# On scheduling algebraic algorithms for parallel execution *

*Hong R. Lee*

*B. David Saunders*

*Ruth Shtokhamer*

Department of Computer and Information Sciences

University of Delaware

Newark, DE 19716

June 21, 1995

**Abstract**

An adaptive run time scheduling approach to parallel algebraic computation is suggested. Study of a toy example, the factorial function, illustrates the need for such an approach on an asynchronous multiprocessing system. The example is run in several parallelized variants on two distinct implementations of ALDES/SAC2 on a Sequent Symmetry computer.

*This work has been supported by the National Science Foundation under Grants No. CDA-8805353 and CCR-8614219.

# 1 Introduction

Algebraic computation holds special challenges for parallel computation. Two obstacles arise which we may call algorithmic and structural. First the algorithmic.

Some algorithms are strongly sequential with each step depending directly on the preceding ones. However, already a substantial theory has developed of parallel algebraic algorithms. For example many problems may be computed in polylogarithmic, $O((log\ n)^k)$, time in a model of computation which allows polynomially many processing elements [4]. It is also often clear in principle how to take advantage of a fixed number of processors effectively on a particular problem. However, in practice, various overheads of communication in a multiprocessor can easily erode the potential gains of using several processors. Furthermore, even if one has an effective way to use $p$ processors on a particular algorithm, that algorithm may be only one part of a given task. When $p$ processors are available, there can be a considerable challenge how to distribute them to the subalgorithms of the task at hand, as we show by an example. One wants a way to express the parallelizability of an algorithm in such a way that it can be used in varying resource environments, whether the variation is that between machines or that of the variable role of the algorithm in larger computations. To deal with algorithmic problems, not only must parallelizable algorithms be found, but effective analysis must be available to express how the speedup depends on the number of processors used.

The structural obstacle we find arises from the heterogeneity of much algebraic data. For highly homogeneous and regular computations, such as many matrix computations where the entries are floating point numbers or other fixed sized data, good practical results have been obtained on the Sequent and similar machines [1]. Near linear speedup (one $p$-th the time with $p$ processors) is to be expected.

On the other hand, most computer algebra data are sparse and highly heterogeneous leading to difficulties in parallelization. For example, the standard matrix multiplication is $O(n^3)$ when the entries are homogeneous and the entry arithmetic steps require constant time. In this case the computation is readily partitioned among $p$ processors. But when the entries are multivariate expressions of widely varying degrees and sparsities, a balanced partitioning of the work is very difficult. In this paper we illustrate this problem and sketch a way to achieve effective speedup through a scheduling discipline.

We explain the main features of two implementations of ALDES/SAC2 for parallel computation. We show how, with either implementation, different strategies for using the processors lead to very different results. This is done in the context of a toy problem, that of computing $n!$ for large $n$ (circa 1000, say). We conclude that an adaptive run time scheduling strategy holds promise for the effective use of tens of processors on algebraic computations.

In section 2 we describe the main features of the two implementations of ALDES/SAC2 that we have done on the Sequent Symmetry shared memory multiprocessor. In section 3 several approaches to parallel computation of the factorial function (for large integers) are described. Then, in section 4, these are tested. Timings are reported and discussed. We sum up with a brief concluding section.

# 2 Parallelizing ALDES/SAC2, The implementations

The Sequent computer is a multiprocessor with shared memory and peripherals under the control of one integrated operating system. Each CPU is a general-purpose 32-bit microprocessor. The Sequent system is tightly coupled, all processors share a single pool of memory, the communication among the processors and sharing of resources is carried out through a single shared memory. Processors, memory modules, and I/O controllers are connected through a high speed common bus. The processors are identical and each can execute user and operating system code. Programs written for single processor system can be run on the Sequent with no modification.

An application program can consist of multiple processes all accessing the same data structures in memory. There is support for mutual exclusion. The system allows the use of hardware to lock any section of the memory.

For this machine, we have developed two distinct parallel implementations of ALDES/SAC2. Each is described in detail below. The first, which we call PSSS (Parallel Stacks, Shared Space), uses the shared memory feature of the machine for all list storage. Parallelization is done using Sequent's C and Fortran callable parallel processing primitives, the basic element of which is the "mfork" [8] routine. The second implementation, AL (Aldes Linda), operates by means of separate and independent Aldes/Sac2 processes which communicate only through Linda's "tuple space" mechanism.

