1984

Report of the Workshop on Supercomputers and Symbolic Computation

Dennis S. Arnon

Report Number:
84-481

REPORT OF THE WORKSHOP ON SUPERCOMPUTERS AND SYMBOLIC COMPUTATION

Edited by
Dennis S. Arnon

CSD-TR-481
June 1984

Report of the Workshop on

SUPERCOMPUTERS AND SYMBOLIC COMPUTATION

held at

Purdue University
West Lafayette, Indiana
May 3-4, 1984

sponsored by the

Purdue Center for Parallel and Vector Computing

edited by

Dennis S. Arnon
Computer Science Department
Purdue University
West Lafayette, Indiana, USA 47907

CSD TR-481
Department of Computer Sciences
Purdue University
June 4, 1984

Support from the Army Research Office made this Workshop possible, and is gratefully acknowledged.

Workshop on Symbolic Computation and Supercomputers

TABLE OF CONTENTS

PREFACE

Section 1: INTRODUCTION. 3

Section 2: THE NATURE OF SYMBOLIC COMPUTATION. 5

Section 3: LANGUAGES. 7

3.1. LISP. 8

3.2. Symbolic computation languages. 8

3.3. Symbolic-numerical interface. 11

Section 4: MEMORY. 12

Section 5: SYSTEM ARCHITECTURE. 15

Section 6: PERFORMANCE EVALUATION. 20

Section 7: GLOSSARY. 22

Section 8: APPENDIX: MAILING ADDRESSES OF PARTICIPANTS. 24

References 27

Preface

This report is the record of a two-day workshop held at Purdue University on high-performance symbolic computation. The title "Symbolic computation and supercomputers" reflects the interests of the participants, many of whom are practitioners of symbolic computing on supercomputer-class machines. The goal of the workshop was to assess the current state of affairs, and identify key issues requiring attention. The report is primarily a compendium of the positions taken, recommendations made, and questions raised by the participants.

The participants were:

John Aldag, Cray Research
Wayne Anderson, Los Alamos Laboratory
Dennis Arnon, Purdue University
Bobby Caviness, University of Delaware
Jagdish Chandra, Army Research Office
Elizabeth Cuthill, David Taylor Naval Ship R&D Center
Alvin Despain, University of California, Berkeley
Robert Douglass, Los Alamos Laboratory
John Fitch, University of Bath, England
Daniel Friedman, Indiana University
Richard Gabriel, Stanford University
Jeffrey Greif, Inference Corporation
Martin Griss, Hewlett-Packard Laboratories
Malcolm Harrison, New York University
Christopher Haynes, Indiana University
Robert Kessler, University of Utah
Wayne Matson, Symbolics, Inc.
Donald Oxley, Texas Instruments
John Rice, Purdue University
John Smit, Goodyear Aerospace Corporation
William Stockwell, Control Data Corporation
Paul Wang, Kent State University

1. INTRODUCTION.

This report is the record of a two-day workshop held at Purdue University on high performance symbolic computation. Twenty-two persons, representing a cross-section of universities and industry, participated in two days of discussions. We chose the title "Symbolic computation and supercomputers" to emphasize the urgency of radical improvements to current symbolic computation performance and availability levels. In the report we attempt to convey the specific and detailed points made by individual participants, as diverse and even contradictory as these sometimes are. Our lack of consensus on many issues is testimony to the need for further clarification and study in this area.

There can be no doubt that high performance symbolic computation is needed today and will be increasingly needed in the future. Symbolic computation is a cornerstone of applied artificial intelligence, e.g. expert systems. DARPA's Strategic Computing initiative [Age83] assigns symbolic computation a prominent role, and says of it:

> current applications in areas such as vision now require about three orders of magnitude more processing than is now available. As future algorithms and applications are developed, even more computing power will be necessary. (p. 44)

Symbolic computation is a large component of the "knowledge information processing" of the Fifth Generation project in Japan [Mot82], as is attention to superscale processors.

The following are some concrete examples of the need for higher performance; the list could be lengthened indefinitely. Speech systems that speak too slowly need speeding up. A program of Moravec [Mor35] guides a robot across a room containing obstacles; currently 15 minutes of computation are required after each step to evaluate what the robot's vision system now sees, and decide in which direction it should take its next step. NASA plans the computation of 30,000 spherical harmonics for the gravity field of the earth. Symbolic computations performed by sophisticated mail programs, e.g. parsing source addresses, sorting messages into conversations, can be unacceptably slow. There

are many potential applications of combined numerical and symbolic computation waiting to be pursued, in which the symbolic computation is the dominant cost (cf. Section 3.3 below). For example, one might generate the equations of motion for joints of robot arms symbolically, then solve them numerically. Current supercomputers already make the numerical component much less costly than formerly; a similar improvement in the symbolic component is both needed and thought to be possible.

