Some Experiments with Parallel Bignum Arithmetic†

Bruce Char and Jeremy Johnson
Department of Mathematics and Computer Science, Drexel University
Philadelphia, PA 19104
David Saunders and Andrew P. Wack
Department of Computer and Information Science, University of Delaware
Newark, DE 19176

Abstract: This paper presents empirical data from the implementation of several algorithms for parallel integer arithmetic on a network of workstations. Algorithms were implemented using Sugarbush, a parallel version of the Maple computer algebra system, which uses C/Linda for parallel operations. The Sugarbush implementation is comprared to a specialized C/Linda implementation. Timings are provided for a network of 27 workstations.

Keywords: parallel computer algebra systems, parallel integer arithmetic, Karatsuba's algorithm, distributed computation

† This material is based on work supported by the National Science Foundation under NSF Grant No. CCR-9200982

1 Introduction

This paper reports on an empirical study of several parallel implementations of arbitrary precision integer multiplication. To date we have investigated several parallel implementations of Karatsuba's algorithm on a network of workstations (our experiments consider up to 27 processors). Our implementations were done using Sugarbush 1.2[4], a parallel version of the Maple computer algebra system [5, 6] which uses C/Linda [3] operations to support parallelism. In addition we compared the Sugarbush implementation to several implementations coded directly in C/Linda. The C/Linda implementations allowed us to study some of the overhead incurred by Maple and Sugarbush and also allowed us to more easily study different parallel scheduling strategies.

The study is analogous to the work done by Küchlin, Lutz, and Nevin in [8], where they studied parallel implementations of integer arithmetic on shared memory multiprocessors. New challenges arise when one attempts to effectively utilize a larger number of processors: obtaining effective efficiency for a larger number of processors can be quite challenging. Moreover, implementations which behave similarly for a small number of processors can behave quite differently when the number of processors is increased.

The original motivation for this study was a challenge to compute one million digits of $\sqrt{2}$. We were given Maple code [7] and wanted to see how easily Sugarbush could be used to parallelize this code and how effective the Sugarbush implementation would be in reducing the computing time for this computation. Most of the time spent in this computation was devoted to integer multiplication. The Maple implementation used Karatsuba's algorithm for integer multiplication. Using a parallel version of Katasuba's algorithm we obtained a 10 fold speedup using 20 workstations. While this speedup was encouraging, there was plenty of room for additional improvement. This led us to a more detailed study of parallel integer arithmetic in general and Karatsuba's algorithm in particular. We wanted to investigate how to obtain better efficiency and the scalability of various algorithms for integer multiplication.

We believe Karatsuba's algorithm is a good place to begin our study:

1. The divide and conquer structure of the Karatsuba multiplication algorithm is similar to many computations in computer algebra, such as Chinese remaindering, evaluation/interpolation, and block matrix operations. From these experiments with integer arithmetic we will suggest tools for expressing parallel algorithms in a way that encapsulates the issues and tradeoffs concerning the effective use of various distributed computing environments on the computation.

2. Candidate strategies for integer multiplication include the fft based, the divide and conquer Karatsuba algorithm studied here, and the classical algorithm. The threshold at which the Karatsuba approach is faster than classical is below the threshold at which parallelism can be effectively used in the distributed environments we target, especially networks. On the other hand, the fft based arithmetic is clearly superior for large enough integers. We believe the tradeoff point between these methods is within the range where parallelism is effective. Thus this work is also a step toward finding the right balance between Karatsuba and fft based multiplication for parallel computation with integers of all sizes.

An overview of Linda and Sugarbush is given in Section 2 and an overview of Karatsuba's algorithm and various parallel implementations and scheduling strategies are given in Section 4. The use of Karatsuba's algorithm in the computation of one million digits of $\sqrt{2}$ is described in Section 3. Empirical data is provided in Section 5. Tentative conclusions are given in Section **??**. The conclusions are tentative since we have not been able to fully explain all of our empirical work and since aspects of this study are still in progress. Additional work in progress is outlined in the conclusion Section **??**.

2 Sugarbush and Linda

2.1 Linda in C

Linda is an approach for adding parallelism to conventional procedural programming languages developed by Gelernter et al. [3]. We review Linda's important features briefly.

