C++ and Fortran 77
Timing Comparisons

Phillip T. Keenan

CRPC-TR93347
November 1993

Center for Research on Parallel Computation
Rice University
P.O. Box 1892
Houston, TX 77251-1892

C++ and Fortran 77 Timing Comparisons

Philip T. Keenan*

November 10, 1993

Abstract

Recently there has been considerable debate within the scientific computation community over the suitability of C++ for large scale numerical computation. This note reports on timing studies of Fortran 77 and C++ conducted on the Intel IPSC/3 Hypercube, the IBM RS-6000 and the Sun Sparc Station 2. Timings are presented for two fundamental algorithms including a dense vector inner product and multiplication of a dense vector by a sparse matrix. Comparison to hand coded assembler routines is also provided in selected cases. The results demonstrate that C and C++ can be just as efficient as FORTRAN and therefore deserve serious consideration.

*Department of Computational and Applied Mathematics, Rice University. Supported in part by a National Science Foundation Postdoctoral Research Fellowship in Mathematical Sciences

Contents

1 Introduction

Recently there has been considerable debate within the scientific computation community over the suitability of C++ for large scale numerical computation. This note reports on timing studies of Fortran 77 and C++ conducted on the Intel IPSC/3 Hypercube, the IBM RS-6000 and the Sun Sparc Station 2. Timings are presented for two fundamental algorithms including a dense vector inner product and multiplication of a dense vector by a sparse matrix. Comparison to hand coded assembler routines is also provided in selected cases. The results demonstrate that C and C++ can be just as efficient as FORTRAN and therefore deserve serious consideration.

A number of hardware and software factors complicate any attempt at comparing timings. The run time of any algorithm depends upon many factors, including

- the exact form of the Fortran 77 or C source code used to implement it,
- the version of the compiler used to compile it,
- the compiler options used, especially the optimization level selected,
- the target machine used to run on, and
- the problem size.

It can therefore be dangerous to generalize empirical results to other machines, compilers, algorithms and problem sizes. Nevertheless, the empirical results presented here provide useful insights. To provide a basis for comparison with other

timing tests, the rest of this introduction documents the machine and compiler versions used in the timing tests. Those wanting a quick summary of the results can find one in the final section of this report.

1.1 Machines

Timing tests were run on the following 3 platforms:

- One `i860` based node of an Intel IPSC/3 Hypercube. By focusing on a single node, issues of parallel communication overhead do not enter in. The node had 16 Megabytes of RAM and an 8k byte data cache.
- An IBM `RS-6000` workstation with 192 Megabytes of RAM.
- A Sun `Sparc` Station 2, with 40MHz clock, 32 Megabytes of RAM, and a Sun-4 floating-point controller version 2, based on a 40MHz TI TMS390C602A-based FPU.

All three machines are modern super-scalar workstations. If they have any vectorizing capability, the vector pipeline is quite short: 3 cycles on the i860 chip. Timings are given below for several alternative formulations of a sparse matrix - vector multiplication. The ranking of the tested algorithms in order of efficiency might change on a traditional vectorizing supercomputer such as a CRAY. However, the intent here is not to study numerical linear algebra algorithms, but to compare the efficiency of the same algorithm implemented in different programming languages. In this respect the results are likely to have wide validity.

1.2 Compilers

The current FORTRAN and C node compilers were used on the i860 platform. These were:

- `if77`: The PGFTN Rel 2.0a node FORTRAN-77 compiler.
- `icc`: The PGC Rel 2.0a node C compiler.

Note that the two compilers are from the same release, making it appropriate to compare them. Indeed, it turns out that they share a great deal of functionality and even generate essentially the same assembly language code in many cases.

The current FORTRAN and C node compilers were used on the RS-6000 platform. These were:

- `f77`: The IBM AIX XL FORTRAN Compiler/6000, Version 2.02.0100.0003. This is the same as `xlf`.

- `cc`: The IBM AIX XL C Compiler/6000, Version 1.02.0000.0000.