In recent years, a number of supercomputers, e.g. CYBER-205, CRAY-1, Denelcor HEP, and Goodyear MPP (Massively Parallel Processor), have been commercially produced. In addition, design and construction are at an advanced stage on such machines as the NYU Ultracomputer, the Stanford University/Livermore Laboratory S-1, and the Columbia University NON-VON. For numerical problems, they have given dramatic, often order of magnitude, performance improvements. A symbolic computation environment that delivers order of magnitude performance improvements over current levels might be termed a "symbolic supercomputer". The question is - how to build one? There is general agreement that very increased use of parallelism is basic. However, while today's supercomputers can deliver substantial speedups on many numerical problems, and may perform well as symbolic processors (e.g. with a LISP implementation), they are not "symbolic supercomputers". Their design has been driven by the requirements of vector processing for large-scale numerical computation.

It is the goal of this report to collect ideas and information pertinent to (1) the design of a "symbolic supercomputer", and (2) the more effective use of existing supercomputers for symbolic computation. At least one participant did not support these objectives; he believed that cost effectiveness of machines is the key issue, i.e. will supercomputers ($1-10M) or powerful personal computers ($1-30K) have the largest positive impact on society?

"Symbolic computation" means different things to different people; we devote Section 2 to its various aspects and key features. We feel there is an urgent need for evolutionary change in general-purpose programming languages for symbolic computation, as we detail in Section 3. Section 4 takes up memory management, a crucial issue which will become more complex and no less important as parallel systems evolve. The topic of Section 5, system organization, may be introduced by a quote from the DARPA report ([Age83], p. 45): "The symbolic processors of the future may well be a collection of special components which are interconnected via a general host computer or by high speed networks". Some of us agree and some do not. Section 6 is concerned with the evaluation of existing and new symbolic computation systems, made increasingly necessary by their proliferation; we make specific methodological recommendations.

We remark that at least one participant felt that we should not spend time arguing architectures or programming languages, but should characterize the specific algorithms we would like speed up, and find out how to do so.

2. THE NATURE OF SYMBOLIC COMPUTATION.

It is unlikely that two persons will agree on a definition of symbolic computation. We list a few possibilities.

(1) DARPA ([Age83], p. 44) says "Symbolic processing deals with non-numeric objects, relationships between these objects, and the ability to infer or deduce new information with the aid of programs which 'reason'."

(2) Kahn [Kah82] says that symbolic computing arises when we have "objects" (e.g. programs, programming languages, circuits) that we need to reason about. The objects are represented as "structured collections of formulae in some well-defined algebraic formalism". The "computer tools" needed to reason about the objects are symbolic computation and theorem-proving.

(3) Buchberger [Buc83] defines symbolic computation as: "all aspects of the algorithmic treatment of symbolic ... objects, where symbolic objects include terms, formulae, programs, geometrical objects ...".

(4) One participant's definition: symbolic computation is characterized by one or more of (A) manipulation of several types of entities (symbols, numbers, sets expressions, lists, rules), (B) application of the same high or low level function to diverse structured objects, functional aggregates of such expressions, and recursive descent through expressions performing operations of this type, (C) (in principle) unbounded memory use in simplification or evaluation of subparts of a computation, (D) searches through (moderately sized) databases in which the matching operation may be complicated, (E) different instructions applied to objects of different type or different objects of the same type (F) random memory references via pointers, (G) application of certain deterministic mathematical processing, e.g arbitrary precision integer arithmetic, or greatest common divisor calculation.

(5) Another participant's definition: Symbolic computation deals with applications that may involve heuristic search, uses comparison of symbols as a basic operation (instead of floating point arithmetic, which is numerical computing, or integer arithmetic, which is signal processing) dealing with certain typical data structures (e.g. semantic nets, frames, productions, etc) involving pattern matching and is very run time dynamic in storage requirements and processing requirements.

(6) One participant observed that the equal presence of algorithms, and heuristic search and pattern matching, seemed to be a characteristic feature of symbolic computation. Another participant observed that not so long ago, symbolic computation meant chiefly symbolic mathematical computation, and this may still be one of the best paradigms.

