MuPAD object to a well-defined byte stream of elementary C data types. So the basics for implementing the concepts of queues and pipes in an efficient way are ready. Now we can use a transport protocol like PVM [10], MPI [12] or MP to send MuPAD objects between clusters. For the first prototype of a distributed MuPAD version which implements exactly the language constructs of the macro-parallelism described in section 2.1, we have chosen MPI because it is widely available. To integrate the transport layer of the distributed version, the concept of dynamic modules [20] is used.

It still remains the question how net variables could be implemented in an efficient way. For this purpose, it is planned to use *DIVA* (The **DI**stributed **VA**riables Library[3]), which is part of the A2 project of the SFB 376.

2.3 Future developments

Besides the work described in section 2.2, an asynchronous object oriented communication protocol named IPP is developed and partly implemented. This protocol has inherited a lot of ideas of PVM and the UNIX I/O concept. On the one hand a lot of ideas of PVM, because PVM is a distributed system and MuPAD is also a distributed (computer algebra) system. So the concept of dynamic host configuration and the creation of new processes are undertaken. On the other hand the concept of file descriptors in an UNIX environment is elegant. Once one has opened a file the interaction with a file, a network device or a screen is similar from a programmer's point of view. So we decided that this should be also possible in IPP. The main communication object in IPP is the *channel*. A channel represents a bidirectional communication connection. After one has opened a channel between two (or more) processes, a process and a file, or between two sockets, there is no difference in using this channel. One can send and receive data with simple function calls like `get()` and `put()`. One special, MuPAD specific aspect of the protocol is that it can transfer the data type *S_Pointer* which is the main MuPAD data type. Internally every MuPAD object is represented by an S_Pointer. When the implementation of the protocol is finished, it could be used to implement a better user interface to the macro-parallelism: for example to start a specific MuPAD procedure on some of the available clusters or on some given cluster, adding a cluster to the system, making the communication between clusters easier and so on. At the moment PVM is used to implement IPP, because it is widely available and supports process creation and dynamic system configuration.

3 Future Work

As already said in section 1, the first priority is to make available to the whole scientific community a (shared-memory) parallel version of MuPAD version 1.3, and to put that version on the MuPAD ftp servers.

Then a second objective will be to develop some nice applications and libraries for this parallel version. Good candidates are the algorithms presented at the first Pasco conference, for example the modular algorithm for sparse multivariate polynomial interpolation [14], parallel Buchberger algorithms [2], and more generally all algorithms on polynomials presented in Paul Wang's overview [22]. At the University of Paderborn, Christopher Creutzig is currently developing a general framework for constraint logic programming using parallel instructions, which can be used in turn in a parallel computation of Gröbner bases.

For the distributed version, we plan to have a first prototype version implementing the macro-parallel language before the end of this year.

[3] `http://www.uni-paderborn.de/sfb376/projects/a2/TP_A2_DIVA.html`

Acknowledgements. Work on the parallel versions of MuPAD is supported by the SFB project 376 *Massive Parallelism*[4] and by the European Procope project *Design and Implementation of Efficient Algorithms for Computer Algebra*[5]. For providing us an access to parallel computers, we would like to thank: the Centre Charles Hermite[6] (Power Challenge with 18 processors and Origin 2000 with 32 processors), especially Alain Filbois and Olivier Coulaud; Ulrich Schwardmann from the GWD in Göttingen (Silicon Graphics computer with 4 processors); Thomas Schramm and Peter Junglas from the University of Hamburg-Harburg (Convex with 16 processors).

References

[1] Attardi, G., and Flagella, T. Customising object allocation. In *ECOOP '94 – Object-Oriented Programming* (Berlin, July 1994), M. Tokoro and R. Pareschi, Eds., vol. 821 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 299–319.

[2] Attardi, G., and Traverso, C. Strategy-accurate parallel Buchberger algorithms. *Journal of Symbolic Computation 21* (1996), 411–425.

[3] Bachmann, O., Schönemann, H., and Gray, S. MPP: A Framework for Distributed Polynomial Computations. In *Proceedings of the 1996 International Symposium on Symbolic and Algebraic Computation: ISSAC'96* (1996), L. Y.N., Ed., ACM Press, pp. 103–112.