Although the FORTRAN compiler is newer, the C compiler produced more efficient code.

The current FORTRAN and C compilers were used on the Sparc platform. In addition, an older C compiler and the GNU C compiler were also examined.

- `f77`: The SC1.0 FORTRAN-77 SPARCcompiler, V1.4, patch release 3 (30Sep1991).
- `cc`: The SC1.0 C SPARCcompiler, V1.1 (1Mar1991).
- `old-cc`: The old (1988) Sparc C compiler, which lacks the `-cg89` compiler option among others. It is included because until these timing tests were done, it was the default C compiler on the Rice CS network.
- `gcc`: The GNU C compiler, version 2.3.3 for Sparc. GNU does not support a FORTRAN compiler at this time.

Although the FORTRAN SPARCcompiler is actually 6 months more advanced than the C SPARCcompiler, the C compiler produced more efficient code.

1.3 Compiler Options

Each compiler family has its own set of command line options controlling optimization. There are too many possible combinations to report on every possibility. However, each compiler was tested at all supported levels of optimization, with and without any extra speed options recommended by the compiler manual.

Many numerical analysts never use optimizations above the basic `-O2` level, in part because some compilers generate incorrect code in certain instances at higher optimization levels! At the least, optimization can change the round off error properties of floating point expressions, thereby introducing errors. However, accuracy checks throughout the test program found only roundoff scale variations in the results across all optimization levels.

For the i860 the optimization levels used were `-O2`, `-O3`, and `-O4`. These were optionally combined with the following speed options:

- `if77: -Mvect`
- `if77: -Mvect -Knoieee`
- `icc: -Mvect -Msafeptr=arg,auto -Mnodepchk`

The `-Knoieee` had no significant effect here. In general it is not recommended, as its use amounts to (possibly) getting extra speed by (probably) giving up several digits of accuracy in the computations. The IEEE standard was carefully designed to produce highly accurate and portable floating point arithmetic. If this flag does make your code run faster it is likely a hint that either the algorithms in it are not numerically robust, or that they make too much use of time consuming operations like division.

For the RS-6000 one simply specifies `-O` to invoke all compiler optimizations. The case of no optimization was included for comparison.

For the Sparc 2 the optimization levels used were `-O2`, `-O3`, and `-O4`. These were optionally combined with the following speed options:

- `f77: -fast`
- `cc: -dalign -cg89`
- `old-cc: -dalign`
- `gcc: -fschedule-insns -fschedule-insns2 -fdelayed-branch`

Note that `gcc` has only the single optimization level `-O`.

1.4 C++ and C

No C++ compiler has yet been mentioned. This is because the standard C++ compiler from AT&T actually translates C++ source code into C source code, which is then compiled using whatever C compiler and options one wishes, for whatever machine one wishes. Thus at the lowest level the comparison is really between specific FORTRAN and C compilers on a given machine. Since C is a subset of C++, any C algorithm is also a valid C++ algorithm and the translation step is trivial.

To the application programmer, however, C++ code can look much nicer than C code. In particular, using an appropriate class library, vector and matrix operations can be specified with "array syntax" as in languages such as FORTRAN 90 and APL. How these operations are implemented is up to the designer of the library. In particular, the designer can select the optimal available routine (written in Assembler, C or even FORTRAN), and use it in the implementation of the class library. This choice is *invisible* to the application programmer and requires *zero* change to the application program. However, proper choice here means that *C++ will always win* against any other compatible language.

When this C++ library implementation is timed, it is denoted `C++ library` in the comparisons below. It is proper to consider this version, as it corresponds to what high performance FORTRAN compilers such as the i860's `if77` do when they

recognize certain special loops, such as dot products, and replace them by calls to hand coded assembly routines. To facilitate this comparison, FORTRAN timings are marked `if77 library` when the `if77` compiler substituted calls to hand coded dot product routines. The `if77` option `-Minfo=loop` makes the compiler mention whenever it makes such a substitution. Note that these substitutions only occur when invoked by the `-Mvect` option.