We think a good working definition of symbolic computation is: computation involving data and control structures which are irregular and unpredictable. In other words, one expects to deal with objects of diverse types. This means, among other things, that if one seeks parallelism, the computations which are to proceed in parallel may be quite different. Pointers to the data structures, rather than the structures themselves, are manipulated. From this arise many of the memory considerations discussed in Section 4.

For illustration, we give a list of fundamental examples of symbolic computation. Some of the items are from [Age83] and [Buc83].

(1) Searching.
(2) Comparing complex structures (pattern matching).
(3) Parsing, parser generators, compilers.
(4) Unification algorithms and solution of term equations.
(5) Evaluation algorithms for logic programming, rewrite rule programming, and functional programming.
(6) Automated theorem proving, in general and in special theories.
(7) Manipulation of abstract data type specifications.
(8) Critical-pair/completion algorithms.
(9) Software prototyping, i.e. rapid generation of possibly inefficient programs from specifications.
(10) Computer-aided program verification, program transformation, symbolic execution, data flow analysis, and program optimization.
(11) Symbolic mathematical computation (computer algebra).
(12) Computer-aided instruction.
(13) Robot control.
(14) Vision/sensory imitation.

3. LANGUAGES.

We are in agreement that there is an urgent need for evolutionary change in general-purpose programming languages for symbolic computation. Specifically, we need: (1) language features that provide the user with the needed level of expression for problem solving (eg. data abstraction, functional abstraction, control abstraction). (2) language

features that support efficient implementation on parallel (or other) machines. eg. protection of name space or elimination of side effects or cdr-coding or tricky storage allocation schemes. There was little agreement on the form this change should take.

3.1. LISP.

The following are statements by individual participants, which do not necessarily have group consensus.

(1) LISP was a good thing, but the time is coming to leave it behind. A massive inertia is carrying it forward. It's not suitable for the parallel processing environment. It is inherently sequential. This workshop should be careful not to allow LISP to be the central topic.

(2) Anti-LISP talk is unjustified. There are man-centuries of development already behind LISP; can't throw that away. LISP has many uses, and there are many existing applications. LISP does support parallelism, and doesn't ignore data structures. LISP is viable on supercomputers, but new implementations are needed. The use of vector processing capabilities in LISP implementations is a rich field awaiting exploration.

(3) The implementation of full LISP with closures is desirable; the question is whether one can get the semantics of deep binding without paying for it, i.e. make up for the loss of the expressibility permitted by FUNARG (see e.g. [Pad83]). Lexical scope rules are also important.

(4) The convergence of LISP and PROLOG is a very promising future direction. Closures and continuations provide a way of implementing the convergence.

3.2. Symbolic computation languages.

We will use the generic term "symbolic computation (SC) languages", except where some particular language is in question. There seemed to be agreement that symbolic

computation languages can be classified as follows: (examples given in parentheses):

Functional Languages (LISP, Scheme - both with and without side-effects).
Declarative languages (PROLOG and variants)
Object-oriented languages (Flavors, Smalltalk, Scheme)
Table manipulation languages (Database Query Languages).
Domain specific languages (IDEAL, APL)
Rule-based languages (OPS-5, EMYCIN)
String-processing languages (Snobol)
Set manipulation languages (SETL)

DARPA doesn't break up the world this way; rather it distinguishes languages as being data-driven, control-driven, or demand-driven ([Age83], p. 47).

The following are statements of individual participants (which we reiterate do not necessarily have group consensus).

(1) It may be desirable to have PRAGMA's in languages so that the programmer can advise the compiler of explicit or implicit parallelism, or other pertinent information.

(2) Programmers may want the facilities from several of the above categories in the same problem. In other words, programmers may want to combine several of the known abstraction facilities. It would be desirable for future symbolic computation languages to permit this, bearing in mind the risks of complicating implementation and ending up with an overly complex, hard-to-use, language. Some efforts have already been made along these lines, e.g. POPLOG, LOGLISP, LISP in PROLOG.

(3) There are fundamental abstractions that "must" be made available, specifically "first class" functions and continuations [Frir)].

(4) Language constructs for exploiting parallelism have not been extensively utilized to date by the symbolic computation community. Initial experiments by Gabriel and Fitch, among others, look promising and show some interesting performance characteristics. Program transformation techniques are used by FORTRAN program restructurers to detect parallelism in scientific applications. The languages used in symbolic computation

show great promise for transformation-based optimizations. Other techniques such as invariant analysis and specification will be important in the detection of parallelism in symbolic applications.

