ples and Practice of Parallel Programming, March 1990.

[10] W. P. Crowley, C. P. Henderson, and T. E. Rudy. The simple code. Technical Report UCID 17715, Lawrence Livermore National Laboratory, Livermore, CA, Februrary 1978.

[11] J. Feo, D. Cann, and R. Oldehoeft. A report on the Sisal language project. *Journal of Parallel and Distributed Computing*, 10:349–365, December 1990.

[12] K. Gopinath. *Copy Elimination in Single Assignment Languages.* PhD thesis, Computer Systems Laboratory, Stanford University, Stanford, CA, 1989.

[13] C. P. Hendrickson. Programming a real code in a functional language (part i). In *Visions of Supercomputing Conference*, September 1991.

[14] R. W. Hockney and C. R. Jesshope. *Parallel Computers.* Adam Hilger Ltd, Techno House, Redcliffe Way, Bristol BS1 6NX, 1984.

[15] P. Hudak. Conception, evolution, and application of functional programming languages. *ACM Computing Surveys*, 21(3):359–411, September 1989.

[16] P. Hudak and A. Bloss. The aggregate update problem in functional programming systems. In *Twelfth Annual ACM Conference of the Principles of Programming Languages*, pages 300–313, January 1985.

[17] J. Hughes. Why functional programming matters. *The Computer Journal*, 32(2):98–107, April 1989.

[18] B. J. MacLennan. *Functional Programming Practice and Theory.* Addison-Wesley Publishing Company, Reading, Massachusetts, 1990.

[19] J. R. McGraw, D. J. Kuck, and M. Wolfe. A debate: Retire Fortran? *Physics Today*, 37(5):66–75, May 1984.

[20] J. R. McGraw, S. K. Skedzielewski, S. J. Allan, R. R. Oldehoeft, J. Glauert, C. Kirkham, W. Noyce, and R. Thomas. SISAL: Streams and iteration in a single assignment language: Reference manual version 1.2. Manual M-146, Rev. 1, Lawrence Livermore National Laboratory, Livermore, CA, March 1985.

[21] F. H. McMahon. The Livermore Fortran kernels: A computer test of the numerical performance range. Technical Report UCRL-53745, Lawrence Livermore National Laboratory, Livermore, CA, December 1986.

[22] R. R. Oldehoeft, D. C. Cann, J. T. Feo, and A. P. W. Bohm. The SISAL 2.0 reference manual. Technical Report UCRL-MA-109098, Lawrence Livermore National Laboratory, Livermore, CA, December 1991.

[23] J. E. Ranelletti. *Graph Transformation Algorithms for Array Memory Optimization in Applicative Languages.* PhD thesis, University of California at Davis, Computer Science Department, Davis, California, 1987.

[24] S. K. Skedzielewski and R. J. Simpson. A simple method to remove reference counting in applicative programs. In *Proceedings of CONPAR 88*, September 1988.

Disclaimer

This document was prepared as an account of work sponsored by an agency of the United States Government. Neither the United States Government nor the University of California nor any of their employees, makes any warranty, express or implied, or assumes any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately owned rights. Reference herein to any specific commercial products, process, or service by trade name, trademark, manufacturer, or otherwise, does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States Government or the University of California. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States Government or the University of California, and shall not be used for advertising or product endorsement purposes.

Table 13: Miscellaneous data for the program suite: Concurrent-Vector.

CRAY Y-MP/864	Concurrent-Vector			UNICOS 6.1
	Memory Image Size Kilowords		Compilation Time Seconds	
Program	Sisal	Fortran	Sisal-H, H-a.out	Fortran-a.out
KIN16	**250**	296	**4.0**, 8.8	7.5
RICARD	**110**	156	**4.7**, 7.1	10.7
CFFT	6571	**5436**	12.1, 24.0	library-link
BMK11A	260	**238**	**16.2**, 31.0	23.8
SIMPLE	734	**392**	30.4, 42.9	**30.0**
UNSPLIT	1315	**932**	**35.6**, 73.0	55.8
WEATHER	381	**208**	103.0, 86.2	**55.2**

allelization. Fourth, programs can run as fast as, if not faster than conventional languages. The time for change is now.

