Parallelizing Algorithms for Symbolic Computation using ||MAPLE||*

Kurt Siegl
Research Institute for Symbolic Computation (RISC-Linz)
Johannes Kepler University
A-4040 Linz, Austria

e-mail: siegl@risc.uni-linz.ac.at
voice: (Austria) 7236 3231 66
fax: (Austria) 7236 3231 30

Abstract

||MAPLE|| (speak: parallel Maple) is a portable system for parallel symbolic computation. The system is built as an interface between the parallel declarative programming language Strand and the sequential computer algebra system Maple, thus providing the elegance of Strand and the power of the existing sequential algorithms in Maple.

The implementation of different parallel programming paradigms shows that it is fairly easy to parallelize even complex algebraic algorithms using this system. Sample applications (among them algorithms solving multivariate non-linear equation systems) are implemented on various parallel architectures. For example a straightforward parallelization of the complex and important problem of real root isolation has been parallelized using a generic Strand program of fewer than 20 lines of code and a slight modification of 5 lines in the original sequential Maple source. Even with such a simple modification we gained a speed-up of 5 times, that is better than those reported by others in the literature.

Keywords: Programming Methods for Parallel Symbolic Computation, Computer Algebra Systems, Logic Programming.

1 Introduction

Symbolic computation is an important part of applied mathematics, physics, engineering and other areas. Nowadays sequential computer algebra systems like Maple, Mathematica and Reduce are widely used. But currently it is only possible to solve small examples, because of the inherent complexity of the problems in symbolic computation. In particular, a lot of problems involving non-linear equation systems cannot be solved symbolically within a reasonable amount of time using today's computers. Furthermore, for a lot of algorithms it is not even possible to predict their runtime and memory demands for a given input.

How should we deal with such problems? An obvious solution is parallelization. Todays fast parallel computers are theoretically able to solve huge problems, but for solving large problems, we face the following situation:

- Sequential computer algebra systems have big sequential libraries and fast implementations of the basic algorithms but are not yet suitable for parallel computers.
- Parallel programming languages are well suited for parallel programming, but have no libraries for solving problems in symbolic computation.

Thus our goal should be to reuse the huge library in existing computer algebra systems and, additionally, to get easy access to the power of today's parallel computers.

Several attempts at parallelizing Maple have been made before. In [Watt, 1986] the UNIX fork and join primitives were used for parallelism. The programmer had to develop his parallel algorithm on a very low level. An improvement was done in [Char, 1990] where Linda is used. Linda is more hardware independent, but programmers still have to worry about creation of parallel processes and communication between them. The system was implemented on a shared memory machine but still had problems with communication and job creation overhead.

There are also several parallelizations of other computer algebra systems available: the C based SAC algebra library has been parallelized for shared memory machines only. An implementation by [Kuechlin, 1990] (called PARSAC) uses MACH thread routines for parallelization. A parallelization by [Hong *et al.*, 1992, Schreiner and Hong, 1993] (called PACLIB) includes several high level functions for communication and process generation. These systems provide the best efficiency but are not available for distributed memory machines.

*Supported by: Austrian Science Foundation (FWF), ESPRIT-III POSSO project

By contrast, ||MAPLE|| [Siegl, 1993] is especially designed for distributed memory architectures and is kept absolutely portable. ||MAPLE|| programs may run on different hardware without any modification or recompilation. All necessary communication is done automatically by the system without any additional programming effort. Since ||MAPLE|| uses implicit parallelism, it allows writing parallel programs without any expert knowledge in parallel programming.

A disadvantage compared to the shared memory systems, our design introduces a higher overhead for communication. Additionally, we have a bigger memory overhead due to the need of copying the required data between the Maple and Strand heaps.

To achieve this goal we developed an interface between Maple [Char *et al.*, 1983] with the parallel programming language Strand [Foster and Taylor, 1989]. The result is a parallel programming system with the full functionality of Maple and parallel power of Strand. This method allows easy writing and porting of parallel algorithms.

We give several sample programs describing different techniques for a quick parallelization of algorithms in ||MAPLE||. All our experiments are done on a 20 processor Sequent shared memory machine and tested on a network of SUN workstations as well as on a 16-processor Transputer distributed memory machine where they showed a good speedup for a minimal programing effort.