(5) Automatic introduction of parallelism by the compiler is currently extremely difficult (but see [Mar80]).

(6) While protecting the large investment in existing LISP code, care in language design (e.g. first-class functions and continuations) can lead to improved algorithms. This is especially important in the convergence of LISP and PROLOG, which may be achieved using continuations.

(7) Languages must support multiple levels of programming. We should be able to normally work at very high levels, but then become very machine specific when performance is crucial.

(8) A language for symbolic computation should have a nice algebra (LISP doesn't, PROLOG does). This is not only for aesthetic reasons, but to facilitate automatic program transformation for more efficient implementation.

(9) What optimizations may be used to provide first-class functional and control objects efficiently when higher order functions are rarely used? (see e.g. [Pad83]).

(10) Despite the fact that the design of today's supercomputers has been guided by the goal of large-scale numerical computation, it is important to implement LISP and other symbolic computation languages on them. Good performance on today's architectures can be achieved, and a lot is yet to be learned for the benefit of future architectures.

(11) What is the future of "smart" compilation of symbolic computation languages vs. special hardware for more "interpretive" execution?

(12) A great deal of attention is being given to the differences in power, convenience, etc. of "LISP-like" programming languages; much more than is appropriate. The symbolic

computation community has not yet acknowledged the overriding need for software portability. We should be concerned about the tradeoff between the limitations needed for language portability, and the thwarted creativity that may thereby result. It may be that any of a number of "modern" symbolic languages are perfectly adequate for the near and intermediate future (4-10 years). The search for language perfection should not be allowed to become a barrier to the main goal: to provide substantial symbolic facilities to the broad spectrum of computer users, especially those in large-scale scientific computation.

(13) Can we get super-optimizing compilers combined with support for program rewriting to adapt a program to a processor? This might allow a processor to exploit "conventional techniques" (e.g. vectors) to get parallelism and performance.

(14) We might like to have a compiler advise us on parallelism for applicative languages like PROLOG. For more imperative languages like LISP, we might allow the programmer to specify parallelism.

3.3. Symbolic-numerical interface.

We agree that symbolic computation languages should be able to support serious numerical computation. There should not be a barrier between the two types of computation in a given environment. Some existing LISP compilers (e.g. MACLISP) generate good code for numerical computation, demonstrating that it is possible.

The following examples illustrate that both sorts of computing may be needed in the same problem.

- Visual analysis of remote sensing data or electron microscopy slides by Bartels at the University of Arizona and Goto at Tokyo University.
- Motion planning for robots by Moravec at Carnegie-Mellon University and Lozano-Peres at MIT.
- Solution of equations in artificial satellite theory [Dep].
- Solution of transfer functions in optics [Got77].

- Solution of partial differential equations, in particular, preprocessing for finite element computations.

The following are statements of individual participants.

(1) Should there be an effort to interface LISP and FORTRAN? Vendors want to know what priority this should be given. They need to know in order to decide what products are warranted. Note that much of the IMSL library of numerical codes has been translated (via program transformation) from FORTRAN to LISP for inclusion in the MACSYMA system. It is to be noted that support of scientific computation will, given recent developments, require support for parallel and vector computations.

(2) Most people grossly underestimate the amount of symbolic computing inherent in scientific computation. Even if this symbolic component were a small fraction of the total, the cost of classical "FORTRAN-like" computation will be or is so low with supercomputers that the cost of the symbolic component can become dominant in the problem solving process. People can spend several days (full-time) to obtain by hand a symbolic result that requires a few milliseconds if done at supercomputer speeds. Even using MACSYMA on a VAX 11/780; it can require more effort and cost to obtain a "simple" symbolic intermediate result than is required to solve a scientific problem (using it) that would take several hours of "number crunching".

4. MEMORY.

The primary limits to the performance of a symbolic computation environment are the amount, speed, ease of access to, and random access to, memory. We would like to have as much memory as possible, with as much of it physical and as little of it virtual as possible.

The following are some memory features to be considered as potential near-term performance improvements. The number of us in favor of each one was variable.

(1) Hardware garbage collection (most likely ways are with invisible pointers and reference counts. Usually a hardware garbage collection can be done in core.)