Central to the operation of any Linda-language is the concept of *tuple space*. Tuple space is a kind of virtual shared memory. Parallel Linda processes add data tuples from tuple space through the procedure *out(tuple)*, and extract it with the operation *in (tuple)* and read (without extracting it) with *rd (tuple)*. *inp* and *rdp* are non-blocking versions of *in* and *rd*. *in rd inp* and *rdp* allow simple pattern matching so that tuples with unknown or unspecified fields can be retrieved. A final Linda operation *eval(expression tuple)* launches one or more new processes which evaluates the elements of the tuple, placing the result in tuple space when the processes terminate.

C/Linda is probably the most widely used Linda language. C/Linda is C with the six Linda operations. The commercially supported version of C/Linda [1] allows into tuple space basic data objects such as integers, floats, and characters (and arrays and structures thereof). It also comes with a parallel simulator and run-time debugging system called the TupleScope which displays tuple operations as they occur (in the simulator) or from postmortem trace files.

Linda's attractiveness is in its simplicity and conceptual elegance. A C/Linda user can foist tuple space management off on the language implementor. This has its good aspects:

- It makes the implementation of the worker/task pool paradigm of distributed computation discussed in section 2.2 particularly easy; just inserting and retrieving task tuples to/from tuple space.

- C/Linda has commercial support and a non-trivial user community. This frees the parallel algebra system implementor from debugging parallel functionality below the tuple space level. It also means that the efficiency of basic interprocess communications (especially vis a vis other parallel programming environments such as PVM) will have been tuned to pass non-trivial scrutiny.

- The same language constructs can be made to work on a variety of machines: shared memory, multicomputer, or networks. While this does not relieve the necessity for tuning for performance, it does eliminate the necessity of recoding for different machines. Since the language specifications for Linda do not specify the precise details of how the tuple space manager is to be implemented, this allows different implementations that may take advantage of characteristics of the particular computation environment, or of additional insight about managing for performance.

On the other hand, working with C/Linda presents the following drawbacks:

- Because neither the Linda specification nor the SCA documentation provides details of how Tuple Space is implemented, the behavior of the system is difficult to predict except the results available through experimentation. There aren't enough conceptual "hooks" in the commercial C/Linda to allow easy decision-making when there are a number of plausible alternatives to programming a solution. This also makes it difficult to make certain judgments about the scalability of certain approaches.

2.2 Sugarbush

Sugarbush is a version of parallel Maple created by adding Linda operations to those available within ordinary Maple. The project started in 1989 when the first version of the system was constructed, running on a Sequent Balance (shared memory system with 8 processors). In addition to the Sequent Balance and Symmetry, Sugarbush has been ported to networks of DEC 5100, and Sun 4 (with Sun OS) workstations. The latest version is Sugarbush 1.2[4] Version 1.2 uses C/Linda (version 2.5.2 from Scientific Computing Associates) and the commercially available OEM version of Maple V release 2.

To exchange Maple expressions between processes, Maple expressions are converted between the dag representation used by Maple internally, and a printable-character string encoding for interprocess communication. The character string is a linearized form of the dag, with back pointers to previous occurrences of subexpressions within the dag. It is the same form as that used to save Maple expressions within Maple worksheets.

The basic philosophy behind the parallel operation of Sugarbush is that of "workers with a task pool". All but the master process in the ensemble of parallel Maple processes running within Sugarbush are running a Maple procedure called `worker` which scans Linda tuple space for tuples tagged as "Maple work". Upon finding work, the worker will perform it and return the result back to tuple space where it will be picked up eventually by whichever process ordered the work performed.

The worker/task pool programming at this point in the evolution of Sugarbush design has little machinery for explicitly assigning workers from the pool to particular tasks. Any available worker is free to take on any task which is placed in the pool.