2 The ||MAPLE|| System

The ||MAPLE|| system has two layers (Figure 1). The top layer is the parallel declarative programming language Strand which controls the parallel execution of an algorithm. For performing sequential tasks we may call arbitrary Maple functions or sequences of Maple statements in the underlying Maple system.

This method combines the advantages of Strand as well as of Maple.

2.1 Strand

Strand is a parallel programming language of the guarded Horn clause type.

That means that the basic construct is a clause of the form:

$$H :- G_1, ..., G_n | B_1, ..., B_m. \qquad (n, m \geq 0)$$

where the bodies B_j are executed in parallel as soon as all the guards G_i are fulfilled.

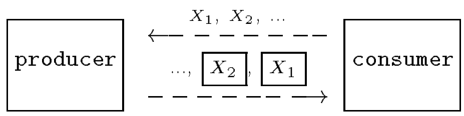
In Strand pattern matching is used instead of unification, and assignments are used to describe output bindings to increase the performance. Data dependencies can be expressed by data-flow synchronization and stream communication. Additionally Strand allows the use of arbitrary sequential subroutines written in C or Fortran as guards or body calls.

To describe the basic communication features, we give an short example of a program of the type "Producer - Consumer Parallelism with Request". The interested reader may find more details about Strand in [Foster and Taylor, 1989]

```
main():-
  producer(Channel),
  consumer(Channel)@fwd.

producer([X|Xs]):-  % Get empty variable X
  create(X),        % Generate value for X
  producer(Xs).

consumer(Out):-
  Out:=[X|Xs],      % Generate empty var X
  use(X),           % Use the value of X
  consumer(Xs).

-mode create(^).    % Generate an value
-mode use(?).       % Take an action.
```

A mode declaration for a procedure says that the argument annotated by "?" is used for input matching and arguments annotated by "^" should be used for output bindings. This declaration must be used if a pattern is used as output to allow data-flow synchronization. The "@fwd" annotation means that the annotated goal should be executed on the next processor.

Starting the `main` procedure generates two parallel subtasks (`producer, consumer`). Both are connected via a shared stream variable (`Channel`). The consumer process sends out a request for data (instantiating the first cons cell of the stream). This tells the producer to `create` a value for the stream element (`X`). As soon as it finished the `consumer` process can `use` the value.



2.2 Maple

Maple is a well known computer algebra system with one of the biggest libraries available. Most of the known algorithms for symbolic computation are already implemented in Maple. Maple has its own programming language that supports functional as well procedural programming styles in a Pascal-like syntax. Maple has a Kernel written in C which can be linked as a subroutine to C programs. Input and output may be handled via strings representing Maple expressions. Further information may be found in [Char *et al.*, 1983, Char *et al.*, 1988]

2.3 ||MAPLE|| Kernel

The main task for the Kernel of the system was to find a minimal set of interface routines combining the syntactic and

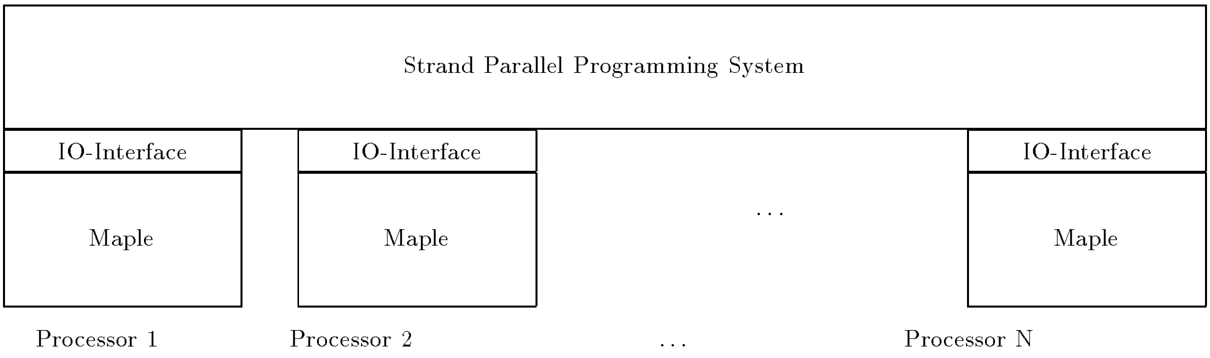


Figure 1: Structure of the ||MAPLE|| system

semantic features of the parallel, declarative programming language Strand with the imperative, sequential language of Maple. Finally we came up with an interface consisting of one new Strand guard and two bodies.