## 2.1 PSSS

The first step in this effort was to adapt the SAC2 library for parallel execution on multiple processors. The library is written in ALDES, which is translated into FORTRAN. The FORTRAN versions are then compiled and archived appropriately on a given machine and file system. At run time the routines call each other in various patterns, including recursive calls. Parameters and other variables manipulated for the most part are list structures representing algebraic entities. To handle the situation, variables are of type integer from FORTRAN's point of view. They serve as pointers to list cells. In SAC2 this means they are indices for the global shared array `SPACE`.

The other fundamental data structure of the system is the `STACK` array. The `STACK`, for each routine in the library, holds variables which must be subject to garbage collection. In SAC2 parlance, these variables are "unsafe". The `SPACE` array holds the list cells which are the basic units of all the data manipulated by procedures in the system. The basic need to adapt to the multiprocessor environment is to arrange for a stack for each process, while maintaining a common SPACE in shared memory. The stack for each process could be a private local entity, but we chose to make the STACK a global common entity like SPACE. For multiprocessing it is partitioned into segments, one for each process. If the STACK array has $n$ elements unused at the time of multiprocess forking, and the machine has $p$ processors, then each processor operates with its stack being a segment of length $n/p$ in the global STACK array. There are two reasons for this choice. For one, a side effect of the scheme is that any process can access any other processor's stack. This capability is used in some versions of garbage collection with which we have been experimenting. Secondly, if each processor is to have a private local stack, there must be an effective way to share that stack with subroutines. A subroutine must share $P$'s stack when called by $P$ and $Q$'s stack when called by $Q$. We didn't feel we had an appropriate (relatively machine independent) mechanism in the FORTRAN available to us on the Sequent to handle this situation.

An additional consideration in the stack implementation is the handling of the current stack pointer, INDEX. In the parallel version there must be an INDEX for each processor. We handle this by creating an array PINDX of $p$ stack pointers, one for each process. Subroutines in the library call IUP when they start. This updates the INDEX appropriately. We replace this with PIUP, which uses the processor id to access and update the appropriate entry in PINDX and also set a *local* variable INDEX. Then the references to INDEX in the remainder of the code are valid and the number of modifications needed is kept to a minimum.

Since a global SPACE array is constantly accessed and modified by all processes, there are, of course, some subroutines where greater adjustment is required to handle the mutual exclusion requirements and avoid racing conditions. In particular, routines which modify list structure (routines which write on SPACE) must be written carefully. Chief among these is COMP (the SAC2 equivalent of Lisp CONS). Mutual exclusion must be assured in the section in which COMP detaches a cell from the available cell list. Secondly, if no cells are available, garbage collection must be initiated, a process which affects the whole computation and involves all processors. At least, this is the case for SAC2's simple mark and sweep garbage collector where we make no attempt to do incremental or local garbage collection.

Our approach to garbage collection depends on the assumption that all processes will access the available cell list often (usually through `COMP`). There is a barrier at the beginning of garbage collection. The assumption is that processors can afford to spin at this barrier because other processors will soon also need a new cell, discover there are none, and join the earlier arrived ones at the barrier. When all processors have arrived, garbage collection commences. Our first implementation ignored one important situation in which this assumption (that all processes will seek new cells) is invalid. This situation occurs when, near the end of the parallel computation, some processes find no work left to do and exit. It is not hard to see that such a process will indeed not seek new cells. Our solution to this is greedy. We have finished processors spin on a GC flag. When an unfinished processor discovers the need for garbage collection, it raises the flag, and finished processors join the active ones at the barrier and participate in garbage collection. Our data indicates that this solution works well [7].

Garbage collection itself is done in two phases separated by a barrier. First each processor marks cells reachable from its `STACK` segment, then spins at a barrier until all processes are through marking. Next, each processor sweeps an assigned region in `SPACE`, and links the available cells it obtains into the global available cell list. The synchronization for the latter is managed by a lock.

