Effective computing at CNLS

(May 26, 2000)

Abstract

This paper reviews current computational capabilities of CNLS: available machines, hardware, and software (compilers and optimized libraries). The computational performances of different architectures are compared using a simple benchmark which model common scientific simulations. These results are analyzed and practical recipes for effective computational usage of available resources are formulated.

I. INTRODUCTION

Let assume that somebody got brand-new CNLS computer account and started using this unfamiliar and attractive computer cluster for some kind of scientific computations. Commercial packages such as Mathematica and Matlab are readily available if numerical modeling is not complicated. However, many problems require significant computer power and could not be solved with these convenient programs which are generally unable to utilize all computational resources. Only customary programs written on good old languages such as Fortran or C are able to handle these heavy-duty calculations efficiently. If this is the case and the code programming is completed, the next step is how to start calculations without spending extra time rediscovering "the wheel". One needs to know the answers to two questions. The first question is very basic: what are the "KEYWORDS" for calling compilers, libraries, and other system utilities in these particular CNLS systems? The second question is more advanced: what are the capabilities of different computers, how are they suited for my computational needs, what are the little tricks to get the best of computer resources? Although we all have the basic knowledge and understanding of UNIX environment and some programmer languages, introduction to the new Unix system is always inconvenient and sometimes painful.

This paper may give some clues and tips to answer to the above questions and may be useful for both novice and experienced users. Sec. II describes hardware resources available in CNLS and their relevant technical specifications such as clock-speed, RAM, cache, etc. Sec. III analyze available software utilities (compilers and libraries). In Sec. IV computational performances of different systems are compared using simple tests which mimic typical scientific simulations and impact of software and hardware factors on computational efficiency is analyzed. Finally, some hints for efficient computer use are formulated in Sec. V.

II. COMPUTATIONAL RESOURCES

The CNLS computers represent a complex mix of different architectures. Most computers are PCs featuring one or two Intel Pentium II (III) processors which serve as workstations and personal desktop machines. They typically have 400-500 MHz speed and 256-512 MB of RAM[1]. Older Pentium Pro (200 MHz) machines are obsolete and are going to be gradually replaced. The old SGI server (`hermit`) features 4 R10000 CPU (180 MHz) and 1.256 GB of RAM. New Sun Ultra-Sparc II server has 4 CPU (450 MHz) and shared 4Gb of RAM. Newer Digital Alpha machines (`tinamou` and `martin`) each have dual 21264 CPU running at 500 MHz and 2 GB of RAM[2]. The oncoming cluster with 5 nodes will feature dual Alpha 21264 CPU running at 600 MHz and 1-2 GB of RAM in each node.

In addition available older Avalon cluster has 140 nodes. Each node features single 21164 Alpha processor running at 533 MHz and 128 MB of RAM. Avalon is physically located outside of CNLS building.

The other CNLS computers such as Suns and SGIs with R8000 CPU are not very suitable for computations and will not be considered here (however, they feature many software applications such as Showcase on SGI and may be useful for other purposes). The full list of computers is available on the CNLS website `http://cnls.lanl.gov/Internal/Computing/Hardware/hardware.html`.

Detailed specifications of computers:

-
  ```
  PC (e.g.  monjita, barbet)
  CPU: 1(2) Pentium II(III) (266-500 MHz)
  Memory:  128 MB -1000 MB
  Secondary Cache:  512 K
  Operating System:  Linux 2.2.X (Red Hat 6.X)
  ```

-
  ```
  SGI (hermit)
  CPU: 4-R10010 (180 MHz)
  Memory:  1.256 GB
  Secondary Cache:  1MB
  Operating System:  IRIX 6.5
  ```

-
  ```
  Sun (nikita)
  CPU: 4-Ultra-SPARC II (450 MHz)
  Memory:  4 GB
  Secondary Cache:  4MB
  Operating System:  SunOS 5.8
  ```

-
  ```
  Alpha-Dec (tinamou, martin)
  CPU: 2-Alpha 21264DP (500 MHz)
  Memory:  2 GB
  Secondary Cache:  4MB
  Operating System:  Digital UNIX (Tru64Unix) V4.0F (Rev.  1229)
  ```