Acknowledgements

Thanks are due to the staff and clients at the National Energy Research Supercomputer Center at LLNL for for their patience while I tuned OSC and collected the data. I also wish to thank Rod Oldehoeft and Tom DeBoni for improving this paper, and the staff at Supercomputing 91 for selecting it for publication in CACM. Also, special thanks are due to Jim McGraw, David Kuck, and Michael Wolfe, as their debate in 1984 made this report possible.

Finally, I would like to thank my wife, Sue, for both her technical and moral support.

This work was supported by the Applied Mathematical Sciences subprogram of the Office of Energy Research, U.S. Department of Energy, by Lawrence Livermore National Laboratory under contract No. W-7405-Eng-48.

References

[1] Arvind and R. Nikhil. Executing a program on the MIT tagged-token dataflow architecture. *IEEE Transactions on Computers*, 39(3):300–318, March 1990.

[2] Jeffrey M. Barth. A practical interprocedural data-flow analysis algorithm. *Communications of the ACM*, 21(9):724–736, September 1978.

[3] J. B. Bell, P. Colella, J. A. Trangenstein, and M. Welcome. Adaptive mesh refinement on moving quadrilateral grids. In *AIAA 9th Computational Fluid Dynamics Conference*, June 1989.

[4] A. Bloss. Update analysis and the efficient implementation of functional aggregates. In *The 4th International Conference on Functional Programming Languages and Computer Architecture*, pages 26–38, 1989.

[5] J. Boyle and T. Harmer. A practical functional program for the CRAY X-MP. Technical Report MCS-P159-0690, Argonne National Laboratory, 1990.

[6] D. C. Cann. *Compilation Techniques for High Performance Applicative Computation.* PhD thesis, Colorado State University, Computer Science Department, Fort Collins, CO, 1989.

[7] J. R. Cann. Analysis of the gel electrophoresis of looped protein-DNA complexes by computer simulation. *Molecular Biology*, (216), September 1990.

[8] J. R. Cann, E. York, J. Stewart, J. Vera, and R. Maccioni. Small zone gel chromatography of interacting systems: Theoretical and experimental evaluation of elution profiles for kinetically controlled macromolecule-ligand reactions. *Analytical Biochemistry*, (175), December 1988.

[9] P. Chang and G. Egan. An implementation of a barotropic numerical weather prediction model in the functional language SISAL. In *Proceedings of the SIGPLAN 1990 Symposium on Princi-*

Table 12: Performance data for CFFT in seconds: Concurrent-Vector.

CRAY Y-MP/864	Concurrent-Vector					Seconds
	One CPU			Eight CPUs		
Language	Gen	Init	Fft	Gen	Init	Fft
Sisal	**0.008**	**0.066**	0.254	**0.002**	**0.020**	**0.049**
Fortran	0.009	0.220	**0.190**	**0.002**	0.068	0.061

The SISAL version of WEATHER, which is the most complex program in the suite, yields the best parallel performance improvement over FORTRAN (a speedup of 5.7). The reason for this difference was the lack of interprocedural analysis and a means to manage separate compilation. For SIMPLE, the third most complex code, the FORTRAN compiler was unable to unravel the true behavior of the equation of state package, regardless of function expansion. Thus the SISAL version of SIMPLE ran 4 times as fast as FORTRAN.

Overall, the performance data speaks for itself. For the vector and concurrent-vector executions of KIN16, RICARD, BMK11A, and LOOPS, SISAL was competitive. See Tables 9, 10, and 11 for a complete summary of LOOPS performance, showing both the arithmetic and harmonic means in MFLOPS. Note SISAL's winning harmonic mean of 11.4 on four processors. For the concurrent-vector execution of CFFT, SIMPLE, UNSPLIT, and WEATHER, SISAL was superior. For a breakdown of CFFT performance see Table 12. Note the high FFT initialization cost for FORTRAN, and SISAL's superior eight processor performance for the transform (20% better than FORTRAN). The overall computation rate of CFFT for SISAL on eight processors was over one GFLOPS (Table 8), while the FORTRAN library yielded 634 MFLOPS.