In addition to the basic Linda operations, Sugarbush provides a number of higher-level parallel operations. Two which are worthy of mention here are *parmap* and *parsplit*. *parmap*(*f*, *expr*, *other args*) is a parallel version of Maple's *map* (similar to Lisp *mapcar*. Each component of work in the map is placed into the task pool for a worker to do. The invoking process then recursively calls the *worker* code so that it can participate in the subcomputation (or whatever work is in the pool at that point). When all the work of the map is done, the worker code of the invoking process when it finishes the task it is working on. Control then returns to the invoking code, where the results are collected. Control then proceeds to the next step of the computation after the *parmap*.

parsplit is similar to *parmap* except that there are provisions for calling auxiliary functions that decide whether to parallelize the map work, or to perform it sequentially within the invoking process. Using standard Maple coding conventions, when *parsplit* is invoked with first argument *f*, it will invoke *SplitWork* with the rest of the arguments to the map. *SplitWork* is loaded automatically within Sugarbush processes. It will check if the user has defined the procedure *SplitWork/f* within the process. If so, it will invoke that function with the rest of the arguments to the map. This provides an opportunity for user-defined decision making as to parallel or sequential execution of the operation.

3 A test problem: computing a million digits of sqrt2 in Maple

By 1993, development of Sugarbush had reached the point where we were interested in exploring the characteristics of the system on problems not originating within our group. One of these was Big Prime Berlekamp polynomial factorization as first programmed by Monagan [9] to solve some of the the von zur Gathen problems. The parallelization of Big Prime Berlekamp is discussed in part by another paper in these Proceedings [2]. Another was problem was the demonstration by Gourdon and Salvy [7] of finding a million digits of the $\sqrt{2}$ in Maple.

Gourdon and Salvy use a modified Newton iteration to calculate their result:

$$u_0 = 7071067811865475, u_{n+1} = \frac{3u_nL_n}{22} - \frac{{u_n}^3}{L_n}, \quad L_n = 10^{2^{n+4}}.$$

The point of the modification is to allow all calculations to be done in integer rather than big floating point arithmetic, with shifting (Maple uses a radix of 10^5 for its integers) instead of division. This calculation was not completely trivial due to the following complications solved by Gourdon and Salvy:

- Maple V release 2 has only classical integer multiplication implemented, which is inefficient for the size of operands encountered in the iteration. Gourdon and Salvy implemented a version of Karatsuba multiplication [7]. In part because of the differential between interpreted and internal operations in the programming language, Gourdon and Salvy set the cutoff point between classical and Karatsuba to be 6000 digits. Because a number of multiplications are squarings, Karatsuba code to handle this special case of multiplication was also included.

- Maple (on 32 bit architectures) is unable to represent integers more than approximately 512,000 decimal digits long. Thus the last two iterations had to be coded specially to handle the extra-large integers as lists of representable integers.

Profiling the computation in Maple indicates that roughly two thirds of the computation time (approximately 26000 seconds on a Sun Sparc IPC running Sun OS 4.1 at the University of Delaware) is taken by the last two iterations. As expected, almost all of the computation time is spent doing Karatsuba operations. We discovered that the code permitted concurrent computation of several Karatsuba multiplications at once. The size of the operands and the times for the operations were such that it was easy to saturate the number of processors we had available (27 homogeneous Sun IPC workstations at the University of Delaware).

While this problem was meant to provide a demonstration of the suitability of Sugarbush to handle the parallelization of the Gourdon-Salvy code, the Karatsuba subcomputation has provided an interesting test example for considerations about scheduling and load-balancing of the recursive, divide-and-conquer computations often found in computer algebra. The next sections of this paper discuss parallelization of the Karatsuba operations. We return to the parallelization of this problem within Sugarbush in Section 6.

4 Karatsuba's Algorithm

The algorithm **KPROD** outlines Karatsuba's algorithm for integer multiplication. For simplicity, we assume the numbers to be multiplied are positive and of the same size. When one is substantially larger than the other, eg. more than twice as long, an adjustment should be made.

$$C \leftarrow \mathbf{KPROD}(A, B, n)$$

[Karatsuba's Algorithm.
Input: positive n digit integers A, B.
Output: the $2n$ digit product $C = A * B$.]

1. If n is sufficiently small, use the classical multiplication.

2. [Split A and B in half.]
Let $k = \lceil n/2 \rceil$ and let α be 10^k;
$A = A_1\alpha + A_0$ and $B = B_1\alpha + B_0$, where A_1, A_0, B_1, B_0 are less than α.