[1]CPU and memory specifications may be found at each machine in `/proc/cpuinfo` and `/proc/meminfo`, respectively

[2]Because of the bug in operational system currently each standalone job is limited to the size of 115 MB

- `Alpha-Linux (oncoming cluster with 5 nodes)`
  `CPU: 2-Alpha 21264DP (666 MHz)`
  `Memory:  1-2 GB`
  `Secondary Cache:  4MB`
  `Operating System:  Linux 2.2.X (Red Hat 6.X)`

- `Avalon (Alpha-Linux, cluster with 140 nodes)`
  `CPU: 1-Alpha 21164DP (533 MHz)`
  `Memory:  256 MB`
  `Secondary Cache:  1MB`
  `Operating System:  Linux 2.1.125 (Red Hat)`

III. COMPILERS AND LIBRARIES

Fortran 77 (90) and C (C++) are the most common programming languages for scientific computations. Compilers are required to generate executable binaries. Typically executable compiled on one computer will run on all other similar machines. However it will not run on computers with other architectures. (e.g. program compiled on PC will run on all other PCs but will not run on SGI and Alpha machines). The very same code could be compiles on one machine on the fly, but compiler on another computer will give severe complains. However, the code written on generic language is usually portable (compatible with different compilers and computers).

Overview of compilers available on CNLS computers:

- Each PC has default GNU (non-commercial) Fortan 77 (g77), C and C++ (gcc and g++) compilers. They are compatible (allow Fortran and C program mixing) and provide usually average performance. gcc is however far superior to fortran. There are several commercial compilers for PC such as Portland (`http://www.portland.com`), Absoft (`http://www.absoft.com`), NAG (`http://www.nag.com`). These usually have compatible suit of Fortran 77, Fortran 90 (95), High Performance Fortran, C, C++, Debugging tools, and provide excellent performance. For example, free test-drive of these utilities is available for Portland package.

- SGI IRIX has integrated Fortran 77 (f77), Fortran 90 (f90), C (cc) and C++ (CC) commercial package. These compiles are compatible and perform very well. However, hermit in particular does not have all system libraries and manuals which may create some problems.

- Sun has integrated Fortran 77 (f77), Fortran 90 (f90), C (cc) and C++ (c++) commercial package. These compiles are compatible and provide a lot of options.

- Alpha-Dec Unix-64 has also integrated Fortran 77, 90, 95 (f77, f90, f95), and C (cc) commercial suit. Fortran is however better developed then C. Fortran compilers provide excellent performance.

- Alpha-Linux has commercial Compaq Fortran package (basically similar to Unix-64 system). It includes Fortran 77 and 90 (fort). These systems have also GNU C and C++ (gcc) compiler. Compaq claims that fort and gcc are compatible.

- Avalon cluster has MPI fortran (mpif77) and MPI C (mpicc). They are based on the GNU tools g77 and gcc. The performance of fortran compiler is below then average. Note that Compaq Linux compiler on 21264 machines could be used to compile programs for Avalon 21162 CPU (with -arch EV56 flag) with better performance then MPI compilers (unless you would like to pursue parallel programming).

Using libraries greatly simplifies the programmer life. Library is a collection of precompiled object files gathered into a single file. Each object file contains several common routines such as matrix multiplication, fourier transform, solution of linear system of equations, etc. In the optimized library these routines are written on the low-level programming languages to use system resources more efficiently which may significantly increase performance. Typical libraries are: the Basic Linear Algebra Subprograms (BLAS) including vector-vector (BLAS1), matrix-vector (BLAS2) and matrix-matrix (BLAS3) operations, the Linear System and Eigenproblem Solver (LAPACK) which uses BLAS and does not need to be optimized, Fast Fourier Transforms (FFT), etc. The codes for these and many other libraries are freely available on (`http://www.netlib.org`). However, precompiled optimized libraries are not readily available.