- A `maple` body call takes an arbitrary Maple function and its arguments as input and returns the result computed by Maple. This procedure covers most of the sequential calls to Maple used in a parallelized program.

  Using that call a sequential ||MAPLE|| program for symbolic integration for example would look like:

  ```
  int(F,X,Result):- maple(int(F,X),Result).
  ```

- To extend the non-deterministic guard check over Maple expressions, we need an `m_test` guard that takes a boolean Maple function and succeeds if the function evaluates to true and fails otherwise.

  Now we are able to write a Strand program finding the maximum of arbitrary Maple data structures by the statements:

  ```
  max(X,Y,Z):- m_test(less(X,Y)) | Z:=Y.
  max(X,_,Z):- otherwise | Z:=X.
  ```

  Since we can also use the guard operation for solving arbitrary constraints in the guard, ||MAPLE|| may also be used as a constraint logic programming language.

- For complex sequential operations, involving several input and output arguments we use an `inline` operation allowing the inclusion of a sequence of arbitrary Maple statements. Additionally we may specify which variables should be used as input and which ones as output. Here a tuple of the form `in(mvar,Svar)` means that the content of the Strand variable `Svar` will be assigned to the Maple variable `mvar` before the Maple program is executed. On the contrary an `out(mvar,Svar)` tuple stands for the assignment of the Maple expression denoted by `mvar` to the Strand variable `Svar` after the computation. The procedure collects all output expressions and returns them in a list as result.

  If one finally finds symbolic expression as the result and wants to generate an executable C program evaluating the result the ||MAPLE|| program looks like:

  ```
  generate_C(Expr,C-prog):-
    inline({in(tmp,Expr)},
      "readlib(C):C(tmp,optimized);",C-prog).
  ```

A critical part in the interface is the data type used for communication. It should be possible to manipulate the data within Strand as well as in Maple. Additionally an efficient communication between parallel tasks require a structure that may be packed in an array easily, because all communication primitiva on distributed systems may only transfer arrays in a single step. So we decided to use strings containing Maple expressions in legible form for communication between Strand and Maple.

In experiments we found out that this method allows easy porting and developing parallel algorithms used in symbolic computation.

3 Parallel Programming in ||MAPLE||

We describe the most important programming concepts necessary to parallelize algorithms in symbolic computation with ||MAPLE||. The reader should have some basic knowledge in parallel logic programming to understand the given programming examples.

3.1 Generating Parallel Tasks

At the beginning of a parallel program we will split several tasks from a single incoming expression. In ||MAPLE|| this can be done with the help of the `inline` function.

As example we consider the problem of integrating an sum expression symbolically. The integral may be easily parallelized by computing all individual terms in parallel. In ||MAPLE|| we generate the independent tasks by the following procedure:

```
split_int(SumExpr,TermList):-
 inline({in(sum,SumExpr)},
   "for term in sum do term; od;",TermList).
```

The `inline` call assigns the `SumExpr` to the Maple variable `sum`. In the in-lined Maple statement we print out all individual terms and collect them in the `TermList` as result list understandable by the Strand part for parallel manipulation.

We are now left over with a list of independent terms. According to the integration rules we may compute all individual terms independently.

3.2 Parallel Evaluation

As the simplest parallel program, we consider the case where we may compute all tasks independently in parallel.

To continue our integration example, we give the program for distributing the data in a parallel shell. For simplicity we assume that our integral has to be solved in the variable x.

```
-mode par_int(Input?,Output^).
par_int([],[]).
par_int([T|Ts],[Y|Ys]):-
   maple(int(T,x),Y),
   par_int(Ts,Ys)@fwd.
```

As first step, we split the list into the single elements and return a list of unbound variables as result immediately. The individual values of these variables are not known at this stage. Then we integrate all the terms of the input list in parallel. The result values are bound to the corresponding variables of the result list as soon as the individual processes integrating the single terms are finished.