3. [Compute .]
$C_0 = \mathbf{KPROD}(A_0, B_0, k)$;
$C_1 = \mathbf{KPROD}(A_0 + A_1, B_0 + B_1, k + 1)$;
$C_2 = \mathbf{KPROD}(A_1, B_1, k)$.

4. Return $C = C_2\alpha^2 + (C_1 - C_0 - C_2)\alpha + C_0$ ∎

Recall that the sequential computing time of Karatsuba's algorithm, $T(n)$, where n is the number of digits in the two integers that are being multiplied, satisfies the recurrence $T(n) = 3T(n/2) + \Theta(n)$, The solution to this recurrence is $\Theta(n^{\log_2(3)})$, where $\log_2(3) \approx 1.58$. With sufficiently many processors, the parallel computing time, $T^*(n)$, satisfies the recurrence $T^*(n) = T^*(n/2) + \Theta(n) = Theta(n)$. More specifically, with p processors, the parallel computing time is $\Theta(\lceil n^{\log_2(3)}/p\rceil + n)$. If $p = \Omega(n^{\log_2(3)-1})$ the parallel time is $\Theta(n)$, and linear speedup can be obtained for $p = O((n^{\log_2(3)-1})$.

We remark that when the numbers are large the space demands can be significant. This presentation masks the space issues. In a nutshell, space is needed for the $2n$ digit answer C and also temporary space is needed for the sums input to the computation of C_1 and for the differences involved in the final computation. In a sequential computation the use of temporary space can be limited to that for an additional $2n$ digits, whereas when the recursive calls in step 2 are done in parallel, we require slightly more than $3n$ digits of temporary space.

It is convenient to view this algorithm in terms of a ternary tree. Each interior node has three children, the computations of the C_i. The leaves are the classical multiplications. The work before and after the recursive calls correspond to activities at the interior nodes. On "the way down" two additions of n digit numbers are required in preparation for the computation of C_1. On "the way up", three addition/subtractions of approximately $2n$ digit numbers are involved.

Our three parallel implementations all use the workerpool approach to parallelism. Each worker process loops on "take a task, perform it (possibly generating additional tasks), return a result", where "take a task" and "return a result" are implemented as getting and putting tuples in Linda's tuplespace. The three implementations associate tasks with nodes and subtrees of the tree in different ways. The Sugarbush implementation assigns a task to each node of the tree, down to the level at which the number of digits n is less than a threshold (8000). The subtrees at that level are of depth 0 or 1, because the threshold for conversion to the classical multiplication is at 6000 digits. In the *flat* implementation a master process handles all the nodes down to a command line specified level. A worker task is to compute a specific subtree at this level sequentially. The master first traverses the tree to generate tasks, then traverses it to collect and combine worker results. This involves less communication than the SB implementation, but also does not allow for parallel computation of the additions/subtractions associated with the nodes at the top of the tree. To address that issue, we finally have the *fork-join* variant in which a task is to 1. fork tasks to compute C_0 and C_2 2. locally compute C_1, but (if $n/2$ is above the threshold) recursively fork tasks to compute 2 of the three size $n/4$ products, etc. 3. get the results, c_0 and C_2 from the tasks forked in step 1 and combine with the locally produced C_1.

The latter two approaches were implemented directly in C-Linda for several reasons. Special purpose implementations often are a factor of 5 to 10 faster than implementation of the same method in general purpose systems such as Maple and Mathematica. We wished to see if this is true regarding the parallel SB situation. More importantly, we wished to compare the effectiveness of different partitions of the computation tree into tasks, and do it in the simpler context of direct C code. And finally we wished to compare the effectiveness of a parallelization realized directly in SB versus the approach in which SB serves as a front end to C code (including Saclib based code). We have a SB to Saclib interface to support this approach. In the long run, the parallel computer algebra system we envision will undoubtedly include components using each approach. Among other things, this experimentation is helping us determine how to decide when the additional work of a direct C coded unit is appropriate.