Overview of optimized libraries available on CNLS computers:

- Excellent BLAS and FFT optimized libraries freely downloadable from `http://www.cs.utk.edu/~ghenry/distrib/` are installed at `/packages/lib/lsblaspii1.2f_03.00.a` (BLAS) and `/packages/lib/lsfftppro1.1h_08.98.a` (FFT).
- SGI IRIX has integrated system BLAS library `/usr/lib64/libblas.so`.
- Sun has integrated optimized Sun Performance Library which includes LAPACK, BLAS, FFT, and LINPACK routines. This library could be linked using `-xlic_lib=sunperf`.
- Alpha-Dec Unix-64 and Alpha-Linux have commercial optimized Compaq DXML and CXML, respectively, libraries. DXML (CXML) includes BLAS, Signal Processing (FFT), Sparse Linear System Processing, and LAPACK subprograms. They could be linked using `-ldxml` (or `-lcxml`) syntax.
- The optimized libraries on Avalon are not available by default. Non-official GEMM library written by Kazushige Goto was used to perform the tests.

Below are the illustrative examples of Makefiles used to compile `bench1.f` and `bench2.f` fortran 77 codes and link BLAS library which have been used for benchmarks described in the next section. Compiler flags invoke basic optimization features, but may be not the very best combination of compiler options. `-O2` option was used for SGI fortran since the quality of the compiler is so good that `-O3` or `-OPT:Olimit=0:roundoff=3:IEEE_arithmetic=3` aggressive optimizations produce programs with performance close to that of optimized library.

- Makefile for PC (libpc-blas.a is BLAS (version 1.2F) library from `http://www.cs.utk.edu/~ghenry/distrib/`)

```
.SUFFIXES : .o .f
.f.o:
        g77 -c -O3 $<
FILES = bench1.o \
        bench2.o

bench:   $(FILES)
        g77 -O3 -o bench $(FILES) libpc-blas.a
```

- Makefile for SGI

```
.SUFFIXES : .o .f
.f.o:
        f77 -c -64 -mips4 -O2 $<
FILES = bench1.o \
        bench2.o

bench:   $(FILES)
        f77 -64 -mips4 -o bench $(FILES) /usr/lib64/libblas.so
```

- Makefile for Sun

```
.SUFFIXES : .o .f
.f.o:
        f77 -c -fast -w $<
FILES = bench1.o \
        bench2.o

bench:   $(FILES)
        f77 -fast -o bench $(FILES) -xlic_lib=sunperf
```

- Makefile for Alpha-Tru64Unix

```
.SUFFIXES : .o .f
.f.o:
        f77 -c -fast -static -w $<
FILES = bench1.o \
        bench2.o

bench:   $(FILES)
        f77 -fast -o bench $(FILES) -ldxml
```

- Makefile for Alpha-Linux

```
.SUFFIXES : .o .f
.f.o:
        fort -c -fast -static -w $<
FILES = bench1.o \
        bench2.o

bench:   $(FILES)
        fort -fast -o bench $(FILES) -lcxml
```

- Makefile for Avalon

```
.SUFFIXES : .o .f
.f.o:
        mpif77 -c -O3 $<
FILES = bench1.o \
        bench2.o

bench:   $(FILES)
        mpif77 -o bench $(FILES) blas-opt.a
```

IV. BENCHMARK RESULTS

Let first estimate what level of performance we would expect from these machines before jumping into benchmarking. Generally computational performance depends from both hardware architecture and software implementation:

- Hardware factors

  1. Processor clock-speed. The higher is the better. Modern computers could reach up to 1GHz frequency (1.5 GHz is expected by the end of 2000).

  2. Processor and motherboard architecture (e.g. 32 bits (PC) or 64 bits (SGI,Alpha)). The chip design has a huge impact on computer performance.

  3. Bus speed and memory speed. These frequencies are typically much lower then the CPU clock (100-200 MHz). They affect overall data flow in the system.

  4. Primary and secondary cache size. The bigger is the better. Cache is a small piece of memory working at (or close to) the processor frequency (much faster than the main memory). It provides very fast data storage and exchange for CPU.

- Software factors

  1. Quality of the compiler. Well written compiler could significantly improve program's executable time.

2. Programming style. Careful programming could free many program's bottlenecks and help compiler to generate faster executables.

3. Optimized libraries could make a real difference.

These computer characteristics are based on personal experience and private communications and therefore may not be fully objective.