In order to provide fair comparisons of what can be conveniently accomplished in each high level language, timings are also presented for direct C++ source code implementations. These give a more realistic picture of the timings that can be expected from explicit loops which are implemented for simplicity rather than speed. These cases are denoted `C++` in the timings below.

1.5 Problem Sizes

This report is primarily intended for researchers solving large sparse systems of linear equations such as those arising in the numerical solution of partial differential equations. Generally these problems are too large to fit in the hardware cache on most machines. While timings are provided for small problems (vectors of length 1000) which do fit in cache, the more important timings are those for large problems (8000 or more vector elements), in which cache effects play only a minor role. Timings for large problems (27,000 vector elements) were consistently about 1% slower than for 8000 element problems, and so are not reported on below.

When the Intel Hypercube is used as a parallel machine, issues of communication time arise that are not dealt with in this report. The problem sizes considered here are relevant, however, on a per-processor basis. For instance, a typical large problem involving one million vector elements on 100 processors might put 10,000 on each processor.

2 Dense Inner Product

The first algorithm considered is the inner product of two dense vectors. The plain Fortran 77 version is

```
      doubleprecision function inf(v,w,N)
      integer N
      doubleprecision v(N), w(N)
      integer i

      inf = 0.0
      do 1, i=1,N
         inf = inf + v(i)*w(i)
```

```
 1    continue
      return
      end
```

The plain C version is

```
double inc(double *v, double *w, int N)
{
  double sum = 0;
  int i;
  for(i=0; i<N; i++)
    sum += v[i]*w[i];
  return sum;
}
```

The C++ version, using the Keenan C++ Class Library, is simply to write

```
  double inp = v*w;
```

C++ automatically implements this as a call to `doubleArray::operator*`, which can be implemented by the designer of the class library in any desired language, as discussed above. For purposes of comparison, the pure C++ implementation used in this class library is

```
double doubleArray::operator*(doubleArray& v)
{
  register double sum=0;
  const double *vp = v.elts;
  const int i0 = fi;
  const int i1 = li;
  for(int i=i0; i<=i1; i++)
    sum += elts[i] * vp[i];
  return sum;
}
```

In addition, on the i860 platform three hand coded assembly language versions are available.

One, denoted **hand assembled**, was hand written and tuned by the author as an experiment. This took two days, even for such a simple operation: one to learn the assembly language, another to write, debug and optimize the code! Assembly language is clearly far too primitive to use for any but the most crucial inner loops of a program.

Two professionally tuned hand coded assembly language versions are also available. The `blas` routine is supplied in the third party `-lkmath` BLAS library on the

i860, while the `dotp8` routine is supplied by Intel. In particular, the `dotp8` routine is used in the `if77 library` case.

2.1 Timing Results

27 compiler and optimization option combinations were timed on the Sparc, 23 on the i860 and 7 on the RS-6000.

Table 1 presents the SPARC timing results, Table 2 presents the RS-6000 results, and Table 3 presents the IPSC results. N is the vector length. To save space in Table 1 only the fastest combinations are listed for the old C and GNU C compilers.

All cases were iterated to get timings of 10 seconds, as the real time clock resolution on these machines is not high. Small variations in the last reported decimal digit of the MegaFlop rate can occur from run to run. Each dot product was counted as $2N$ floating point operations. Cache effects decrease as the vector length increase. Cache effects are less noticeable on the IPSC except in the professionally tuned assembly language routines which are written to make maximum use of cache. As a result they run quite fast on small problems. This is not very realistic, however, as cache does not speed up the first execution of a given routine with a given data set: only if the data is accessed again prior to being pushed out of cache by new data will there be any significant speed increase. In real codes one does not repeatedly take the dot product of the same pair of vectors, so except for very small problems these high Megaflop rates are not realistic. However, some manufacturers like to quote rates like those in the $N = 1000$ column as "peak" performance numbers.