3.3 Combining the Result

Finally we have to combine all individual subexpressions to a single expression. In our example this is done by the following procedure.

```
combine_int(ResultList,ResultExpr):-
  list_to_maple(ResultList,MapleList),
  maple(convert(MapleList,"`+`"),ResultExpr).
```

The builtin ||MAPLE|| call `list_to_maple` synchronizes the `ResultList` and converts its elements to a Maple list expression. Finally we have to convert the list back to a sum expression using the corresponding Maple function.

The whole parallel integration program now looks like:

```
sum_int(SumExpr,Result):-
  split_int(SumExpr,TermList),
  par_int(TermList,ResultList),
  combine_int(ResultList,Result).
```

To improve the performance we may use a dynamic load balancing scheme based on the manager-worker concept (see: Section 4.3).

Experimenting that example shows that a higher amount of garbage collection in a sequential system allows a superlinear speedup with a low number of processors. Maple has the ability to remember previously computed functions and need not compute such a function twice. The fact that the parallel system cannot utilize that feature and the unbalanced nature of the problem restricts the speedup achievable with a high number of processors.

3.4 Meta Programming Methods

Maple has the ability to return partially evaluated functions. We use that fact if we already have a sequential program available and would like to parallelize it with a minimum of effort.

Instead of immediately evaluating all functions, we just return them with evaluated arguments. A control process picks up all partially evaluated functions and distributes them to the parallel processors.

As an example, we consider a parallelization of the following Maple procedure:

```
seq_prog:=proc()
  var:= do_something();
  [p1(var),p2(var),p3(var)]
end;
```

This procedure generates some value for `var` and returns a list of the three elements computed by the independent sub procedures $p_{1..3}$.

For parallelizing this procedure, we use a generic parallel evaluation procedure that takes a list of Maple statements as input and returns a list of results.

```
-mode par_list(Input?,Output^).
par_list([],[]).
par_list([Expr|ExprList],[Y|Ys]):-
   inline([],Expr,Y),
   par_list(ExprList,Ys)@fwd.
```

The parallelized Maple program looks like:

```
par_prog:=proc()
  var:= do_something();
  print('p1'(var));
  print('p2'(var));
  print('p3'(var));
end;
```

Now Maple evaluates only the variable *var* and returns the independent sub procedures $p_{1..3}$ unevaluated.

To complete the program we have to combine the Strand and Maple part.

```
example(Result):-
 inline([],"par_prog():",WorkList),
 par_list(WorkList,ResultList),
 maple_to_list(ResultList,Result).
```

For an efficient parallelization of more complicated Maple programs we have also implemented generic versions of the parallel evaluation and a generalized divide and conquer (see: Section 4.1) method based on dynamic load balancing using the manager-worker scheme (see: Section 4.3).

4 Main Parallelization Techniques

We describe several programing techniques used for parallelizing algorithms in ||MAPLE||. Each technique is explained on an example of an important algorithm used in symbolic computation.

4.1 Divide&Conquer: Real Root Isolation

An important problem in symbolic computation is the real root isolation problem. The goal is a procedure for finding rational intervals around all real roots of polynomials with real coefficients, where each interval contains only a single root. In Maple this is done with the Uspensky procedure [Collins and Loos, 1982]. The algorithm is of the divide and conquer style.

The idea of parallelizing a divide-and-conquer algorithm is clear. The input is split into independent subproblems of the same type and the subproblems are solved on different nodes. On each of them we recursively solve the subproblem again until a base case is reached. This leads to a tree structure of parallel processes.

The Uspensky procedure splits the range of roots into two pieces and calls itself recursively until we found an isolating interval. Since roots normally are not equally balanced, only one half is worth for further search. Because of this fact we do not expect a high speedup for that problem.

For parallelizing that algorithm we used a generic divide and conquer algorithm (Figure 2), based on the meta programming technique (3.4). We modified the original Maple algorithm in the way that Maple returns the recursive calls unevaluated instead of computing the result. These calls are distributed to the parallel processors for further computation.