1. PC. In the last few years performance of these machines have grown enormously. Their orientation to the fast integer-point calculation important to the Office applications has radically changed in 1995 with the introduction of revolutionary Pentium Pro processor which performed for floating point operations extremely well (this is important for example for 3D graphics and most computational problems). PCs have invaded the workstation market developing at much faster pace compared to traditional brands (IBM, HP, SGI, DEC). Currently fierce competition between Intel and AMD resulted in incredible CPU clock-frequency jump from 500 to 1,000 MHz during period of less then eight month (08/99-03/00). Now PCs provide a level of computational power comparable with the best workstations at much lower price.

2. SGI. Development of RXXXXX RISC line of processors was slow and could not compete with PC. Three year old R10000 CPU is still on the market today. Traditional strength of SGI enhanced by CRAY technology is high system bandwidth which have resulted in very efficient (and expensive) Origin 2000 giants. Although this performance is heavily crippled with the low CPU frequency (e.g. hermit has 180 MHz and 300 MHz is the best you can get for today).

3. Sun. Similar to SGI slow development of Ulta-Sparc line of processors based on RISC architecture currently does not make Suns top performers. Sun's highest clock-speed of 450 MHz is already far behind PC and Alpha. However, good old workstation architecture still provides a respectable overall system throughoutput.

Development of RXXXXX RISC line of processors was slow and could not compete with PC. Three year old R10000 CPU is still on the market today. Traditional strength of SGI enhanced by CRAY technology is high system bandwidth which have resulted in very efficient (and expensive) Origin 2000 giants. Although this performance is heavily crippled with the low CPU frequency (e.g. hermit has 180 MHz and 300 MHz is the best you can get for today).

4. Alpha. Nowadays the CPU line of this brand based on RISC architecture is fast developing and still stays ahead of PC. The older systems (21164) were flawed with low memory bandwidth which heavily slowed their performance for large-size computations. However machines based on the 21264 processor seem free of this drawback, and combined with the high MHz speed we would anticipate the best performance (not price) for these computers. The oncoming third generation of Alpha CPU (21364) would raise the computational speed even higher.

Standard benchmarks such as SPECint and SPECfp are usually used to test the processor performance. However, we wanted some very simple, practical and illustrative benchmark for the floating point computations which are the central piece of every scientific simulation: square matrices multiplications were conducted to estimate the overall system performance. The matrices were filled with the double precision random real numbers. The memory bandwidth performance then could be seen by varying the matrix size. This operation has been repeated several times (especially for small sized) to get average statistics and comparable overall timing. Used compilers and their options are given in the Makefiles in the previous Section[3].

Several different benchmark setups could model most typical computational tasks:

I Small memory computations such as Monte-Carlo simulations:

- **a)** Matrix size 2X2 (4,000,000 times) Program size 10K
- **b)** Matrix size 10X10 (400,000 times) Program size 13K

II Moderate memory computations, most common case in scientific modeling:

- **a)** Matrix size 100X100 (4,000 times) Program size 250 K
- **b)** Matrix size 500X500 (40 times) Program size 6 MB

III Large memory computations such as heavy quantum chemistry calculations:

- **a)** Matrix size 2,000X2,000 (4 times) Program size 96 MB
- **b)** Matrix size 4,000X4,000 (4 times) Program size 384 MB

Memory and floating point performances interplay is illustrated with three different ways of matrix multiplication:

(i) Using external optimized BLAS library.

(ii) Explicit programming ("good style"):

```
DO I=1,N
 DO J=1,N
   C(I,J)=0
  DO K=1,N
   C(I,J)=C(I,J)+A(K,I)*B(K,J)
  ENNDO
 ENDDO
ENNDO
```

Here the load to the memory is minimized since CPU calls data from memory allocated for A and B mostly *sequentially* (i.e. A(1,I) and A(2,I) elements are "neighbors" in the memory array).