Parallel performance can be heavily dependent on communication costs. In our situation the most uncertain and difficult to analyze or predict are the effects of `SPACE` array accesses. Each processor has a local cache. A high proportion of cache hits among the memory accesses is very valuable. In addition, there can be considerable contention on the bus for some memory access patterns.

## 2.2 AL

The AL (Aldes - Linda) parallel implementation of ALDES uses linda for communication between self standing SAC2 sessions. The main program is a C-Linda program which creates a command line parameter specified number of processes each running an ALDES main program. Each process has its own STACK and SPACE arrays, evokes BEGINn for itself, etc. No ALDES/SAC2 structures are shared. However each process may use the Linda inspired primitives TAKE, GIVE, COPYIN, COPYOUT, EXISTS. These correspond to the linda primitives in, out, rd, -, inp.

- L = TAKE(I)

  Similar effect to Linda's "in(I,?L)". That is, L is assigned a value that was stored in the tuple space in association with the value of I. If no such tuple exists, the calling process waits until one is created by another process. But also, here L may be any SAC2 object, and if it is extensive, it will have been stored in several tuples. The structure as represented in tuples is transferred into the calling process's own SPACE array in the standard SAC2 representation, L indexing the lead cell of the structure.

- GIVE(I,L)

  Opposite of TAKE, similar to Linda's "out(I, L)". SAC2 structure L is placed in tuples in association with the index value I. The user should assume that the structure L in the SPACE array is removed (see COPYOUT).

- L = COPYIN(I)

  Similar to TAKE, but the structure L in tuple space is not removed, only copied in the calling process SPACE array.

- COPYOUT(I,L)

  Similar to GIVE, but the structure L in the calling process SPACE array is not removed, only copied in the tuple space.

- B = EXISTS(I)

  Motivated by Linda's "inp". B is set to 1 (true) if there is an object L stored in the tuple space in association with I, otherwise B is set to 0 (false). In neither case does the calling process wait.

Uses of these primitives are much like Linda usage. See [1, 2, 3] for examples of Linda-style parallel programming.

# 3 Approaches to Parallelization of the Factorial function

We have compared several approaches to the use of parallelism in computing the factorial of large numbers.

A straightforward implementation of factorial might be as follows:

```
               C = FACT(N)
[Factorial of n, where n is a BETA digit.]
[ C is N!, a positive integer]

(1) C := 1; for k = 1, ..., N do C := IDPR(C, k); return
```

Here BETA is the SAC2 parameter which bounds the size of small (word size) integers, say $2^{29}$, and an integer variable such as C is understood to be a "bignum", i.e. list of BETA digits if necessary. In our tests, for instance, N is 1200 and C is quite large. Also IDPR is a procedure for multiplying an integer, whether bignum or single BETA digit, by a BETA digit.

Using P processors, the obvious parallelization is to let each processor do N/P of the loop iterations. Thus each processor executes a loop of this form, where i is the index of the processor.

```
(1) [strip mine] C[i] := 1;
    for k = i, i+P, ..., N do C[i] := IDPR(C[i], k);
```

This "strip mining" of the loop, an instance of static scheduling, leads to P partial products. These partial products can be multiplied 2 at a time using P/2 processors, and continuing in that style. We choose to implement this phase of the computation using dynamic scheduling. Here RESULT is a shared, lockable variable but S and temp are local to the process. Each process executes the following code segment. When all have finished, RESULT holds the final result. RESULT is initially 0, a conventional value to indicate no part of the product is stored there.

```
(2) [Combine results]
   lock RESULT;                  begin critical section
   while RESULT # 0 do begin
      temp := RESULT;
      RESULT := 0;
      unlock RESULT;             end of critical section
      S := S*temp;               local bignum multiply
      lock RESULT;               begin of critical section
                                 for next iteration
   end
   RESULT := S;
   unlock RESULT;                end of critical section
                                 begun in loop
   return;
```

We will refer to the above described approach as the "coarse grained" implementation.

As the partial products are combined, fewer and fewer processors each multiply larger and larger numbers. We found that the last multiplication, done on only one processor, can easily dominate the entire computation. To address this problem, we implemented a version that parallelizes the later multiplications using available idle processors. Specifically, the dynamically scheduled accumulation phase uses the Karatsuba [5] divide and conquer approach in which each multiplication is split into 3 multiplications of numbers of half the length. These 3 multiplications are done in parallel to the extent that processors are available. We call this implementation "hybrid grained".