Performance aside, there remain two unanswered questions: memory consumption and compilation time. Most skeptics claim that functional languages use memory profligately. In an industrial setting, where every last byte counts, this would prohibit functional computation altogether—regardless of performance. Table 13 shows the mean memory image size of the concurrent-vector runs. Only the SISAL runs for SIMPLE and WEATHER showed a discrepancy, using about twice the as much memory as FORTRAN; however, these two codes aggressively outperformed FORTRAN on both four and eight processors. Note that the mean memory image for RICARD and KIN16 are both smaller than FORTRAN.

Another argument against functional languages has been compilation time. Table 13 shows the compilation time for both languages. In general, the SISAL compiler can generate optimized C and FORTRAN code (Sisal-H) in less time than it takes to compile FORTRAN into executable code, and OSC spends most of its time in in the local compilers. This is reasonable. The use of C and FORTRAN for intermediate code would not be the correct choice for a production compiler. But the data does show that the required optimizations for SISAL are cheap. In fact, their cost is linear in the number of operations in the program (see [11]).

7 The Evidence Mounts

In this report, we have attempted to rekindle an old debate: Is our reliance on FORTRAN necessary, or can we transcend the imperative *tar pit* and reach a new level of abstraction? The biggest concerns to date have been performance, memory utilization, and compilation time, but recent advances in the functional language community have dispelled these. The data presented in this report is further evidence, and it corroborates the work of Boyle at Argonne National Laboratory who has demonstrated parity with FORTRAN using Lisp [5].

We do not claim, however, that SISAL and Lisp are the successors of FORTRAN. What we do claim is that the successor should be functional, and performance is no longer an issue. Tomorrow's parallel machines will not only provide massive parallelism, but will present programmers with a combinatorial explosion of concerns and details. The imperative programming model will continue to hinder the exploitation of parallelism.

In summary, the functional paradigm yields several important benefits. First, programs are more concise and easier to write and maintain. Second, programs are conducive to analysis, being mathematically sound and free of side effects and aliasing. Third, programs exhibit implicit parallelism and favor automatic par-

Table 8: Performance data for the program suite in MFLOPS: Concurrent-Vector.

CRAY Y-MP/864	Concurrent-Vector				MFLOPS	
	One CPU		Four CPUs		Eight CPUs	
Program	Sisal	Fortran	Sisal	Fortran	Sisal	Fortran
KIN16	**232**	218	**884**	841	1501	**1555**
RICARD	208	**213**	602	**664**	955	**1123**
CFFT	**249**	196	**824**	549	**1177**	634
BMK11A	42	**48**	59	**67**	61	**70**
LOOPS	30	**37**	39	**42**	42	**43**
SIMPLE	20	**21**	**61**	21	**85**	21
UNSPLIT	105	**133**	**380**	213	**626**	217
WEATHER	**14**	15	**52**	16	**87**	15

Table 9: Performance summary for Livermore Loops: Vector.

CRAY Y-MP/864	Vector		MFLOPS	
Measure	Sisal		Fortran	
Minimum	**1.3**	Loop 13	0.8	Loop 13
Maximum	260.7	Loop 7	**271.5**	Loop 7
Arithmetic Mean	68.0		**84.4**	
Harmonic Mean	9.1		**11.0**	

Table 10: Performance summary for Livermore Loops on one CPU in MFLOPS: Concurrent-Vector.

CRAY Y-MP/864	Concurrent-Vector/1		MFLOPS	
Measure	Sisal		Fortran	
Minimum	**1.3**	Loop 13	0.8	Loop 13
Maximum	**249.7**	Loop 7	246.6	Loop 7
Arithmetic Mean	64.3		**71.3**	
Harmonic Mean	9.1		**10.7**	

Table 11: Performance summary for Livermore Loops on four CPUs in MFLOPS: Concurrent-Vector.

CRAY Y-MP/864	Concurrent-Vector/4		MFLOPS	
Measure	Sisal		Fortran	
Minimum	**1.3**	Loop 13	0.8	Loop 13
Maximum	**535.6**	Loop 7	529.3	Loop 7
Arithmetic Mean	92.0		**108.7**	
Harmonic Mean	**11.5**		10.9	

Table 5: Performance data for the program suite in seconds: Vector.