Table 1: SPARC Timing Results: Dot Product

Algorithm	Compiler Options	MegaFlops(N=1000)	MegaFlops(N=8000)
plain	f77 -O2	4.1	2.5
plain	f77 -O3	6.1	3.2
plain	f77 -O4	6.1	3.2
plain	f77 -fast -O2	4.5	2.7
plain	f77 -fast -O3	7.2	3.5
plain	f77 -fast -O4	7.3	3.5
plain	gcc -O -f...	4.5	2.7
plain	old-cc -dalign -O4	6.8	3.4
plain	cc -O2	4.1	2.5
plain	cc -O3	6.7	3.2
plain	cc -O4	6.8	3.2
plain	cc -dalign -cg89 -O2	4.5	2.7
plain	cc -dalign -cg89 -O3	8.2	3.6
plain	cc -dalign -cg89 -O4	8.1	3.5
plain	C++ -O2	4.1	2.4
plain	C++ -O3	6.8	3.2
plain	C++ -O4	6.7	3.1
plain	C++ -dalign -cg89 -O3	4.6	2.7
plain	C++ -dalign -cg89 -O3	8.1	3.5
plain	C++ -dalign -cg89 -O4	8.2	3.6
C++ library	C++ -O	7.6	3.5

Table 2: RS-6000 Timing Results: Dot Product

Algorithm	Compiler Options	MegaFlops(N=1000)	MegaFlops(N=8000)
plain	f77	2.6	2.5
plain	f77 -O	40.5	24.5
plain	cc	4.1	3.9
plain	cc -O	40.5	24.5
plain	C++	4.1	3.9
plain	C++ -O	40.5	24.5
C++ library	C++ -O	40.5	24.5

Table 3: i860 Timing Results: Dot Product

Algorithm	Compiler Options	MegaFlops(N=1000)	MegaFlops(N=8000)
plain	f77 -O2	5.0	4.8
plain	f77 -O3	6.5	5.9
plain	f77 -O4	6.5	5.9
if77 library	f77 -Mvect -O2	12.2	12.6
if77 library	f77 -Mvect -O3	12.3	12.6
if77 library	f77 -Mvect -O4	12.3	12.6
plain	cc -O2	5.0	4.8
plain	cc -O3	6.5	5.9
plain	cc -O4	6.5	5.9
plain	cc -Mvect -O2	5.0	4.8
plain	cc -Mvect -O3	6.5	5.9
plain	cc -Mvect -O4	6.5	5.9
plain	C++ -O2	5.0	4.8
plain	C++ -O3	6.5	6.2
plain	C++ -O4	6.5	6.2
plain	C++ -Mvect -O2	5.0	4.8
plain	C++ -Mvect -O3	6.5	5.9
plain	C++ -Mvect -O4	6.5	5.9
blas library	assembler	17.9	9.7
hand coded	assembler	8.2	7.2
C++ library	C++ -O	12.1	12.6

3 Sparse Matrix Multiplying Dense Vector

Dot products are so simple to compute that they do not accurately represent timings for real codes. Indeed, some manufacturers supply professionally tuned assembly language code for dot products. Dense linear algebra in general is well studied and many compilers are tuned by using LINPACK (now superseded by LAPACK) as a benchmark. However, partial differential equation solvers deal with *sparse* linear algebra, which is less well understood. In particular, none of the compilers tested in this study had access to hand tuned assembly code for the case of a sparse matrix multiplying a dense vector. Thus this example gives a more realistic comparison of the abilities of the various compilers.

Multiplying a vector by a matrix is a more interesting example from a computer science perspective as well, as two new issues come in. First, the process involves a double loop, so that reordering the loop or changing between row major and column major matrix storage becomes a possibility. Moreover, the answer is a vector, not a scalar, and hence involves storing into array elements. This raises the specter of data dependencies. The conventional wisdom is that C must be slower in such situations, as it allows arbitrary dependencies, while FORTRAN can do additional optimizations because it prohibits arbitrary dependencies. Fortunately, however, modern compiler writers are aware of this issue and provide a compiler flag that can be used to tell the C compiler that it is safe to do FORTRAN style optimizations on a particular subroutine. As a result, C compilers can produce code just as efficient as FORTRAN compilers create.