In more detail a Maple routine (Mcall) called by the generic divide and conquer algorithm (divconq) simply has to print out all recursive calls unevaluated. They are collected in the Strand list Parts. Additionally the global Maple variable compose_fun has to be assigned with a function for composing the evaluated parts again. In case of the base case compose_fun has to be assigned to 0. Until the base case is not yet reached, the split_comp procedure collects all the results of the recursive calls (split procedure) and transfers them to a Maple list (Mlist). Finally the composition function (compose_fun), returning the result, will be executed.

```
divconq(Mcall,Result):-
   inline(out("compose_fun",Comp),
     Mcall,Parts),                        % divide
   split_comp(Comp,Parts,Result).

-mode split_comp(?,?,^).
split_comp('0',[X],X):-   % Base case: 0 function
split_comp(Comp,Parts,Result):-
   otherwise |
     split(Parts,Results),
                          % Wait for sub-proc be comp
     list_to_maple(Results,Mlist),
                              % Compose partial results
     inline(in("in_args",Mlist),Comp,Res),
     strip_list(Res,Result).

-mode split(?,^).
split([X|Xs],[Y|Ys]):-
   divconq(X,Y)
   split(Xs,Ys)@fwd
split([],[]).
```

Figure 2: Generic Divide&Conquer Algorithm

This function has access to the results of the recursive calls over the Maple list variable in_args.

An improved version of that algorithm uses a dynamic load balancing algorithm based on the manager-worker scheme for the recursive calls.

We give the results on instances of the Chebyshev polynomials $T_n(x)$ on a 20 processor Sequent shared memory machine. Times are given in seconds of total execution time, including time for garbage collection and loading the required modules. The numbers in parentheses denote the speedup gained.

n	Maple V	‖MAPLE‖ on	
		4 Proc.	16 Proc.
20	16	13 (1.2)	12 (1.3)
40	252	98 (2.6)	71 (3.5)
50	956	365 (2.6)	190 (5.0)

As we expected by the analysis of the program, the speedup gained is not very high. But compared with parallelizations reported by the literature ([Char, 1990]) where a maximum speedup of 1.7 with the same examples was achieved, we found ourselves rather successful. The better performance comes mainly from a lower communication and programming overhead in our system.

4.2 Pipelining: Gröbner Bases Computation

The Gröbner bases algorithm invented by [Buchberger, 1965] is one of the most important and time intensive algorithm in symbolic computation. It is used for solving many problems in polynomial ideal theory. In particular, it can be used for the symbolic solution of arbitrary nonlinear multivariate polynomial equations. The parallel implementation of the Gröbner bases algorithm described in [Siegl, 1990, Siegl, 1991] uses a pipeline principle for polynomial reduction. Here we give only a sketch of the overall method. All details and proofs are given in [Siegl, 1990].

In a pipeline algorithm we have to test or modify the input with respect to a set of properties. After the test modify part we may (like in the well known Concurrent Prolog implementation of the Sieve of Eratosthenes [Shapiro, 1987]) append some of the tested values to our set in order to use it for testing the rest of the input. In some cases the result of a test has to be used for generating new input values. Typical examples are all Critical Pair Completion procedures. Here a set of critical pairs is built and tested with respect to a given set of properties. Pairs that are not canceled are appended to the test set and new critical pairs are built.

For parallelizing this method, we assign each element of the test set to one parallel process connected in a pipe. The input values are driven subsequently through the pipe. Each intermediate process takes the first element from its previous neighbor and tests, modifies or cancels it. The result is sent to the next process. Then it waits for the next input for computing the result. Whenever a value reaches the end of the pipe-line it is possible to extend the test set by simply creating a new process holding the new value at the end of the pipe. Thus, the future input values will be tested with respect to the new value, too.

If necessary, we have to send the computed values back through the pipe for generating new input. In each intermediate pipe process, we generate new input values and send them back to the front end. The front end process takes the new values coming back from the pipe and sends them through the pipe again.

The sequential Gröbner bases algorithm is a Critical Pair Completion algorithm. Given a set of input polynomials as initial basis, one computes critical polynomials and reduces them with respect to the basis polynomials. The resulting polynomials are added to the basis and new critical pairs are formed.