CRAY Y-MP/864	Vector	Seconds
	One CPU	
Program	Sisal	Fortran
KIN16	**74.9**	79.5
RICARD	39.4	**38.4**
CFFT	**0.37**	0.42
BMK11A	4.4	**4.1**
LOOPS	16.0	**12.3**
SIMPLE	9.4	**9.0**
UNSPLIT	10.2	**7.3**
WEATHER	**6.1**	6.2
totals	160.8	**157.2**

Table 6: Performance data for the program suite in seconds: Concurrent-Vector.

CRAY Y-MP/864	Concurrent-Vector					Seconds
	One CPU		Four CPUs		Eight CPUs	
Program	Sisal	Fortran	Sisal	Fortran	Sisal	Fortran
KIN16	**74.8**	79.9	**19.7**	20.7	11.6	**11.2**
RICARD	39.8	**38.9**	13.8	**12.5**	8.7	**7.4**
CFFT	**0.33**	0.42	**0.10**	0.15	**0.07**	0.13
BMK11A	4.8	**4.2**	3.4	**3.0**	3.3	**2.9**
LOOPS	16.1	**12.9**	12.3	**11.4**	11.3	**11.0**
SIMPLE	9.4	**9.0**	**3.1**	8.8	**2.2**	8.8
UNSPLIT	10.1	**8.0**	**2.8**	5.0	**1.7**	4.9
WEATHER	**7.3**	7.0	**2.0**	6.7	**1.2**	6.8
totals	162.6	**160.3**	**57.2**	68.3	**40.1**	53.1

Table 7: Performance data for the program suite in MFLOPS: Vector.

CRAY Y-MP/864	Vector	MFLOPS
	One CPU	
Program	Sisal	Fortran
KIN16	**232**	218
RICARD	211	**216**
CFFT	**225**	197
BMK11A	46	**49**
LOOPS	**30**	39
SIMPLE	20	**21**
UNSPLIT	104	**146**
WEATHER	**17**	**17**

Table 3: Compilation options for FORTRAN.

CRAY Y-MP/864 UNICOS 6.1 cf77:cft77 5.0.2.0	
Desired Execution Mode	**Options**
Vector	-Zp -Wd"-ev68 -dci" -Wf"-pd"
Concurrent-Vector	-Zp -Wd"-ecv68i" -Wf"-pd"

Table 4: Compilation options for SISAL.

CRAY Y-MP/864 UNICOS 6.1 OSC 12.0	
Desired Execution Mode	**Options**
Vector	-hybrid -noconcur -vector
Concurrent-Vector	-hybrid -concur -vector

library provided by Cray Research Inc. (microtasking version)[2]. For the comparison we used a problem size of 2^{19} complex values.

The next largest code is BMK11A, a benchmark developed at Los Alamos National Laboratory to evaluate Cray computer systems. BMK11A exercises excerpts of a particle transport code. For the comparison we exercised each excerpt 200 times using vectors of 2560 elements.

The fifth code in the suite is the Livermore Loops (LOOPS), which is a set of 24 scientific kernels taken from production codes at LLNL[21]. For many years, scientists have used the kernels to benchmark high-performance computers. For the comparison we timed 4000 repetitions of each kernel.

The next code is SIMPLE[3], a Lagrangian hydrodynamics benchmark developed at LLNL [10]. This program simulates the behavior of fluid in a sphere, and uses a non-vector, but concurrent equation of state package. Here we ran a 100×100 problem requiring 62 time steps to complete.

UNSPLIT, the second largest code in the suite, is an unsplit integrator taken from an adaptive mesh refinement code at LLNL [3]. For the comparison we propagated a shock down a tube comprising 160×80 zones, timing the first 40 steps of execution.

