Programming exercises on RISC processors and shared memory machines

PDC Summer School 2001

Anders Ålund

1 About this exercise

The aim of this exercise is to give an introduction to Fortran optimization for RISC processors and parallelization of programs on shared memory machines. The exercise consists of two parts. The first one contains a few small examples where you get used to the available tools and review some of the optimization ideas from the lectures. In the second part you will try your skills on a "real" code.

In this exercise you will use the parallel shared memory machine `boye.pdc.kth.se`, a 12 processor Silicon Graphics Onyx2. You may also try some of the exercises on an SMP-node of the IBM SP2, `nf01n01.pdc.kth.se`, or on the workstation on your desk. Makefiles for different computer systems are found in the source directory.

Some modern compilers are very advanced and many techniques used to achieve high performance are "known" by the compilers and will automatically be performed, provided the optimization level is high enough. Sometimes extra "hints" in the form of *compiler directives* must be given.

2 Introduction to code optimization

The optimization process is iterative. First you establish an initial *benchmark* of your code and then repeatedly apply *optimization techniques* to improve performance. Always check the results to assure correctness and repeatability. Use the *performance tools* to let the computer collect and present statistics about the execution of your code.

2.1 Benchmark

After choosing a performance measurement criterion and selecting a *timer* on the SGI Onyx2, you should establish an initial benchmark of performance. You will use this initial benchmark to measure performance gains and verify correctness after optimizing your code.

2.2 Optimization techniques

The available techniques to improve performance of the code are of two kinds: One is to modify the command line for the compiler, i.e., requesting the compiler to automatically improve performance of the code. This is often enough to achieve good performance but with "vintage" code (use of COMMON, SAVE etc.) the compiler fails to improve performance or the "optimized" code produces incorrect results. You can also request the linker to use optimized versions of library routines. The second technique involves modification of the source code. By the use of *compiler directives* in your source code you give the compiler useful hints or directives allowing it to apply more aggressive optimization. By making changes to the Fortran source code you may implement

a more efficient algorithm. A combination of the two methods is often necessary to get the best performance out of your application.

2.3 Some useful command-line options to `f90` on the SGI Onyx2

The command line options of the `f90` compiler are described in the `man`-page. Below are listed some of the more useful ones.

- To generate code producing a *runtime profile* when executed and examine statistics with `prof` use the command sequence

```
f90 -o prog prog.f90
ssrun -pcsamp ./prog
prof prog.pcsamp.mxxxxx
```

- To automatically inline subprograms `sub1` and `sub2` use

```
f90 -ipa -INLINE:list=ON:must=sub1_:sub2_ prog.f90
```

- Recognize parallelization directives in the source code with

```
f90 -mp prog.f90
```

- Apply automatic parallelization with

```
f90 -pfalist prog.f90 -lmp
```

  and look at `prog.list` for a summary of parallelized loops.

Documentation for the SGI is found on `http://techpubs.sgi.com/library`.

2.4 Timers

here are several ways to measure time on Unix systems. Wall clock time could be measured with the `/bin/time` command. Unfortunately it also counts time spent loading the executable and that is sometimes not what you want. As an alternative, you may use the function `TIMEF()`. It returns the elapsed wall clock time in milliseconds (as `DOUBLE PRECISION`).

2.5 Some useful performance analysis tools

You can use the following tools to measure the speed of your program and get statistics on the use of the computer's resources. Detailed examples of their use can be found in the `man`-pages and in Appendix A.

- `perfex -a prog` for overall CPU performance
- `perfex -e event0 -e event1 prog` for specific hardware counters
- `ssrun` for sampling on usage of system resources

2.6 Initial preparations

To ensure that the exercises are run on only one processor you should set the number of threads to 1 with the environment variable `OMP_NUM_THREADS` using the command

`export OMP_NUM_THREADS=1` in ksh or bash

`setenv OMP_NUM_THREADS 1` in csh or tcsh

When running in parallel you can change the variable, i.e. `export OMP_NUM_THREADS=4` will let you run on four processors.