(iii) Explicit programming ("bad style"):

[3]450 MHz Pentium III PC with 512 MB of RAM has been used for benchmarks

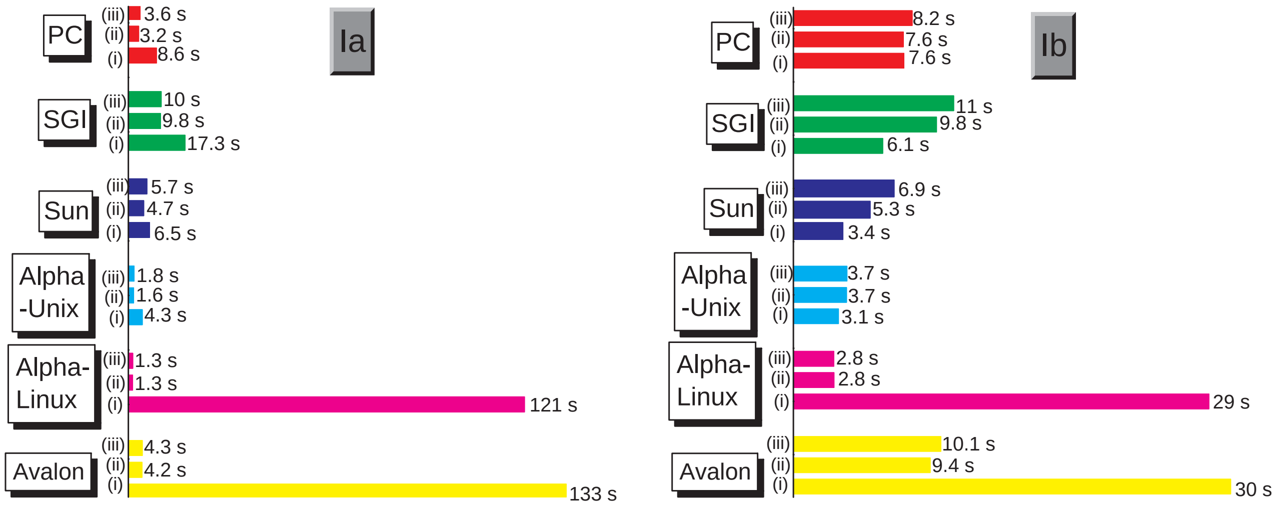

FIG. 1. Timing results for small memory jobs: 2X2 [10K] (Ia) and 10X10 [13K] (Ib) matrices. (i) optimized library, (ii) minimal memory load, (iii) maximal memory load.

```
DO I=1,N
 DO J=1,N
   C(I,J)=0
  DO K=1,N
   C(I,J)=C(I,J)+A(I,K)*B(J,K)
  ENNDO
 ENDDO
ENNDO
```

Here the load to the memory is maximized since CPU calls data from memory allocated for A and B mostly *randomly* (i.e. A(I,1) and A(I,2) elements are separated by N other elements in the memory array). Note that programmer should not care about correct index order when using the optimized library (timings for AB, A^TB, AB^T, and A^TB^T multiplications are about the same).

These tests were performed on five different systems described above: PC (Pentium III 450 MHz with 512 MB of RAM), SGI, Alpha-Tru64Unix, Alpha-Linux, and Avalon. All results are given for a single processor performance without invoking any parallel programming utilities. We are showing the actual timings in seconds which provide easier estimate of computational capabilities then MFlops.

Fig. 1 shows the results for small memory (Monte-Carlo like) jobs. This setting does not test neither floating point nor raw memory performance, rather the ability of the system to initialize the library (4,000,000 and 400,000 times) or `DO` loops (28,000,000 and 2,800,000 times). It clearly demonstrates that using optimized libraries does not make any sense in this case: computer spends more time calling library then actually utilizing it. Program is small and memory performance is not crucial (ii and iii timings are about the same). Going to the size 10 (Ib) makes all cases (i-iii) perform similarly, except Alpha-Linux and Avalon machines where optimized libraries still create huge bottleneck.