In particular, the i860 C compiler option `-Msafeptr=arg,auto` enables FORTRAN style optimizations. On the RS-6000 aggressive optimization is the default, with an option provided to turn it off when needed. The SPARC compiler does not provide such an option, but its timings do not suffer, as shown below. This implies that even the FORTRAN compiler does not find these optimizations profitable on the SPARC architecture.

A sparse matrix M is represented in this example by a double precision matrix D and an integer matrix C. If M is N by N and has at most K non-zero entries per row, then D and C are N by K arrays. The entries are related by the equation

$$d_{ij} = m_{i,c_{ij}}.$$

To avoid needing branch statements to detect unused entries in d inside the inner loop, explicit zeros are placed in d where needed, rather than using a special code value in the c matrix. To give the FORTRAN versions an extra advantage, all indices started at 1 rather than zero.

This representation was chosen as typical of the sort of indirect addressing needed to handle general geometry and general boundary conditions in partial differential

equations. For the numerical example, $K = 7$ and the 7-point stencil of a three dimensional rectangular finite difference code was used for simplicity.

Four different implementations were timed in each language. Two used row-major memory storage for the matrices, and two use column-major storage. For each storage pattern, one routine used a loop of row dot products while the other used a loop of column sums (often called a *saxpy*). It is interesting to note that the fastest running versions were invariably the row major dot products, which are the natural thing a C program would use, but which are quite un-natural in FORTRAN. This is due both to the sparsity representation and to the use of modern super-scalar workstations in this study. The results might be quite different on a traditional vector processor with a long vector pipeline, such as a CRAY, since K is generally small compared to N.

The plain FORTRAN 77 versions of the sparse matrix – vector multiply were

```
c  Column Major Storage, Saxpy Form

      subroutine cef(data,cols,nrows,ncols,dest,src)
      integer nrows, ncols, cols(nrows,ncols)
      doubleprecision data(nrows,ncols), dest(nrows), src(nrows)
      integer r, c

      do 1, r=1,nrows
         dest(r) = 0.0
 1    continue

      do 2, c=1,ncols
         do 3, r=1,nrows
            dest(r) = dest(r) + data(r,c)*src(cols(r,c))
 3       continue
 2    continue
      return
      end

c  Column Major Storage, Dot Product Form

      subroutine cdf(data,cols,nrows,ncols,dest,src)
      integer nrows, ncols, cols(nrows,ncols)
      doubleprecision data(nrows,ncols), dest(nrows), src(nrows)
      integer r, c
      doubleprecision sum
```

```
      do 1, r=1,nrows
         sum = 0.0
         do 2, c=1,ncols
            sum = sum + data(r,c)*src(cols(r,c))
 2       continue
         dest(r) = sum
 1    continue
      return
      end

c  Row Major Storage, Saxpy Form

      subroutine ref(data,cols,nrows,ncols,dest,src)
      integer nrows, ncols, cols(ncols,nrows)
      doubleprecision data(ncols,nrows), dest(nrows), src(nrows)
      integer r, c

      do 1, r=1,nrows
         dest(r) = 0.0
 1    continue

      do 2, c=1,ncols
         do 3, r=1,nrows
            dest(r) = dest(r) + data(c,r)*src(cols(c,r))
 3       continue
 2    continue
      return
      end

c  Row Major Storage, Dot Product Form

      subroutine rdf(data,cols,nrows,ncols,dest,src)
      integer nrows, ncols, cols(ncols,nrows)
      doubleprecision data(ncols,nrows), dest(nrows), src(nrows)
      integer r, c
      doubleprecision sum

      do 1, r=1,nrows
         sum = 0.0
         do 2, c=1,ncols
```

```
            sum = sum + data(c,r)*src(cols(c,r))
 2        continue
          dest(r) = sum
 1     continue
       return
       end
```