The largest and last program in the suite is WEATHER [9]. This code, developed at Royal Melbourne Institute of Technology, is a one-level barotropic weather prediction code. The Department of Meteorology at the University of Melbourne provided the original FORTRAN, written and tuned for a VAX uniprocessor; as such, the code is explicitly scalar[4]. Differing from the other programs, which occupy single source files, WEATHER occupies 33 files and manipulates complex data. For this code, we timed 20 cycles of execution using a 420 km grid.

For all the codes, the data reflects real arithmetic (64 bits). This includes the times presented in the previous section (Table 2).

6 Performance and Analysis

Tables 5 and 6 present the performance data for all the runs in seconds. The first is for vector operation exclusively, and the second is for concurrent and vector operation combined. We present the associated MFLOPS rates in Tables 7 and and 8 respectively[5].

The data show two trends. First, the vector and single processor concurrent-vector performance for all the codes is nearly identical, as shown by the totals. Second, the disparity in parallel performance between SISAL and FORTRAN is proportional to the complexity of the program. That is, as the complexity of the application increases so does SISAL's predominance. Observe the eight processor concurrent-vector execution times for SIMPLE, UNSPLIT, and WEATHER.

[2] Note, we assume that Cray wrote the CFFT library routine in FORTRAN; however, it may contain CAL assembly language.

[3] Different versions of SIMPLE exist. We used version 2a for this report.

[4] Personal communication with Martin Dix, CSIRO Division of Atmospheric Research, Australia.

[5] We collected the single processor, concurrent-vector MFLOPS rates for SISAL using the Cray Y-MP hardware performance monitor (except for CFFT, where we monitored the FORTRAN library routine, and LOOPS, where we used the operation counts presented in [21]). We calculated the other SISAL and FORTRAN MFLOPS rates as a function of the monitored rates.

Research Supercomputer Center (NERSC) at LLNL. We quickly draw two conclusions from the data: The cost of an implementation that naively copies is truly exorbitant, but the applied optimizations are dramatically effective.

The reader can find a discussion of all the optimizations and their origins in [2,4,6,12,16,23,24]. Further elaboration is beyond the scope of this report. The point we wish to make here is that intelligent compilation can eliminate copying.

5 Potpourri

In the next section we compare FORTRAN to SISAL 1.2 using the CRAY Y-MP at NERSC. This section enumerates some important considerations, concerns, and caveats, and describes the conditions of the study and the characteristics of the program suite.

5.1 Code Quality

The SISAL compilation system (OSC[1]) generates optimized C and FORTRAN as its intermediate code and relies on local compilers to finish compilation. This scheme ultimately simplifies code generation and improves portability, but it places the responsibility for code quality on the resident compilers. Although improving (motivated by the proliferation of UNIX), the quality of most C compilers remains poor. In the study we used OSC version 12.0 and SCC version 3.0.1 to compile the generated C. In general, we were impressed with the quality of SCC. We used CF77 version 5.0.2.0 to compile both the generated FORTRAN and the FORTRAN codes themselves.

5.2 Code Development

Each SISAL code in our benchmark suite is a transliteration of its FORTRAN equivalent (with the exception of CFFT, which we developed from first principles). At no time were the algorithms changed to improve performance. However, the SISAL codes were written to exploit the true data dependencies of the respective algorithms.

5.3 Levels of Improvement

The data presented in this report does not measure the degree or ease with which tuning could improve the SISAL and FORTRAN codes. We did not manually annotate the FORTRAN to improve parallel performance. However we did enable as many compiler options as possible to yield the most effective execution. Tables 3 and 4 show the options we used for both languages. In general, since most performance tuning involves a restructuring that modifies the flow of data through a computation, we would expect that SISAL would be easier to tune.

5.4 Library Software

Although we only evaluate SISAL's performance in this report, it is important to note that in our work we have addressed the issues of library software and mixed language programming. These concerns are important in the complete evaluation of a programming language and its ability to perform in an industrial environment. We have implemented a bidirectional interface linking SISAL to FORTRANand C, and as a feasibility study, researchers at LLNL are using the interface and migrate several production codes into SISAL [13]. For this report, we did not use the interface to access Cray's library software to boost performance.

5.5 Data Gathering

The resident operating system on the NERSC CRAY Y-MP is UNICOS 6.1. The runs were made at off hours and were repeated to insure timing accuracy. We did not accept a parallel run unless we received at least 95% of the machine for the entire execution.