(2) Use of reference counts

(3) Parallel memory access

(4) Tagged architectures (note that use of tag bits is a property of the implementation, rather than of the language. Tagged architectures do not necessarily make things machine dependent. The Symbolics machines have given special attention to this issue).

(5) Dispatching

(6) Associative memory/processing

(7) Generic functions (can't implement without runtime type check).

(8) Invisible pointers

(9) Cdr coding

(10) Take exceptions in hardware.

The following are statements of individual participants.

(1) Serious concern was expressed concerning architectural limitations of some current machines, including some widely used super-minis, that have made implementation of an SC language unduly difficult. Specifically, in some cases, all bits of a 32-bit word are used, the paging is wrong, and built-in function calling is bad.

(2) The usefulness of caches is unclear. Stack buffers work as well or better.

(3) What is the degree of runtime typing required for symbolic computation? Is there intrinsically a sufficiently large amount of run time type dispatching to justify hardware support for tagging or is the use of tag dispatching sufficiently infrequent to allow it to be done in-line with no substantial performance penalty?

(4) Virtual memory will always be needed, even in a single user environment (workspaces can fill all of physical memory), but real memory is needed where high performance is the issue.

(5) Must memory organization be completely redesigned to support heap allocation?

(6) The crucial architectural consideration which conventional supercomputer design ignores is memory. Symbolic computation is very memory intensive, and is (to a large extent) dependent on garbage collection. Attempts by the PROLOG community to circumvent garbage collection have been less than successful. There are a number of things which can be done in hardware to facilitate garbage collection, e.g. write-back caching, short use count in the cache, and stack buffers. Concurrent on-the-fly garbage collection can be done with software. There are interesting ideas relating to 'transactional memory' which need more work.

(7) A hard issue is: on which side of the cache do we do garbage collection? This has strong implications for multiple processors unless shared memory is feasible.

(8) Memory organization in the context of parallelism is unclear. It is not clear whether a shared memory or local memories is better; perhaps a blend of the two is best. It is clear that we want a "uniform", shared address space to exist, where we can get at everything, and each processor refers to a given item in the same way.

(9) Current SC language implementations depend heavily on a "uniform" shared address space where we can get at everything and each processor can refer to a given item in the same way. It is clear that this makes implementation simpler and cleaner. However, some existing parallel algorithms "spend" up to half the processor "power" reorganizing memory, moving data around, making copies of data structures, etc. Parallel implementations of SC languages might assign several processors the task of maintaining a "virtual, uniform address space" in a distributed memory architecture. Associative memory is

another possible solution, used, for example, by the Goodyear MPP.

5. SYSTEM ARCHITECTURE.

The following are statements of individual participants.

(1) The five most important changes to get into the next commercially produced supercomputer to affect symbolic computation languages are: tags, garbage collection support, fast function calls, stacks, user-generic datatypes.

(2) One thing that can be done is to identify core capabilities and build plug-in hardware modules for them ("augmentations" of systems). Some candidates:

- unification
- pattern matching
- bignum's (arbitrary precision integers)
- bigfloats (arbitrary precision floating point numbers)
- integer greatest common divisor
- integer FFT with dynamic primitive root of unity in finite field
- garbage collection

It is interesting to contrast this list with what the DARPA report ([Age83], p. 45-46) calls the "special components" of a symbolic processor:

- semantic memory
- signal to symbol transduction
- production rules
- fusion (permitting multiple sources of information to share knowledge)
- inferencing
- search

(3) Suitability/possibility of very powerful LISP machines (100X the performance of a current LISP machine for 20X the price). It is noteworthy that the DARPA proposal ([Age83], p. 45) states that an ultimate performance improvement of about 50X current levels can be achieved for a uniprocessor LISP machine.

(4) Is massive parallelism (more than 1000 processors) useful in symbolic computation? What is the right number of processors?

(5) Are systolic arrays useful? (example: integer greatest common divisor chip [Dav]).

(6) How does the need to support multiple users impact a system design?

(7) Personal supercomputers vs. shared backend "crunchers"? Do we run our screen editor on a supercomputer-class machine, or on a smaller support system? Is the right environment a LISP machine networked to a massive symbolic processor in the background?

(8) Can symbolic computation languages take advantage of pipelines and vector operations? (Current supercomputer LISP implementations do not use the vector pipeline.)

(9) It may be time to give up the Von Neumann architecture in favor of a new execution model, if we expect to achieve truly large (radical?) improvements in performance on symbolic problems. For example, it may be of advantage to employ a new control mechanism more adapted to symbolic calculations, e.g. replace the program counter with a unification functional unit. There are a number of interesting new ideas in this area, such as the OPS-5 machine, DADO, the FAIM machine of A. Davis at SRI, and the NON-VON machine.