Since the drop in parallel efficiency with increasing processors is somewhat greater with the SB tasking (see data section below), it would be interesting to implement that tasking approach in C for direct comparison with the other two. We also are considering variants in which groups of nodes of the tree are combined into tasks in a more sophisticated way by Wack's scheduling algorithm, [10]. This takes into careful account the times at which task results will be available, in view of the compute times and communication times involved. For such schedules to make a significant difference, the processors must be fairly well synchronized. Thus far our experiments indicate that unpredictable variations in the fraction of the cpu cycles we get on each processor, and perhaps also the vagaries of network load, conspire to effectively mask the benefit of careful scheduling on this problem.

5 Empirical Data

We have timed performance of the three implementations discussed above on a network of workstations. The network is ethernet and the workstations are Sun IPC's, sparcs running at 25mhz, with a specint rating of about 14.

Each number recorded in Table 1 is the best time seen for the given problem and number of processors. The best is taken on the grounds that it represents the performance one could expect on a dedicated network, that the longer times are due to external factors. We are not 100 percent sure of this claim, however, in view of the details of our experiences with the network.

Most runs were done at night when the machines were lightly loaded. Even so, times much worse than those reported occured fairly often. The times were much more variable when 27 processors were used than in the cases where 1, 3, or 9 processors were applied. In the latter cases most times were within a few percent of the best time. However the times for 27 processors varied considerably from a few percent more to several multiples of the the best times. Occasional outliers as many as 10 times the best were seen. We explored this matter in the case of the "flat" implementation. Start and end time for each worker were recorded, and the master recorded the time of receipt of each result. Two phenomena accounted for the suboptimal times. Sometimes an individual worker would be slow to complete it's task. This is evidently due to another activity on that processor. Our fraction of the cpu was normally over 90%, but in these cases of a slow worker it fell to closer to 50% for that processor. The second phenomena, more damaging to performance and more puzzling, was that in some cases the worker result was ready when expected, but the master failed to receive the result for an extended period of time. In one case we have evidence that no progress was made in our computation for 9 minutes until the master finally succeeded in receiving the expected data. At this point we do not know what causes this problem. We suspect everything from network problems (repeated loss of certain UDP packets?) to Linda bugs (inappropriate response to loss of a packet?). We will explore this matter further, in the weeks ahead.

The following is a summary of the best times recorded for each implementation when the number of processors is a power of three. The heading "Sug" denotes the Sugarbush implementation using nested calls to `parmap`. This implementation descends the ternary tree for Karastuba multiplication to the level at which the number lengths are below a threshold. In contrast the two C implementations descend the tree to just that level at which the number of nodes is the number of processors in use. The column header "f-j" denotes the fork-join strategy. Each process, except at leaves of the ternary tree of Karatsuba computation, `out`'s 2 tasks, recursively calls itself on the third task, then `in`'s 2 results and adds together. In the "flat" approach, workers deal only in tasks at a fixed level of the tree. They perform a single sequential multiplication, and do not spawn additional tasks. A master process (3^k + 1st proc) walks the top of the tree first to set tasks, then to collect and combine results.

The C sequential times are about a third the Maple times. Maple (or other computer algebra system) often costs this much or more over a special purpose implementation. Indeed, factors of 5 or 6 over special purpose code are not uncommon. The low multiple here is probably because the code spends a great deal of time in the Maple kernel. The (interpreted) Karatsuba multiplication reduces to many calls to the basic multiplication in the (compiled and optimized) kernel.

The fork-join approach is designed to parallelize more of the work at the top of the tree without adding communication overhead. Thus analysis suggests that fork-join would always be better than the flat approach. However, The times for the two C implementations hardly differ, suggesting that the vagaries of the network of shared workstations environment mask the small gains expected for fork-join. The times for the two methods on 27 processors do differ, but we attribute this more to the network delays discussed above than to intrinsic differences in the amount of work done in parallel. For this same reason we expect better times are possible on 27 processors, if these mysterious delay problems can be solved. Note that the parallel efficiencies are quite good for up to 9 processors, but are reduced and more variable for 27 processors. For example, the efficiencies are all above 90% for the C implementations with 9 processors and numbers of 64,000 digits or more.