5.6 The Program Suite

We will now describe the program suite (see Table 1). The smallest code is KIN16, a production code developed at the University of Colorado Medical Center (UCMC) [7]. KIN16 simulates the electrophoresis of looped protein and DNA complexes. We ran a complete production run requiring 42,000 time steps and 2 columns of 14,285 levels.

The next largest code in the suite is RICARD, also a production code developed at UCMC [8]. RICARD simulates experimentally observed elution patterns of proteins and ligands in a column of gel. For the comparison we ran a production run requiring 40,000 time steps to complete a simulation involving 5 proteins and a column of 1,215 levels.

The third code is CFFT, which realizes a one-dimensional complex fast Fourier transform [14]. The FORTRAN version was taken from the signal processing

[1] OSC stands for Optimizing SISAL Compiler.

Table 1: The program suite.

Program	Source Lines	Time Steps	Problem Size	Subject Area
KIN16	**225**	**42000**	**2x14285**	**Gel Electrophoresis**
RICARD	**297**	**40000**	**5x1315**	**Gel Chromatography**
CFFT	**517**	**1**	**524288**	**Fourier Transform**
BMK11A	**1003**	**200**	**2560**	**Particle Transport**
LOOPS	**1116**	**4000**	**100-1000**	**Scientific Kernels**
SIMPLE	**1527**	**62**	**100x100**	**Hydrodynamics**
UNSPLIT	**1937**	**40**	**160x80**	**Hydrodynamics**
WEATHER	**2712**	**20**	**420km**	**Weather Modeling**

Table 2: The effectiveness of copy elimination analysis.

Cray Y-MP/864	**Vector**		**Seconds**	
Program	**NO Optimization**	**WITH Optimization**	**Speedup**	**% Copies Removed**
KIN16	11405.1	74.9	152	100%
RICARD	1877.9	39.4	48	100%
CFFT	41.9	0.37	113	100%
BMK11A	1002.6	4.4	228	100%
LOOPS	681.6	16.0	43	100%
SIMPLE	177.7	9.4	19	99%
UNSPLIT	399.1	10.2	40	99%
WEATHER	303.5	6.1	50	99%

3 Sisal Overview

In this section we highlight the salient features of SISAL 1.2 [20] and introduce its soon to be released upgrade: SISAL 2.0 [22]. In the interest of promoting the upgrade, we use 2.0 syntax for all the examples in this report. To begin the discussion, we present a SISAL version of Gaussian Elimination without pivoting. It solves a set of linear equations of the form `Ax=B`, where `A` is an `N`×`M` matrix and `x` and `B` are `N`×`1` column vectors. Note the Pascal-like syntax and data types.

```
type OneI   = array [..] of integer;
type OneD   = array [..] of double;
type TwoD   = array [..] of OneD;

function Reduce( piv:integer; A:TwoD;
                 B:OneD
                 returns TwoD, OneD )
  let
    % FORM THE MULTIPLIERS FOR THIS
    % REDUCTION STEP
    mults := A[..,piv]/A[piv,piv];
  in
    % REDUCE ALL THE ROWS OF A AT
    % INDEX i (row IS A VECTOR!)
    for row in A at [i] do
      nrow,
      nB := if i = piv then
              % REDUCE THE PIVOT ROW
              row /A[piv,piv],
              B[i]/A[piv,piv]
            else
              % REDUCE A NONPIVOT ROW
              row-mults[i]*A[piv],
              B[i]-mults[i]* B[piv]
            end if
    returns array of nrow,
            array of nb
    end for
  end let
end function

function Gauss( N:integer; A:TwoD; B:OneD
                returns OneD )
  % APPLY N SUCCESSIVE REDUCTIONS TO A
  % AND RETURN THE SOLUTION VECTOR
  for i in [1..N] do
    new A, new B := Reduce( i, A, B );
  returns B
  end for
end function
```

This example shows the two loop forms found in SISAL. The loop in function `Reduce` defines parallel execution while the loop in function `Gauss` defines sequential execution. The absence or presence of reserved word `new` distinguishes between the two. As expected, only the sequential form allows the specification of data dependencies across iterations (`new A` in the example).