The plain C are essentially identical to the FORTRAN versions, except for being written in C notation, and so will be omitted, except for the first, which gives the flavor of the rest.

```
/* Column Major Storage, Saxpy Form */

void cec(data, cols, R, C, dest, src)
     double *data, *dest, *src;
     int *cols;
     int R, C;
{
  int r, c;
  src --;
    /* since for fortran's benefit columns start at 1 */

  for(r=0; r<R; r++)
    dest[r] = 0.0;

  for(c=0; c<C; c++)
    for(r=0; r<R; r++)
      dest[r] += data[c*R+r]*src[cols[c*R+r]];
}
```

The C++ versions are also quite similar and so again only the first will be shown here:

```
// Column Major Storage, Saxpy Form

void ceC(cMatrix& data, cIntMatrix& cols,
            doubleArray& dest, doubleArray& src)
{
  const int c0 = cols.f2;
  const int c1 = cols.l2;
  const int r0 = cols.f1;
  const int r1 = cols.l1;
```

```
  dest = 0;

  for(int c=c0; c<=c1; c++)
    {
      doubleArray& da = data[c];
      intArray& ca = cols[c];
      for(int r=r0; r<=r1; r++)
        dest(r) += da(r)*src(ca(r));
    }
}
```

These C++ versions are somewhat less efficient than the corresponding C versions. This is in part because the underlying implementation of vectors and matrices used here is not the most efficient possible, within C++, thus preventing certain optimizations. The Keenan C++ Class Library is still evolving and the experience of these timing studies will influence the next revision. However, even the existing library allows the programmer to access C++ vectors and matrices as C arrays. The C++ library versions cited in the timings below use this fact to advantage by calling the optimal C routine instead.

3.1 Timing Results

108 compiler, option and implementation combinations were timed on the Sparc, 80 on the i860 and 28 on the RS-6000. To reduce the volume of information, the results are presented in a summary format.

Tables 4, 5, and 6 present, for each machine separately, the fastest methods in each category. That is, for each combination of language, memory storage layout and algorithm (dot or saxpy), the set of compiler options which produced the best time on vectors of length 8000 is reported.

As before N is the vector length. All cases were iterated to get timings of 10 seconds, as the real time clock resolution on these machines is not high. Small variations in the last reported decimal digit of the MegaFlop rate can occur from run to run. Each sparse matrix product was counted as $8N$ floating point operations. Cache effects are less noticeable here because of the larger storage requirements for matrices.

In general there was little difference between the `-O3` and `-O4` levels.

Table 4: SPARC Timing Results: Fastest Methods: Sparse Matrix

Algorithm	Compiler Options	MegaFlops(N=1000)	MegaFlops(N=8000)
row, dot	f77 -fast -O4	2.7	2.5
col, dot	f77 -fast -O3	2.7	2.3
col, saxpy	f77 -fast -O4	2.4	1.7
row, saxpy	f77 -fast -O4	1.1	0.9
row, dot	cc -dalign -cg89 -O4	2.7	2.5
col, dot	cc -dalign -cg89 -O4	2.8	2.4
col, saxpy	cc -dalign -cg89 -O3	2.1	1.6
row, saxpy	cc -dalign -cg89 -O4	1.1	0.9
row, dot	C++ -dalign -cg89 -O3	2.0	2.0
col, dot	C++ -dalign -cg89 -O3	2.1	1.9
col, saxpy	C++ -dalign -cg89 -O4	1.8	1.4
row, saxpy	C++ -dalign -cg89 -O4	0.6	0.6
row	C++ (library)	2.7	2.5
col	C++ (library)	2.8	2.4

Table 5: RS-6000 Timing Results: Fastest Methods: Sparse Matrix

Algorithm	Compiler Options	MegaFlops(N=1000)	MegaFlops(N=8000)
row, dot	f77 -O	13.1	12.8
col, dot	f77 -O	13.2	12.5
col, saxpy	f77 -O	11.3	9.6
row, saxpy	f77 -O	5.7	5.3
row, dot	cc -O	13.4	13.0
col, dot	cc -O	13.7	13.0
col, saxpy	cc -O	11.3	9.6
row, saxpy	cc -O	5.1	4.8
row, dot	C++ -O	6.2	6.1
col, dot	C++ -O	4.9	4.8
col, saxpy	C++ -O	4.5	4.2
row, saxpy	C++ -O	2.6	2.5
row	C++ (library)	13.3	13.1
col	C++ (library)	13.7	13.0