These approaches were also implemented in AL so that we can compare the effect of the different demand on shared memory that arises from the different implementations of SAC2. In addition, we implemented an "extended hybrid grained" version to be used with 4 processors. This version uses the 3 processor parallel Karatsuba multiplications to combine the 4 initial partial products from the strip mining phase into 2 partial products, as in the hybrid grained approach. However the last multiplications is parallelized to two levels, resulting in 9 multiplications to be performed on numbers one fourth the length of the inputs.

Finally we tried an approach that ignores the opportunity for parallelism at the coarsest level, the factorial loop, and instead parallelizes at a finer grain, the loop in IDPR. This procedure multiplies a bignum by a BETA digit, which involves multiplying each BETA digit of the bignum by the given BETA digit and then adding in the appropriate way. To parallelize, we break the bignum into P segments, multiply each on a different processor by the BETA digit (using IDPR), and finally add the results with appropriate shifts on a single processor.

- For master processor

```
A:=1;
for b=2,...,N do
    if A < (p BETA digit number) then A:=IDPR(A,b)
```

```
else
    - Find length of bignum A
    - Divide bignum A into p segments
    - Provide tasks to slave processors by storing
      p-1 segments, seg(2) through seg(p),
      with their position,pos(2) through pos(p),
      within A and operand b in Linda's Tuple Space
    - C:=IDPR(seg(1),b)
    - Add up all partial products from Tuple Space
```

- For slaves

```
do forever
    - Get task from Tuple Space
    - C:=IDPR(seg(i),b)   [seg, i, b are from task]
    - Shift C pos(i) times
    - Store C in Tuple Space
```

We call this version the "finer grained" implementation.

# 4 Timings and discussion

We ran each of the parallelizations of factorial described in section 3 with 1 to 7 processors. This with the exception of the extended hybrid grained implementation, which was designed for exactly 4 processors. The table shows the timings in seconds that resulted. Times are from an instant after the processes are created but before any computation begins to the end of all computation.

A number of conclusions can be drawn from this data. First, it may be surprising to see superlinear speedup for the coarse grained and hybrid grained implementations. This is due to the fact that these approaches "inadvertently" lead to a better sequence of multiplications. For example with two processors, one processor multiplies the odd numbers up to 1200 and the other multiplies

| Method | Time | | | | | | |
|---|---|---|---|---|---|---|---|
| | No. of processors | | | | | | |
| | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| PSSS | | | | | | | |
| coarse grained | 12.38 | 5.14 | 3.52 | 2.93 | 2.80 | 2.82 | 2.66 |
| hybrid grained | 12.37 | 4.56 | 2.65 | 1.90 | 2.12 | 1.88 | 1.67 |
| AL | | | | | | | |
| coarse grained | 12.62 | 5.39 | 3.82 | 3.24 | 2.96 | 2.91 | 2.64 |
| hybrid grained | 12.84 | 5.01 | 3.43 | 2.45 | 2.11 | 2.09 | 1.87 |
| extended hybrid | | | | 2.37 | | | |
| finer grained | 17.35 | 185.10 | 231.86 | 292.27 | 355.36 | 451.32 | 544.80 |

Figure 1: Timing table

the even numbers. Then these two large numbers are multiplied. This approach is more efficient even on a single processor, which computes the even and odd subproducts sequentially. In fact we measured that approach using AL, getting 8.62 seconds. The speedup of two processors, with a time of 5.39, over that time is only 1.6. This also points up the need for a good analysis of one's algorithm, since a parallel version, no matter how simply implemented, is a different algorithm. In this case the parallel version has pointed out that the sequential algorithm is suboptimal. Good analytical tools are necessary to find the best approach for computation either sequentially or in parallel.

Second the times for the PSSS implementation are slightly smaller than the times for AL on the two variants implemented on both. The difference varies from essentially none to 29% in the case of 4 processors, hybrid grained implementation. This suggests that for this problem the overhead of the linda database accesses outweighed the higher costs of shared memory access over the bus due to the less independent processes in PSSS. This in contrast to our experience on another problem, the CAD algorithm [7].