Regarding efficiency of speedup, it is also interesting to note that the speedups drop somewhat faster with number of processors for the Sugarbush implementation. This may be due to the finer task granularity and to the

Digits	Sequential		3 processors			9 processors			27 processors		
	C	Sug	f-j	flat	Sug	f-j	flat	Sug	f-j	flat	Sug
4000	1.05										
8000	3.11										
16000	9.30	25.3			15.7			16.8			13.6
32000	28.0	78.4	9.74	10.7	41.9	3.57	3.69	25.6	4.73	3.40	21.7
64000	84.0	242.	28.6	30.6	111.	10.2	10.2	53.8	6.37	5.92	17.1
128000	252.	741.	88.6	84.8	306.	29.5	29.5	149.	19.3	28.7	92.1
131000									15.9	23.4	
256000	754.	2317	254.	267.	oflo	88.3	87.2	400.	45.3	45.2	179.
512000	2267		757.	760.		259.	259.		120.	153.	

Table 1: Parallel Computing Times for Karatsuba

fact that the general purpose `parmap` tool involves more linda tuplespace operations per task than do the special purpose C implementations. Further study is needed to be sure of the cause, however.

6 Parallelization of the sqrt2 computation in Maple

Gourdon and Salvy's code was parallelized in the following way:

- The recursive calls Karatsuba procedures `karaprod` and `karasqr` were merged into one parallel call through the use of the Sugarbush facilities for task dispersal and result merging. In order to do this, the sequence of some of the operations had to be modified compared to that of the sequential code.
- Because the last two iterations were specially coded by Gourdon and Salvy (see [7] for details) due to Maple's limitations on the size of internally representable integers, similar parallelizations of the Karatsuba operations were made in that code as well.
- Using Sugarbush's facilities for user-specified conditionalization of parallelization, we used a parallelization-cutoff of 8000 decimal digits. For smaller numbers of processors, we also used heuristics to prevented parallelization when all worker processes were busy.

	Sequential time (in seconds)
	26313
Number of Processors	Parallel time
2	15023
3	10678
5	6461
9	4959
15	2442
19	2166
25	1889
27	1748

Table 2: Parallel Computing Times for sqrt2

The times in Table 2 indicate the time required to solve the $\sqrt{2}$ problem on a collection of (relatively) unloaded Sun Sparc IPC workstations at the University of Delaware. Since the workstations are freely available to users at Delaware, there was significant variability in timing (often several hundred seconds' difference). Where variation was noticed, we used an average of several runs.

7 Use of load balancing heuristics in Sugarbush

Optimal scheduling of a set task with a pre-determined number of homogeneous, unloaded processors is one thing; getting plausible performance during a computation chock full of possibility of heterogeneous concurrent tasks on system with other users is another. Sugarbush 1.2 provides a "hook" for user-specified scheduling-on-the-fly, along with a number of simple ways to assess system load, such as a predicate that is true when no workers are idle[1].

We did not consider the original Sugarbush timings to be fully satisfactory using the original cutoff of 8000 decimal digits. Compared to the C programs employing "optimal splitting" for the problem size, the Sugarbush code is "over splitting" — choosing to try to parallelize subtasks when all workers would already be fully occupied executing their given tasks sequentially. We did additional runs with an increased static cutoff of 16000 digits, as well as a run where a Karatsuba task was further parallelized if the problem size was 8000 digits *and* if the Sugarbush *SomeWorkersIdleHeur* predicate returned true. These experiments are reported in Table 3

The heuristic in general seems to have a better effect except in the case of the largest number of processors and the largest problems. We are currently exploring the phenomena involved in this situation, but it appears that a combination of "too small" granularity combined communication overhead leads some processors to reach the bottom of the Karatsuba computation tree ahead of the division of the work for the other large subproblems. Because of this, some of the workers spend most of their time working on small subproblems farmed out by others, rather than the "optimal schedule" of all workers handling their own subproblem of a reasonable size. In future work, we intend to seek inexpensive measures to remedy this situation.