(10) Issues for faster symbolic processing:

(A) Architecture of words. Tags are essential for nearly all symbolic languages. This adds to memory costs (an 8-bit tag in addition to a 32-bit word is a good ratio).

(B) Architecture of the memory system in a parallel machine. (i) How is memory shared? Is it true sharing? Is it message-based on some network? (ii) How is cache coherence done (if at all)? (iii) Lazy evaluation will play a key role in parallel symbolic languages. It must be supported by full/empty bits or by the general tagging system, and also by synchronization primitives.

(C) Some operations need to be faster, function-calling in particular. Don't make the super-mini mistake of supporting the wrong function-call primitive.

(D) Hardware or generic arithmetic? The latter would take the form of branch prediction-style hardware for numeric and maybe vector/array instructions.

(E) The Symbolics 3600's ephemeral object space may be a big win to help locality and garbage collection. There should be hardware support for it.

(F) Process creation (i.e. given the code and local store that constitute a closure, create a task and schedule it) has to be fast.

(G) Pipeline turbulence has to be handled. Either shorter pipes, or a Denelcor HEP-like virtual multi-processor, could be useful for parallel LISP. The latter solves turbulence and makes for a good machine for certain LISP constructs, e.g. QLAMBDA.

(H) Parallel languages have to encourage people to program in them effectively. They must be high-level enough so that obvious algorithms/programs are easy to write.

(I) It is not clear what role is to be played by SIMD machines. They are clearly advantageous in certain situations, e.g. as filtering devices on large searches. In such situations, one can narrow down the choices, but not make the final ones.

(J) LISP could be "adjusted" into a "mediumly-typed" language so that more inter-procedural things could be done.

(11) We need to characterize the type of parallelism in symbolic programs/algorithms/applications in terms of granularity, volume and topology of communication required between concurrent entities. For example, there are indications that unification might give on the order of ten parallel search paths of medium granularity. Also, Gabriel reported a study (at the workshop) that showed 100-150 fold parallelism on a MYCIN-like system.

(12) In some pattern matching activity use can be made of SIMD style machines, but in general to achieve the necessary high performance MIMD is required. This carries with it problems. Either we must rely on the user to write explicit language level constrictions to unlock the power, or we must rely on some 'compilation' to determine available

parallelism. Alternatives include data flow, use of pure functional programming, or intelligent analysis of existing (LISP) code to isolate independent (no side effect overlapping) functions or sections of code.

(It should be noted that at least one participant strongly disagreed with the assertion that SIMD is less useful than MIMD for symbolic computation, and in fact the suggestion was made that SIMD machines could be competitive with other architectures for such tasks as processing of rule-based languages).

(13) While pure functional has many mathematical attractions, the commonest implementation by combinators, and using normal order reduction is strongly non-parallel. Some user action is required for multiple processor exploitation (see the work of Warren Burton at University of Colorado). The intelligent compiler approach deserves serious consideration.

(14) Symbolic computations can contain a high degree of parallelism at the high-level (conceptual, functional, algorithmic), but much less at the instruction or operand level. in what way can they be performed on, or with the assistance of existing and future supercomputers? To use a vector computer or pipelined computer (either on instructions or data) seems to be of use only on specialized subproblems mostly connected with numerical processing or table searching, etc.

(15) What sorts of parallel architectures are useful? Single instruction machines are not appropriate for the general computation. MIMD machines, probably with shared memory, seem to be the most appropriate architectures. They can be programmed at a high level, yet take advantage of parallelism. It would seem best to have a system that allowed the generation of several processes each implementing a self-contained subtask, appropriately scheduled. Existing vector machines might best be used as specialized co-processors for floating point manipulation, or particular algorithms, as might SIMD machines.

(16) Can we do combined symbolic/numeric work best in LISP on a current commercial supercomputer, or with a LISP machine and such a supercomputer, or with a 68000 and a supercomputer, or with a 68000 and a fast floatng point board?