[4] Bartlett, J. F. Compacting garbage collection with ambiguous roots. Tech. Rep. 88/2, DEC Western Research Laboratory, Palo Alto, CA, Feb. 1988.

[5] Bartlett, J. F. Mostly-copying garbage collection picks up generations and C++. Technical Note TN-12, DEC Western Research Laboratory, Palo Alto, CA, 1989.

[6] Fuchssteiner, B., and al. *MuPAD Benutzerhandbuch.* Birkhäuser, Basel, 1993. In german.

[7] Fuchssteiner, B., and al. *MuPAD Tutorial.* Birkhäuser, Basel, 1994.

[8] Fuchssteiner, B., and al. *MuPAD User's Manual.* Wiley Ltd., 1996.

[9] Geddes, K. O., Czapor, S. R., and Labahn, G. *Algorithms for computer algebra.* Kluwer Academic Publishers, 1992.

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

[4] Cf `http://www.uni-paderborn.de/SFB376`
[5] See `http://www.loria.fr/~zimmerma/mupad/procope.html`
[6] Cf `http://www.loria.fr/CCH`

[11] Gray, S., Kajler, N., and Wang, P. S. Design and Implementation of MP, a Protocol for Efficient Exchange of Mathematical Expressions. *Journal of Symbolic Computation*. To appear.

[12] Message Passing Interface Forum. MPI: A Message-Passing Interface Standard. Tech. rep., University of Tennessee, Knoxville, Tennessee, June 1995. Available from `ftp.mcs.anl.gov` in `pub/mpi/mpi-1.jun95/mpi-report.ps`.

[13] Morisse, K. Parallele Computeralgebra-Systeme. *mathPAD 4*, 3 (Dec. 1994), 13–19. In german. URL `http://math-www.uni-paderborn.de/~cube/MATHPADS`.

[14] Murao, H., and Fujise, T. Modular algorithm for sparse multivariate polynomial interpolation and its parallel implementation. *Journal of Symbolic Computation 21* (1996), 377–396.

[15] Naundorf, H. *MAMMUT – Eine verteilte Speicherverwaltung für symbolische Manipulation*. MuPAD Reports. Teubner, Stuttgart. In german. To appear.

[16] Naundorf, H. Threads unter Solaris. *mathPAD 4*, 3 (Dec. 1994), 20–23. In german. URL `http://math-www.uni-paderborn.de/~cube/MATHPADS`.

[17] Naundorf, H. Parallelism in MuPAD. In *Electronic Proceedings of the 1st International IMACS Conference on Applications of Computer Algebra* (may 1995), M. Wester, S. Steinberg, and M. Jahn, Eds., http://math.unm.edu/ACA/Proceedings/MainPage.html.

[18] Naundorf, H. *Ein denotationales Modell für parallele objektbasierte Systeme*. MuPAD Reports. Teubner, Stuttgart, Dec. 1996. PhD Thesis. In german.

[19] Shoup, V. A new polynomial factorization algorithm and its implementation. *Journal of Symbolic Computation 20*, 4 (Oct. 1995), 363–397.

[20] Sorgatz, A. *Dynamische Module*. MuPAD Reports. Teubner, Stuttgart, 1996. In german.

[21] Vorkötter, S. Proposed OpenMath Specification. URL `http://www.can.nl/~abbott/OpenMath`, July 1995.

[22] Wang, P. S. Parallel polynomial operations on SMPs: an overview. *Journal of Symbolic Computation 21* (1996), 397–410.

Unit´e de recherche INRIA Lorraine, Technopˆole de Nancy-Brabois, Campus scientifique,
615 rue du Jardin Botanique, BP 101, 54600 VILLERS LÈS NANCY
Unit´e de recherche INRIA Rennes, Irisa, Campus universitaire de Beaulieu, 35042 RENNES Cedex
Unit´e de recherche INRIA Rhˆone-Alpes, 655, avenue de l'Europe, 38330 MONTBONNOT ST MARTIN
Unit´e de recherche INRIA Rocquencourt, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex
Unit´e de recherche INRIA Sophia-Antipolis, 2004 route des Lucioles, BP 93, 06902 SOPHIA-ANTIPOLIS Cedex

Éditeur
INRIA, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex (France)
ISSN 0249-6399