8 Conclusion

This work will continue in the coming weeks, and additional experiments can be expected to sharpen the results. Currently we are unable to provide satisfactory explanations for our timings when a large number of processors were used. We will perform additional experiments to help explain this situation. In addition, we will continue our study by including fft based algorithms, comparing our results to other paradigms and systems for parallel computation, and by providing timings for other parallel computing environments. We intend to compare our results to a message passing implementation using PVM and to a threads based approach on a 128 node KSR computer (a simulated shared memory machine).

Tentatively we conclude that 1. The differences in communication patterns do not have much effect on performance with small numbers of processors for the sizes of numbers which require significant time (minute-plus sequential time, say). 2. The communication overheads take a toll sooner (at a smaller processor number) for the SB context than for the special purpose implementation context. 3. Our experience shows the network performance with many nodes (more than 20, say) can be severely impacted by problems in the communication system. To solve these may require careful control of the task to processor ratio for load balancing and may require attention to robustness (fault-tolerance) to get acceptable performance.

In view of this we propose to encapsulate the attention to these matters in a parallelism tool, analogous to parmap, for dealing with "tree" (recursively expressed) computations. For example if a function is `f(x) := if x < threshold then g(x) else h(f(`p_1`(x)), f(`p_2`(x))) fi`. Then treemap(depth, x, g, h, p_1, p_2) could handle the task. It can then be implemented variously depending on the nature of the distributed computing environment and the number of processors involved.

[1] Since this predicate is based upon the Linda *rdp* procedure, it is meant to be regarded as a heuristic

Acknowledgements

This work is supported in part by the National Science Foundation Grant CCR-9200982. We are grateful to Bruno Salvy and Xavier Gourdon for making their Maple code available to us.

References

[1] Scientific Computing Associates. *C-Linda User's Guide and Reference Manual.* Scientific Computing Associates, 265 Church St. New Haven CT 06510-7010, 22.1 edition edition, April 1993. Corresponds to Original and Network Linda version 2.5.2.

[2] Tracy Burke. The parallel computation of $f(x)^{(p-1)/2} \bmod h(x)$ on a network of workstation-

	9 procs	18 procs	27 procs						
Digits	8000	16000	8000 + Idle	8000	16000	8000 + Idle	8000	16000	8000 + Idle
16000	16.8		16.2	23.0	28.5	19.1	13.6	19.3	14.2
32000	25.6	37.3	24.4	26.1	38.1	28.2	21.7	32.5	32.9
64000	53.8	49.7	63.2	74.4	52.5	48.4	70.8	53.4	49.7
128000	149.2	137.5	126.8	138.6	100.3	106.8	92.1	68.9	86.0
256000	400.3	343.4	319.0	260.2	242.2	241.9	179.1	165.9	185.4

Table 3: Parallel Computing Times using Different Cutoff Criteria

s using Sugarbush 1.1. *Proceedings of PASCO94*, 1994.

[3] Nicholas Carriero and David Gelernter. *How to write parallel programs: a first course.* MIT Press, 1990.

[4] Bruce W. Char. A user's guide to sugarbush — parallel Maple through Linda. Technical Report MCS-94-01, Drexel University Department of Mathematics and Computer Science, May 1994.

[5] Bruce W. Char, Keith O. Geddes, Gaston H. Gonnet, Bento n L. Leong, Michael B. Monagan, and Stephen M. Watt. *Maple V Language Reference Manual.* Springer-Verlag, 1991.

[6] Bruce W. Char, Keith O. Geddes, Gaston H. Gonnet, Bento n L. Leong, Michael B. Monagan, and Stephen M. Watt. *Maple V Library Reference Manual.* Springer-Verlag, 1991.

[7] Xavier Gourdon and Bruno Salvy. Computing one million digits of $\sqrt{2}$. *The Maple Technical Newsletter*, (10):60–71, Fall 1993.

[8] Wolfgang W. Küchlin, David Lutz, and Nicholas J. Nevin. Integer multiplication in PARSAC-2 on stock microprocessors. In *Applied Algebra, Algebraic Algorithms, and Error-Correcting Codes: 9th International Symposium, AAECC-9*, pages 206–217, 1991.

[9] M.B. Monagan. Von zur Gathen's factorization challenge. *SIGSAM Bulletin*, 27(2):13–18, 1993.

[10] A. Wack, 1994. Unpublished Notes.