In the parallel version we have a pipeline of processes reducing the input polynomials with respect to an increasing set of basis polynomials. The irreducible input values are appended to the reduction processes. For generating new critical polynomials (called S-polynomials) as input, the new polynomials joining the basis are sent back through the pipe building the S-polynomials with the earlier computed basis polynomials. The process at the front of the pipe only needs to collect the computed S-polynomials and sends them to the reduction processes again.

As a result we get a bidirectional pipe forming a dynamically expandable ring. The organization of the pipe to compute the Gröbner basis is shown in (Figure 3):

```
-mode main(InPoly?,Basis^).
main(InPoly,Basis):-                      % Main loop
    append_spol(InPoly,Spol,Testpoly),    % Top end
    sieve(Testpoly,Spol,Basis)@fwd.       % Back end

-mode sieve(Test-polys?,S-polys^,Basis-polys^).
sieve([BP|TPs],[sp(BP,[])|SPs1],[BP|BPs]):-
    BP = \ = '0' |                        % New basis poly
      filter(BP,TPs,SPs,RPs,SPs1),
      sieve(RPs,SPs,BPs)@fwd              % Extend pipe
sieve(['0'|TPs],SPs,B):-                  % Zero reduction
    sieve(TPs,SPs,B).
```

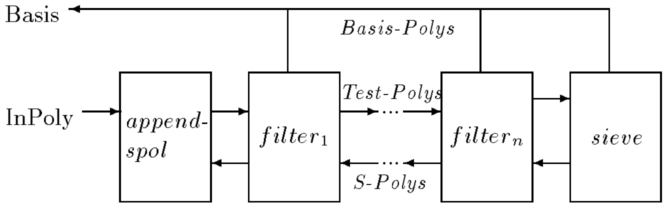


Figure 3: Organization of the Pipe for Computing Gröbner Bases

In more detail the main procedure generates the top end (append_spol) and the back end (sieve) process of the pipe. The top end process collects the newly generated S-polynomials (Spol) from the pipe, appends them to the initial input polynomials (InPoly) and sends the whole to the pipe again. The intermediate (filter) processes take each polynomial, reduces it with respect to its basis polynomial and returns the result to the next pipe process. As soon the back end receives a non zero polynomial it extends the pipe by a new intermediate (filter) process and sends that new basis polynomial (BP) back through the pipe for generating new S-polynomials. Polynomials reducing to zero are canceled out. As soon there will be no further polynomials flowing in the pipe the algorithm terminates.

The benchmarks are done with standard example from the literature ([Boege *et al.*, 1986]) computed with different term orderings (total degree, pure lexicographic term order).

Example	Order	‖MAPLE‖ on		
		1 Proc.	4 Proc.	16 Proc.
Trinks2	tdeg	125	53 (2.4)	28 (4.5)
Robbiano	tdeg	156	73 (2.1)	31 (5.0)
Katsura3	plex	147	38 (3.9)	11 (13.4)
Katsura3	tdeg	196	51 (3.8)	14 (14.0)
Trinks2	plex	272	62 (4.4)	16 (17.0)

The Gröbner bases algorithm has a nondeterministic behavior. It is not possible to decide in advance which order of reduction should be chosen. In our implementation polynomials flowing through the pipe are allowed to overtake each other if they will be reduced faster. In general it turns out that using faster generated and therefore simpler polynomials earlier reduces the overall computation time. This may lead to a super-linear speedup.