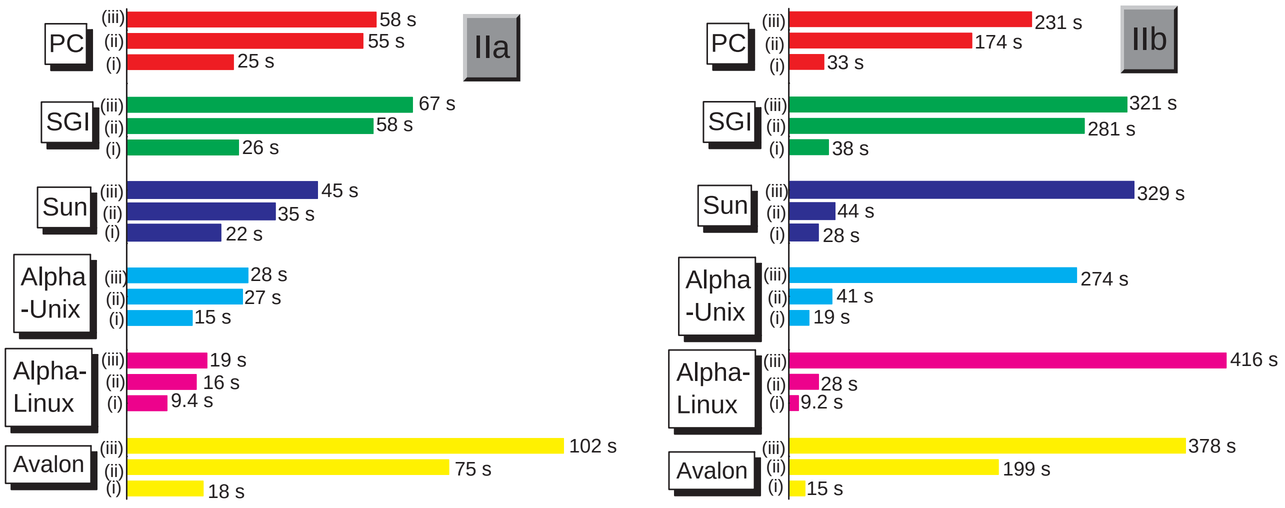

FIG. 2. Timing results for medium memory jobs: 100X100 [250K] (IIa) and 500X500 [6MB] (IIb) matrices. (i) optimized library, (ii) minimal memory load, (iii) maximal memory load.

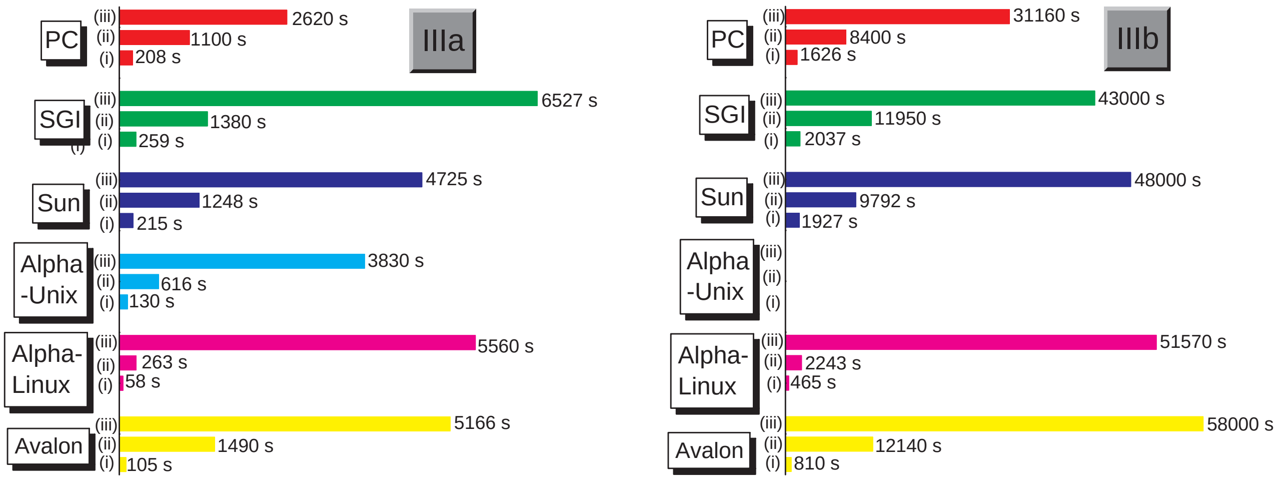

FIG. 3. Timing results for large memory jobs: 2000X2000 [96MB] (IIIa) and 4000X4000 [384MB] (IIIb) matrices. (i) optimized library, (ii) minimal memory load, (iii) maximal memory load. Alpha-Tru64Unix results are not available for IIIb because of 110MB memory limitation.