Other features of SISAL include reduction operations, records, unions, array modifiers, and streams for non-strict computation. SISAL 2.0 includes several other features, including higher-order functions, true matrices, modules, element wise operations on arrays and streams, and typesets. Of these, higher-order functions are the most important. They add an additional level of abstraction to the model, allowing functions to be treated as data. The next most important are typesets, which allow data types to be derived from argument types and operations that transform types. Both features are important in the development of library software.

4 Problems and Solutions

Earlier in this report we noted that functional languages, because of their semantics, meet all the requirements of a parallel programming language. But by raising the level of abstraction have we hindered performance? We agree with Kuck and Wolfe that performance is the bottom line. Until recently, poor performance has been the strongest argument against functional computing.

Functional languages are considered inefficient because of their side effect free semantics. For example, without side effects an array update must define an entirely new array. However in-place operations can be automatically compiled into most functional programs without violating their computational semantics (called *copy elimination*). The goal is to identify the last consumer of an object and configure it for in-place operation. In this section we illustrate both the costs of copying and the power of copy elimination. To do this we consider the SISAL programs used in this report and present their performance with and without optimization. Table 1 presents the program suite itself. We defer further description of the individual programs until the next section. For now, it will suffice to say that they are all array and floating point intensive, and exhibit parallelism. Table 2 summarizes their performance with and without the copy optimizations. The times are in seconds and were gathered on the Cray Y-MP/864 at the National Energy

2 Programming Alternatives

In 1984, McGraw noted that by all indications future supercomputers would be multiprocessors. Today, most supercomputer users and vendors would agree. But can programmers take advantage of the horsepower and not sacrifice portability, longevity and correctness? In this section we highlight the aesthetics of functional programming and contrast the functional model to the imperative model of FORTRAN. To begin, we list the desired characteristics of a true high-level parallel programming language [1]:

1. The language must insulate the programmer from the underlying architecture. Deriving and expressing a parallel algorithm is difficult enough; one should not have to reprogram it for each new machine.

2. Parallelism must be implicit in the semantics of the language. The compilation system should not have to unravel the behavior of the computation.

3. When a programmer desires determinancy, the language should guarantee it. Regardless of the conditions of execution, a program that realizes a determinate algorithm should yield the same results for the same data.

Of the three items, the last is an issue only when automatic parallelizing compilers are not available and the programmer is responsible for expressing and managing parallelism. Programmers will make mistakes, and these mistakes may remain hidden until system activity changes the rate of execution. This is all we will say about determinancy, as most parallel machines support automatic parallelizing compilers.

Regarding the first two items, however, imperative languages fail to meet the requirements. Remember that languages like FORTRAN were designed to exploit von Neumann machines. As such their computational model assumes that a single program counter will step through a program in the order of the statments within it. This is not necessarily the best path. Because the programmer is responsible for defining the order, he must explicitly map the data to the physical resources of the machine. This mapping is machine dependent, and requires redefinition when a program is moved to a new machine. Further, the choice of mapping affects the compiler's ability to exploit the available parallelism. If the compiler cannot unravel the implied behavior, it cannot remap the data and select the best order of execution. Hence, the compiler is at the mercy of the programmer, and must react conservatively when faced with potential conflicts. The culprit, of course, is the assignment statement.

For example, consider the following FORTRAN excerpt:

```
A = Conduc(X)
B = Newque(Y)
```

Determining if these statements can execute in parallel requires a full understanding of both functions. Because of COMMON blocks, they might share data. Further, because of aliasing, some combination of `X`, `Y`, `A`, or `B` might represent the same memory cell. Hence the parallelism in this excerpt is not immediately obvious, and its discovery requires interprocedural analysis or function expansion.