(17) There are substantial classes of important problems which require enormous symbolic computational power. The main characteristics of these problems are that they deal with data structures and control structures which are unpredictable, and are thus not amenable to implementation on vector or array machines. The main source of computational power for these problems must be parallelism, preferably with an architecture that imposes as few constraints on the programmer as possible. The most flexible of these architectures has been called a 'paracomputer' by Schwartz. This architecture is also well-suited to those numerical problems which lack the structure appropriate for pipelined or vector implementation, so machines of this type are likely to be available in the near future. The central question which arises, therefore, is that of the amount of parallelism inherent in symbolic computations. At the present time there is little data on this, though most participants (at the workshop) felt that many problems could yield to parallel algorithms. However, there has been relatively little work done in this area; most programming and algorithm design has been oriented towards serial architectures.

(18) There are arguments for both sides of the question of "personal supercomputer" vs "backend supercomputer". If the user has a desktop supercomputer with a decent environment, would he/she be more productive? Quite possibly! On the other hand, backend machines will always have their place. Massive rule based expert systems to solve large problems will not go away. One possible arrangement would be to have a powerful desktop machine for software development, and a backend machine for really tough problems. One could also picture hardware assistance in compiler work, since massive amounts of pattern matching are being done. A hardware pattern matcher would be great.

(19) A typical computation application is searching and pattern matching. A massively parallel computer such as the Goodyear MPP would be very useful in speeding up these basic operations. This implies that value-based systems will be able to select a rule in parallel. There are some problems in doing this such as how large should the "associative" content-addressable memory be.

(20) In symbolic mathematics systems, one could picture the following basic operations being hardware based to increase performance:

integer arithmetic
indefinite precision floating point arithmetic
rational number arithmetic
integer greatest common divisor
integer FFT with dynamic root of unity in a finite field

(21) Tags are artifacts of current implementations and not intrinsic to SC languages - or even to efficient implementations of them. The functionality of such things as full/empty bits and dynamic typing are being provided in other systems by other mechanisms.

(22) One participant's opinion: a 68000 plus infinite speed floating point arithmetic will not produce much power.

6. PERFORMANCE EVALUATION.

The state of this important area is not advanced. We specifically request that funding agencies support benchmarking studies, for example, memory usage statistics, or the behavior of a CONS cell during its lifetime.

The following is the best list we could construct of current benchmark programs for symbolic computation:

(1) R. Gabriel LISP benchmarks

(2) M. Griss LISP benchmarks

(3) The Reduce test file [Hea84].

(4) Macsyma test files

(5) Theorem proving examples

(6) PROLOG benchmarks in David Warren's thesis (available from Frank Kuo, SRI)

(7) OPS-5 rule sets

(8) Compilation timings (e.g. time LISP compiler compiling the Reduce system, the blocks world)

(9) The SIGSAM problems (see the ACM SIGSAM Bulletins).

The following are statements of individual participants.

(1) Can we identify problems that need just a factor of two to five improvement to become feasible? Knowledge of such problems can help drive near-term development. Some examples: getting expert systems to work in real time (e.g. DARPA's battlefield manager must run at five times real time), increasing the subsets of English which natural language systems can handle, and VLSI design.

(2) Good benchmarks provide a measure of performance that allows a comparison between competing architectures. Benchmarks should be general in nature i.e. DAIS min or whetstones. They should never be the only measure of performance.

(3) Case studies of four to five "standard" symbolic problems, using state of the art hardware, are needed.

(4) A benchmark for symbolic computing:

step 1: Write down 3 functions of variables x,y,z,t

a(x,y,z,t) = ordinary but lengthy mathematical expression

b(x,y,z,t) = " "

c(x,y,z,t) = " "

step 2: define f(x,y,z,t) = (a+b-c)/(1+a*b)

step 3: Display those values in x,y,z and t where f is infinite.
These may be curves, points or surfaces.

step 4: Generate Taylor's series expansions with 2 terms for
1/f along all the loci where f is infinite.

step 5: Compute a complex formula depending on f at a fine grid
of points in x-y-z-t space.

7. GLOSSARY.

CDR-CODING

A technique for eliminating unnecessary pointers in LISP lists.

GARBAGE COLLECTION

The automatic reclamation of memory which is not currently in use, without explicit dereferencing by the programmer being required.

LISP

A widely-used programming language for symbolic computation.

MIMD ARCHITECTURE

"Multiple Instruction Multiple Data" parallel architecture.

PROLOG

A widely-used programming language for symbolic computation.

SIMD ARCHITECTURE

"Single Instruction Multiple Data" parallel architecture.

SYSTOLIC ARRAY

A parallel architecture consisting of an array of processors with a uniform protocol for passing data.