Fig. 2 shows the results for medium memory jobs. We note that optimized libraries start to make a difference. The results for cases IIa(ii) and IIa(iii) are mostly similar (binaries are small and may fit into a cache and therefore the main memory does not show up. However, for IIb the memory bottleneck is obvious: (iii) computations take much more time then (ii), especially on Alpha machines. In terms of floating point performance, PC, Sun, and SGI have about the same efficiency, whereas Alphas are faster by a factor of 2-3. Surprisingly, Alpha-Linux is significantly faster then the Alpha-Unix and Avalon for optimized libraries (i) case, even though their clock-frequencies are not very different. Poor Avalon compiler produces (ii) executable which performs on par with PC and SGI.

Fig. 3 shows the results for large memory jobs. Now we could see the extreme power of optimized libraries which provide a factor of ∼10-100 efficiency increase compare to perfor-

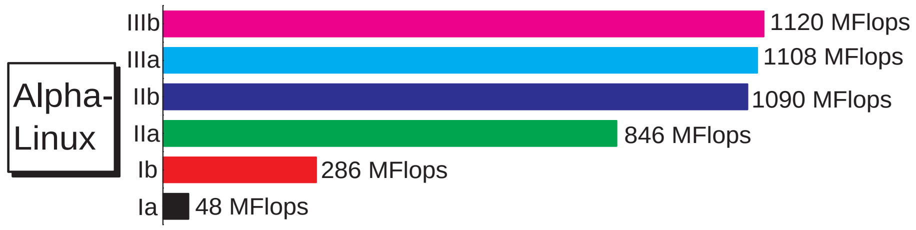

FIG. 4. Floating-point performance of Alpha-Linux for Ia-IIIb settings calculated for the best timing results as $2N^3/10^6t$, where N is a matrix size.

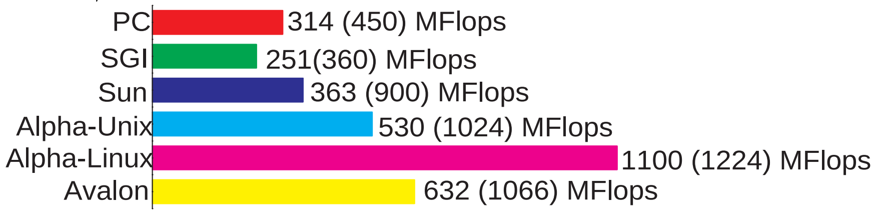

FIG. 5. Comparison of the best floating-point performances of different machines. MFlops values are calculated as $2N^3/10^6t$, where N is a matrix size. The theoretical peak performances are given in round parenthesis.

mance of codes compiled without them (ii and iii). All trends we observe in the previous case (Fig. 2) are extremely exaggerated here. Poor programming (case iii) chokes up all systems and is unacceptable for this kind of jobs. Programs compiled without optimized library (ii) perform similarly on PC, Sun, SGI, and Avalon, and faster on Alpha-Unix and Alpha-Linux. Avalon is much faster if the optimized library is used. The champion is Alpha-Linux which significantly outruns the other systems.

To compare the computational efficiency in cases Ia-IIb, actual floating-point performance (MFlops) has been calculated for Alpha-Linux and displayed in Fig. 4 (the other computers show similar trends). This graph demonstrate that small-memory jobs are not very efficient since the computer spends most of the time initializing loops as was pointed above. If this is the case, vectorization of the code may be useful (e.g. computing several Monte-Carlo points in one run). Finally, comparison of the very best floating point speed for different computers are shown in Fig. 5. These results were obtained using optimized libraries. The theoretical peak speed is measured by the CPU clock-frequency, whereas actual computer speed depends on many other factors. That is why the system architecture is so important to achieve higher actual performance. PC with optimized library could reach up to 70% of its theoretical peak. This is not bad considering that CPU architecture is congested with compatibility with old i086, i286 and i386 instructions. SGI optimize library is not great, even proper compiler options could increase computer speed up to 80%. It is interesting to note fairly low Alpha-Tru64Unix performance which is even slower then the older Avalon's Alpha processors. It can not be attributed to compiler and library since essentially the same Compaq compiler and library are so great on Alpha-Linux. Curiously, Alpha-Tru64Unix and Sun show the peak performance for IIb case (100X100 size), whereas all other machines have the top for the largest matrices (IIIa and IIIb). For Alpha this low performance may be related to the hardware problems (memory limitation?) or operational system. Sun may have low overall memory performance. On the other hand, Alpha-Linux tops almost maximum possible number-crunching power. Even though the older Avalon's

2164 Alpha CPU has large theoretical peak performance, it could be rarely achieved, and newer 21264 have significantly better overall performance.