4.3 Manager/Worker: Parallel Interactive User Interface

||MAPLE|| may not be used for developing parallel algorithms only. It is even possible to parallelize Maple's interactive user interface, in a way that several user queries can be fullfiled simultaneously. The user interface of ||MAPLE|| is internally parallelized, using a manager worker scheme of parallelism.

```
shell(N):-                          % Shell with N workers
    merger(Workers,Request),        % Merge streams
    init_worker(N,Workers),         % Spawn N workers
    manager(Work,Request),          % Initialize manager
    shellinput(Work).               % Get user request

-mode init_worker(?,^).
init_worker(0,[]).
init_worker(N,[merge(Request)|Rs]):- N>0 |
    worker([],Request)@N,           % Place worker
    N1 is N-1,
    init_worker(N1,Rs).

manager([],_).
manager([{Mexpr,Res}|Work],[m(Expr,Mres)|Request]):-

    Expr:=Mexpr,                    % Send expr. to worker
    Res:=Mres,                      % Send result to user
    manager(Work,Request).

-mode worker(?,^).
worker(X,[m(Mexpr,Result)|Request]):-
    data(X) |                       % 1 instance per proc.
      maple(Mexpr,Result),          % Compute request
      worker(Result,Request).
```

Figure 4: Parallel User Shell

The manager-worker approach is used for parallelizing algorithms where a big set of independent parts and a limited number of nodes is used. The computing time of the various parts may vary and the load should be equally distributed among the nodes.

Many problems that may be parallelized with other methods may also be parallelized using the manager-worker approach. In practice, the method is very often used for avoiding load balancing problems.

In the shell, each user request is processed on a different node. The computing time of each request will vary greatly.

For parallelizing the user interface (Figure 4), we use a set of worker processes which subsequently solve one query. Each worker requests a piece of work from the manager process. The manager responds to the request with the next user query. Whenever one worker has finished its job it sends the result back to the manager and receives a new query. This allows the user to make queries without having to wait for the results.

In more detail, we spawn a set of workers (init_worker) and a manager process. Whenever a worker is idle it sends a request for new work (m(Mexpr,Result) tuple with empty variables). A merger combines the individual requests of the workers to a single stream connected to the manager. Whenever the manager gets new input to compute ({Mexpr,Mres}) and finds a worker requesting new work, it sends the work to the corresponding worker (Expr:=Mexpr) and returns the location of the result to the I/O-process (Res:=Mres). The worker may compute now the user request. As soon it is ready the result is sent directly to the I/O-process and a new request for work is sent to the manager.

In the actual implementation of the ||MAPLE|| user interface, we may also describe dependencies between parallel tasks.

The parallelized user interface has the advantage that the user may type in one query after the other and never be blocked waiting for the results. One can not give objective values for the performance gained by such an user interface. As soon one can type in queries faster than the whole system can solve, one will gain up to a linear speed-up. But the subjective performance due to the non blocking behavior will be much higher than the measurements will tell. This gives a significant speedup in the problem solving capacity.

5 Conclusion

We showed the usage of a system for parallelizing algorithms in symbolic computation in a simple and elegant way. Several programming techniques are given for developing and parallelizing algorithms. We found out that even with a quick parallelization it is possible to get up to a linear speedup as long as there is enough inherent parallelism and the grain size is large enough.

Due to the high level approach used in ||MAPLE|| we could keep our algorithms portable across different parallel architectures. Since ||MAPLE|| is available on many parallel computers ranging from shared memory systems over several distributed memory systems up to workstation networks we are able to develop algorithms on one machine architecture and test them on different others without any modification. Utilizing the whole sequential library of Maple enabled us to parallelize rather complicated algorithms.

In various tests we found that the main problem in achieving a high speedup is to find a proper division strategy of the algorithm to exploit the inherent parallelism. For algorithms for symbolic computation, an efficiency of 50 percent may be a very successful result. While utilizing only coarse grain parallelism, the actual communication time in the system has a minor influence on the performance.

Currently, the interface uses strings for communication which introduces a small linear overhead for parsing. Parsing and transferring for example the factorial of 1000, which is an 2568 digits long number, from Strand to Maple and back again requires approximately 0.3 sec on an i386 processor, whereas just squaring this number in Maple requires around 3 seconds. Independent operations in the sequential Maple kernel should be large enough to cover this overhead for parsing the expressions to strings suitable for the interface. In later releases, we will use a linearized form of the internal data structure for communication to allow a finer granularity.

We will also investigate various algorithms in more detail and try to find better parallelization strategies for a better efficiency.

References

[Boege *et al.*, 1986] W. Boege, R. Gebauer, and H. Kredel. Some Examples for Solving Systems of Algebraic Equations by Calculating Gröbner Bases. *Journal of Symbolic Computation*, 2(1):83–96, 1986.