8. APPENDIX: MAILING ADDRESSES OF PARTICIPANTS.

John Aldag
Applications Department
Cray Research
1440 Northland Drive
Mendota Heights, Minnesota 55120

Wayne Anderson
Group C-10
Mail Stop B296
Los Alamos National Laboratory
Los Alamos, New Mexico 87545

Dennis Arnon
Computer Science Department
Purdue University
West Lafayette, Indiana 47907

Bobby F. Caviness
University of Delaware
Department of Computer and Information Sciences
103 Smith Hall
Newark, Delaware 19716

Jagdish Chandra
Army Research Office
P.O. Box 12211
Research Triangle Park, North Carolina 27709

Elizabeth Cuthill
David Taylor Naval Ship Research and Development Center
Code 1805
Bethesda, Maryland 20084

Alvin Despain
University of California
Computer Science Division
Department of EECS
503 Evans Hall
Berkeley, California 94720

Robert Douglass
Group C-10
Mail Stop B296
Los Alamos National Laboratory
Los Alamos, New Mexico 87545

John Fitch
School of Mathematics

University of Bath
ENGLAND

Daniel Friedman
Computer Science Department
Indiana University
Lindley Hall 101
Bloomington, Indiana 47405

Richard Gabriel
Computer Science Department
Stanford University
Stanford, California 94305

Jeffrey Greif
Inference Corporation
5300 W. Century Blvd.
Fifth Floor
Los Angeles, California 90045

Martin Griss
Computation Research Center
3U3
Hewlett-Packard
1501 Page Mill Road
Palo Alto, California 94304

Malcolm Harrison
New York University
251 Mercer Street
New York, New York 10012

Christopher Haynes
Computer Science Department
Indiana University
Lindley Hall 101
Bloomington, Indiana 47405

Robert Kessler
University of Utah
Department of Computer Science
3160 Merrill Engineering Building
Salt Lake City, Utah 84112

Wayne Matson
Symbolics, Inc.
243 Vassar Street
Cambridge, Massachusetts 02139

Donald Oxley
Texas Instruments

P.O. Box 226015
Mail Station 238
Dallas, Texas 75266

John Rice
Computer Science Department
Purdue University
West Lafayette, Indiana 47907

John Smit
Goodyear Aerospace Corporation
D920/G3
1210 Masillon Road
Akron, Ohio 44315

William Stockwell
Control Data Corporation
HQW10N
Post Office Box 0
Minneapolis, Minnesota 55440

Paul S. Wang
Kent State University
Department of Mathematical Sciences
Kent, Ohio 44242

References

Age83.
DARPA (Defense Advanced Research Projects Agency), *Strategic Computing - New generation computing technology: a strategic plan for its development and application to critical problems in defense*October 28, 1983.

Buc83.
B. Buchberger, "A 'Prolog' to EUROCAL 85," *SIGSAM Bulletin (of the Association for Computing Machinery)* **17**, pp. 8-11 (1983).

Dav.
J. Davenport and Y. Robert, "PGCD et VLSI," in *Comportement des automates et applications (Proceedings)*, Springer-Verlag(to appear)

Dep.
R. Deprit, *Celestial Mechanics* **1** p. 2

Frir).
D.P. Friedman, C.T. Haynes, and E. Kohlbecker, "Programming with Continuations," *Proceedings of Workshop on Program Transformation and Programming Environments*, Springer-Verlag, ((to appear)).

Got77.
E. Goto and Soma, *Optik* **48** p. 77 (1977).

Hea84.
A. Hearn, *Reduce 3.1 Users Manual*, Rand Corporation(1984).

Kah82.
G. Kahn, "The scope of symbolic computation,," pp. 237-242 in *Fifth generation computer systems*, ed. T. Moto-oka,North-Holland, Amsterdam(1982).

Mar80.
J. Marti, "Compilation techniques for a control-flow LISP system," pp. 203-207 in *Proceedings of the LISP '80 Conference*, (1980).

Mor35.
H. Moravec,, *Obstacle Avoidance and Navigation in the Real World by a Seeing Robot Rover*, Stanford University, AIM-340, CS-813(1980. Cost: $4.35). (Thesis: PhD in Computer Science)

Mot82.
T. Moto-oka (ed.), *Fifth generation computer systems*, North-Holland, Amsterdam(1982).

Pad83.
J.A. Padget and J.P. Fitch, *On a New Model for Dynamic Access Environments*, School of Mathematics, University of Bath(1983). ms., 12pp.