The above results do not pretend to show the best system performances one able to get, but rather serve a purpose of comparison of different hardware architecture and illustrate the importance of software factors onto computer efficiency. The differences in timing results for (i)-(iii) cases in the above tests are extreme. Actual code should show less variations.

V. CONCLUSION: TIPS FOR RUNNING SCIENTIFIC SIMULATIONS

Let discuss first whether the parallel programming is needed for scientific applications. General recommendation would be: do not go there if you do not realize clearly what you have to do. Writing parallel programs is usually more complicated task compared to "standard" programming, special care should be taken about code portability. Using parallel programs which utilize only 2-3 processors is barely justified. On the other hand, examples in the previous section show that very simple changes such as linking the optimized library, could drastically increase efficiency by a factor of ∼10-100. However, using parallel environment for large-scale calculations in complicated and well-developed computational packages could be very beneficial.

These conclusions are follows from results described in the previous section:

1. In the small memory Monte-Carlo like jobs (<500K) using optimized libraries may not be helpful if the library routines are repeatedly called many times. It is useful to avoid repetitive initializations of loops (e.g. `DO`) and conditional statements (e.g. `IF`) as much as possible. Program vectorization (e.g. computing several Monte-Carlo points in one run) may greatly increase its efficiency.

2. In the medium-size jobs (500K-10MB) linking optimized libraries is a real plus. It significantly simplify programing and increases code performance. Although it is generally recommended to write the programs correctly from the scratch, spending extra time optimizing the code may be justified only if it runs slowly, otherwise it is not worth your time.

3. In the large memory jobs (>10MB) using optimized libraries is necessary. Careful and correct programming will help to avoid unnecessary memory load (see (iii) results). Generally, each program has few places (or bottlenecks) which consume most of computational time. Finding and optimizing them could really make your program to fly.

In addition we could formulate common recommendations for efficient computing:

- Using compiler optimization options such as loop unrolling, enabling software pipelining, etc. could significantly speed up calculatios. For example, programs compiled on SGI with `-O3` and `-OPT:Olimit=0:roundoff=3:IEEE_arithmetic=3` optimizations run with efficiency comparable to optimized library performance.

- In addition to available libraries using pre-defined routines for solving standard computational problems is highly recommended. A wealth of such routines and libraries written on Fortran and C is freely downloadable from Netlib: (`http://www.netlib.org`).

These programs are bug-proof, correctly written and portable. You could easily find desirable routine by searching or browsing Netlib. The file then has to be downloaded *with dependencies* and linked to your code. However, the blind usage of these routines could be dangerous. At least some prior knowledge of computational algorithm utilized in the code and how this technique will work for your problem is required.

- Always compare your program demands with the computer hardware capabilities before submitting the job. For example, executing programs with size larger then computer physical RAM will not work. At best, computer will not accept the job. In the worse case program execution may take forever since computer will start to use a part of Hard Drive (Swap) as RAM, which is extremely slow. This may also happen if the total size of all running processes will exceed the RAM (e.g. the net size of your code and another program running on the second computer CPU).

- The choice of optimal computer for executing your codes may require little experience and experiments. The net efficiency depends on complicated interplay of CPU clock-speed, memory and Cache performance, and software factors. It is hard to say beforehand which computer is best suited (see, how diverse the benchmark results are). Several trial runs may be helpful.

- Finally, let discuss what is the acceptable wallclock time for program execution. Couple seconds - few hours are the most typical cases. More demanding computations run overnight, over-weekend, week, which is also acceptable. However, longer runs (weeks, months) may not be practical. Computer reboots because of, for example, power surges, will kill all running programs. In these long runs the restart options could be a real plus.