[Buchberger, 1965] Bruno Buchberger. *Ein Algorithmus zum Auffinden der Basiselemente des Restklassenringes nach einem nulldimensionalen Polynomideal (An Algorithm for Finding a Basis for the Residue Class Ring of a Zero-dimensional Polynomial Ideal) (In German)*. PhD thesis, University of Innsbruck, Department of Mathematics, Austria, 1965.

[Char *et al.*, 1983] Bruce W. Char, Keith O. Geddes, W. Morven Gentleman, and Gaston H. Gonnet. The Design of Maple: A Compact, Portable and Powerful Comuter Algebra System. In *EUROCAL '83, Proceedings of the International Symposium on Symbolic and Algebraic Computation*, pages 101–115. Springer, Berlin, 1983.

[Char *et al.*, 1988] Bruce W. Char, Keith O. Geddes, Gaston H. Gonnet, Michael B. Monagan, and Stephen M. Watt. *MAPLE — Reference Manual.* Symbolic Computation Group, Department of Computer Science, University of Waterloo, Ontario, Canada, 5th edition, March 1988.

[Char, 1990] Bruce W. Char. Progress Report on a System for General-Purpose Parallel Symbolic Algebraic Computation. In *Proceedings of the ISSAC'90, Tokyo, Japan, August 20–24*, pages 96–103, Department of Computer Science, University of Tennessee, Knoxville, TN 37996-1301, 1990. ACM Press, New York.

[Collins and Loos, 1982] G.E. Collins and R. Loos. Real zeros of polynomials. *Comuter Algebra, Symbolic and Algebraic Computation*, pages 83–94, 1982.

[Foster and Taylor, 1989] Ian Foster and Stephen Taylor. *Strand — New Concepts in Parallel Programming.* Prentice-Hall, Englewood Cliffs, New Jersey, 1989.

[Hong *et al.*, 1992] Hoon Hong, Wolfgang Schreiner, Andreas Neubacher, Kurt Siegl, Hans-Wolfgang Loidl, Tudor Jebelean, and Peter Zettler. PACLIB User Manual. Technical Report 92-32, RISC-Linz, Johannes Kepler University, Linz, Austria, May 1992. Also: Technical Report 92-9, ACPC Technical Report Series, Austrian Center for Parallel Computation, July 1992.

[Kuechlin, 1990] Wolfgang Kuechlin. A Parallel SAC-2 Based on Threads. Technical report, Computer and Information Sciences, Ohio State University, Columbus, Ohio, April 1990.

[Schreiner and Hong, 1993] Wolfgang Schreiner and Hoon Hong. PACLIB — A Library for Parallel Algebraic Computation on Shared Memory Computers. In *Parallel Systems Fair at the Seventh International Parallel Processing Symposium*, Newport Beach, CA, April 13–16, 1993.

[Shapiro, 1987] Ehud Shapiro, editor. *Concurrent Prolog, Collected Papers, Volumes 1 and 2.* MIT Press, Cambridge, Massachusetts, 1987.

[Siegl, 1990] Kurt Siegl. Gröbner Bases Computation in STRAND: A Case Study for Concurrent Symbolic Computation in Logic Programming Languages. Master's thesis, Johannes Kepler University, Linz, Austria, 1990. Also: Technical Report 90-54, RISC-Linz, Johannes Kepler University, Linz, Austria, 1990.

[Siegl, 1991] Kurt Siegl. Parallel Gröbner Bases Computation in ||MAPLE||. Technical Report 92-11, RISC-Linz, Johannes Kepler University, Linz, Austria, 1991. Also: Technical Report 92/13, Austrian Center for Parallel Computation, July 1992.

[Siegl, 1993] Kurt Siegl. ||MAPLE|| — A System for Parallel Symbolic Computation. In *Parallel Systems Fair at the Seventh International Parallel Processing Symposium*, Newport Beach, CA, April 13–16, 1993.

[Watt, 1986] Stephen Michael Watt. Bounded Parallelism in Computer Algebra. Research Report CS-86-12, Faculty of Mathematics, University of Waterloo, Ontario, Canada, 1986.