Table 6: i860 Timing Results: Fastest Methods By Language: Sparse Matrix

Algorithm	Compiler Options	MegaFlops(N=1000)	MegaFlops(N=8000)
row, dot	f77 -Mvect -O4	4.0	4.0
row, saxpy	f77 -Mvect -O4	3.8	3.7
col, saxpy	f77 -Mvect -O4	3.3	3.2
col, dot	f77 -Mvect -O4	3.9	3.2
row, dot	cc -O4	4.0	4.0
col, dot	cc -O3	4.0	3.3
col, saxpy	cc -O4	2.7	2.6
row, saxpy	cc -O4	1.4	1.4
row, dot	C++ -O4	2.7	2.7
col, dot	C++ -O4	3.1	2.5
col, saxpy	C++ -O4	2.0	1.9
row, saxpy	C++ -O2	0.9	0.9
row	C++ (library)	4.0	4.0
col	C++ (library)	3.9	3.1

4 Summary

For inner products, Tables 7, 8, and 9 present the fastest methods in each language category, for vector length 8000.

Table 7: SPARC Timing Results: Fastest Methods: Inner Product

Algorithm	Compiler Options	MegaFlops(N=8000)
plain	f77 -fast -O4	3.5
plain	gcc -O -f...	2.7
plain	old-cc -dalign -O4	3.4
plain	cc -dalign -cg89 -O3	3.6
plain	C++ -dalign -cg89 -O3	3.6
library	C++ (library)	3.5

Table 8: RS-6000 Timing Results: Fastest Methods: Inner Product

Algorithm	Compiler Options	MegaFlops(N=8000)
plain	f77 -O	24.5
plain	cc -O	24.5
plain	C++ -O	24.5
library	C++ (library)	24.5

For sparse matrices, Tables 10, 11, and 12 present the fastest methods in each language category, for vector length 8000.

Table 9: i860 Timing Results: Fastest Methods By Language: Inner Product

Algorithm	Compiler Options	MegaFlops(N=8000)
plain	f77 -O3	5.9
library	f77 -Mvect -O3	12.6
plain	cc -O3	5.9
plain	C++ -O3	5.9
library	C++ (library)	12.6
BLAS	assembler	9.7
hand-coded	assembler	7.2

Table 10: SPARC Timing Results: Fastest Methods: Sparse Matrix

Algorithm	Compiler Options	MegaFlops(N=8000)
row, dot	f77 -fast -O4	2.5
row, dot	gcc -O -f...	2.4
row, dot	old-cc -dalign -O4	2.4
row, dot	cc -dalign -cg89 -O4	2.5
row, dot	C++ -dalign -cg89 -O3	2.0
row	C++ (library)	2.5

Table 11: RS-6000 Timing Results: Fastest Methods: Sparse Matrix

Algorithm	Compiler Options	MegaFlops(N=8000)
row, dot	f77 -O	12.8
row, dot	cc -O	13.0
row, dot	C++ -O	6.1
row	C++ (library)	13.1

Table 12: i860 Timing Results: Fastest Methods By Language: Sparse Matrix

Algorithm	Compiler Options	MegaFlops(N=8000)
row, dot	f77 -Mvect -O4	4.0
row, dot	cc -O4	4.0
row, dot	C++ -O4	2.7
row	C++ (library)	4.0

References

[1] G. GOLUB AND C. V. LOAN, *Matrix Computations, 2nd. ed.*, Johns Hopkins, 1989.

[2] S. LIPPMAN, *The C++ primer, 2nd ed.*, Addison Wesley, 1992.

[3] B. STROUSTRUP, *The C++ Programming Language, 2nd ed.*, Addison Wesley, 1991.