Functional languages, on the other hand, meet all the requirements listed above and do not require analysis for the discovery of parallelism [1,15,17,18]. A functional program is a collection of mathematically sound expressions comprised of both intrinsic and user defined functions. These functions are *well defined* and *determinate*. That is, they define unique mappings between domains and ranges. A function passed the same set of values will yield the same results regardless of the environment of invocation. This establishes *referential transparency*, which implies that the evaluation of an expression, or the sharing of its subexpressions, does not change the value it denotes. Consequently, expressions are *side effect free.* The concept of a FORTRAN COMMON block does not exist here. In the absence of side effects, programmers cannot see the target machine; thus the concept of *data* replaces memory, and the concept of *creation* replaces update. Further, in the absence of side effects, programs are implicitly parallel.

Now let us return to the code excerpt discussed earlier in this section, but this time in a functional setting. Here `Conduc` cannot access the value of `Y` as it is not passed as an argument; and `Newque` cannot access the value of `X` as it is not passed as an argument. Further, `Conduc` cannot access `B`, as it is not one of its arguments, nor alter `Y`, as it is passed by value. For the same reason, `Newque` cannot access `A` or alter `X`. Since there are no dependencies between the two calls, they can execute simultaneously. We did not require interprocedural analysis to unravel their behavior. The required information was local and implicit.

Retire Fortran? A Debate Rekindled

David Cann
Computing Research Group, L-306
Lawrence Livermore National Laboratory
P.O. Box 808, Livermore, CA 94550
cann@sisal.llnl.gov

Abstract

In the May 1984 issue of Physics Today, Jim McGraw debated David Kuck and Michael Wolfe on the question of retiring FORTRAN. They addressed such questions as: Is FORTRAN the best tool for decomposing problems for parallel execution? Is FORTRAN the programming language we should carry into the 21st century? Are there any alternatives? While McGraw argued forcefully in favor of retiring FORTRAN, concerns about performance weakened his position. He could not rebut the claim that only FORTRAN could provide the performance required for scientific computing. In this report, we use the current performance of CRAY SISAL, a functional language for large-scale scientific computing, to counter that claim. If McGraw had had our data in 1984, he could have overcome what some say is the only defense in favor of keeping FORTRAN. The results show that we can move beyond the constraints of imperative programming. We can raise the level of abstraction and *retain performance*.

1 Introduction and Motivation

In May of 1984, a debate between Jim McGraw of Lawrence Livermore National Laboratory (LLNL) and David Kuck and Michael Wolfe of Kuck and Associates, appeared in Physics Today [19]. The subject was whether to retire FORTRAN. Eight years have passed, and we wish to reopen the debate and provide further evidence that FORTRAN is not the *sine qua non* of high speed computing.

Many believe that the existing investments in FORTRAN and the quality of existing FORTRAN compilers are preventing a change in programming methodology. Many also feel that support for FORTRAN must continue because the language is familiar and widely available. Unfortunately, the complexity of writing correct parallel programs in FORTRAN is perpetuating today's software crisis. We believe, as did McGraw in 1984, that increased productivity, generality, utility, portability, and performance are only possible if programmers avoid the constraints of imperative languages and adopt a higher level of abstraction. We must escape the morass of imperative semantics and attain a level of abstraction that separates the programmer from the machine, stresses problem definition over the mechanics of solution, and provides complete information to the compiler.

While Kuck and Wolfe would probably have agreed with the above statements, they wisely argued that programmers simply would not use languages that did not allow them to get the performance they required. To this McGraw had no counterargument, for in 1984, FORTRAN and only FORTRAN provided the performance needed for large-scale scientific computing. The intent of this report is to reopen the debate and present overwhelming evidence that functional supercomputing is possible. To this end, we compare the execution performance of SISAL, a functional language for large-scale scientific computing, and FORTRAN on a CRAY Y-MP/864. The CRAYs remain the most heavily used machines at our nation's supercomputer centers, and are the preferred machines of most scientific programmers. Thus, it is here that languages must outperform FORTRAN if they are to replace it.

For completeness and for the benefit of those readers that are not familiar with the principles of functional computing, the next two sections examine the functional programming paradigm, discuss its attributes and advantages, and highlight the salient features of SISAL. In the remaining sections we illustrate the potential inefficiencies of functional computing, present our most recent performance data, and give some closing remarks regarding FORTRAN and the future of high speed computing.