The issue we most wanted to examine is the loss of effective parallelism in the coarse grained implementation, because the last multiplications of large numbers are done on one or two processors with the others sitting idle. This is clearly seen. No significant speedup occurs on PSSS for the coarse grained approach with the addition of processors beyond the 4th. Even on Al this approach gets a speedup of $3.24/2.64 = 1.23$ in going from 4 to 7 processors where $7/4 = 1.75$ is hoped for. Improving on this is the motivation for trying a hybrid approach.

For the hybrid approach we find better performance for all numbers of processors 2 to 7. This is because of better use of processors, less idle time, in the later large multiplications. Even this implementation tends to lose efficiency of speedup for the larger numbers of processors, and we suggest that this is due to the limited parallelization of the multiplication function. Indeed, our extended hybrid grained implementation, which further refines the parallelism on the largest multiplication, makes an additional speedup in the time. This suggests that a more general implementation of an adaptive strategy, one that creates parallel versions of activity further nested in the computation when processors are free and refrains from doing so when they are not, will be successful.

That latter observation is important, that we should refrain from parallelizing at a deeper level of nesting, when to do so would produce many more tasks than processors. Each task involves communication overhead and this can overwhelm the computation. This is clearly shown by the last row of data, the timings for an implementation that parallelizes at a too fine granularity, one at which communication overhead completely dominates. Similar results have been observed, for example, for matrix multiplication [1].

# 5 Conclusion

The algorithms used in algebra systems have a good deal of inherent parallelism. In addition many parallel algorithms have been designed but not implemented.

To take advantage of these possibilities for parallel computation we will need suitable systems for exploiting the capabilities of various parallel architectures. Much experience in parallel computation shows greatest success when the computation is regular and homogeneous in detail, as in computation on matrices with floating point entries. In other contexts that share algebraic computation's properties of irregular and heterogeneous detail, parallelization has not been as successful. For many architectures it is demonstrated that greatest efficiency is obtained when the granularity is as coarse as possible. This experiment shows that that idea has to be interpreted carefully. To keep the computation at a granularity appropriate to the number of processing units will require adaptive scheduling, because the proper degree of granularity in the process of the computation does not correspond to a fixed place in the text of the program. We also see the need for algorithm analysis tools to help predict for what size problem parallel scheduling can achieve a gain.

We conclude that there is a good chance for success if the algorithms are expressed in a way that makes clear the opportunities for concurrent computation and the run time system is adaptive, i.e., able to schedule a task for parallel or sequential computation according to the availability of processors and knowledge of parallel and sequential performance of the algorithm.

# References


[1] Ahuja, S., Carriero, N., Gelernter, D.: "Linda and Friends," *IEEE Computer*, 19, No. 8, pp. 26-34, 1986.

[2] Carriero, N., Gelernter, D.: "How to write parallel programs—A guide to the perplexed," Comp. Sci. Tech Report No. DCS/RR-628, Yale University, May, 1988.

[3] Carriero, N., Gelernter, D.: "Application Experience with Linda," In em Proc. of the ACM Symposium on Parallel Programming, July, 1988.

[4] Gathen, J. von zur: "Parallel algorithms for algebraic problems," *SIAM J. Comp.*, 13, pp. 802-824, 1984.

[5] Karatsuba, A., Ofman, Y.: "Multiplication of Multidigit Numbers on Automata," *Dokl. Akad. Nauk SSSR* 145, pp. 293-294, 1962, (In Russian). [ English translation in Soviet Physics – Doklady 7, pp. 595-596, 1963.]

[6] Loos, R. G. K.: "The Algorithm Description Language ALDES (Report)," *SIGSAM Bulletin*, vol. 10, No. 1, pp. 15-39, 1976.

[7] Saunders, B. D., Lee, H. R., Abdali, S. K.: "A Parallel Implementation of the Cylindrical Algebraic Decomposition Algorithm," *Proc. ISSAC'89*, pp. 298-307 , 1989.

[8] Sequent Computer Co.: *A Guide to Parallel Programming*, 2nd ed., Beaverton, OR., 1987