3 Exercise 1 - Getting started with the SGI Onyx2

To get used to the SGI Onyx2 you start by compiling and running three sample programs, demonstrating some of the different factors affecting performance of a program on a RISC architecture computer. In `precision.f90` the impact of `DOUBLE PRECISION` is demonstrated. Cache conflicts' effect on performance is illustrated in `memory.f90` and the benefits of inlining small routines to reduce subroutine call overhead is demonstrated in `inline.f90`. Then, you turn to parallelization. In `parallel.f90` is shown how automatic parallelization of a matrix-matrix multiplication can be achieved and in `scale.f90` you repeat one example from the lectures. Finally, you will be introduced to the software repository `NETLIB` and in `linpack-lapack.f90` you will use subroutines from the `BLAS`, `LINPACK` and `LAPACK` libraries. The source for the exercises can be found in `/misc/pdc/sp2/HPCsummer2001/RISCLAB` and are printed in Appendix B.

3.1 PRECISION exercise

Compile and run the `precision.f90` exercise. What can be said about single/double precision floating point arithmetic performance on a RISC processor? Why is the difference so small? Under which circumstances is single precision twice as fast as double precision?

3.2 MEMORY exercise

Compile and run the `memory.f90` exercise. Explain the behavior. Use `perfex` to find out what happens in the caches. Why is the code so slow? Increase optimization to `-O3`. What happens?

3.3 INLINE exercise

Compile and run the `inline.f90` exercise and look at the output from the program. Explain what you see after studying the source.

3.4 PARALLEL exercise

Compile and run the `parallel.f90` exercise. Compare with the Fortran 90 built-in function `MATMUL()`. Use `-O1` optimization. Why is the "manual" matrix-matrix multiplication slower? Change the code and implement the "inner-product" formulation from the lecture. Any improvements? Increase optimization level to `-O3`. Add parallelization directives (`!$OMP PARALLEL DO`). Any speedup? Does `MATMUL` also run in parallel? Extra, if you have time. Use the BLAS3 routine `DGEMM`. How is the performance? Does it run in parallel?

3.5 Scaling by rows

Scaling of matrices was discussed in the lectures. In `scale.f90` we compare scaling by row and scaling per column. Compile and run the example. Why is one of the loop orders slower? Increase optimization. The compiler changes the loop order. Parallelize the subroutine, either automatically with the `-pfa` compiler option, or, by manually adding appropriate OpenMP `!$OMP PARALLEL DO` directive to the source.

3.6 LINPACK vs. LAPACK

A commonly used software package for linear algebra is *LINPACK*. In 1992 *LAPACK* became available and offers re-designed, more efficient implementations of the linear algebra routines based on *BLAS3*. Optimized versions of BLAS, LINPACK and LAPACK are often supplied by the machine vendors and these libriaries should be used whenever it is possible. On the SGI Onyx2, you link your program with `-lcomplig.sgimath` to access the libraries. There is also a parallel (multi-threaded) version of this library, linked with `-lcomplib.sgimath_mp -lmp`

- Find the LAPACK routine corresponding to the LINPACK routine `DGEFA` factorizing a general, full matrix. LAPACK documentation is found at *Netlib*, a collection of mathematical software, papers, and databases. Use the Netscape web-browser and look at `http://www.netlib.org/lapack`. Note: you do not need to copy the source - optimized versions are available by linking with `-lcomplib.sgimath`.
- Compare the performance of the two routines on matrices of order 100, 500, and 1000. You may use the program `linpack-lapack.f90`.
- Link with the parallel versions of the libraries, i.e., link with `-lcomplib.sgimath_mp -lmp`. Try with 1, 2, and 3 processors. Does it scale?
- If you have never used Netlib before: Find and fetch the source of the LINPACK routine `DGEFA`.

4 Exercise 2 - Optimizing a "real" application

The sample code is called `FDTDA`, a 3D Finite difference Time Domain (FDTD) electromagnetic analysis code from Pennsylvania State University, using a scattered field formulation of Maxwell's equations. `FDTDA` was written by Kunz and Luebbers [1] and the source code is available in the source directory `/misc/pdc/sp2/HPCsummer2001/RISCLAB` or diretcly from the authors' server on `ftp://ftp.emclab.umr.edu/pub/aces/psufdtd`. A brief description of the FDTD method is given in Appendix C.

Your task is to make the necessary changes to the `FDTDA` code and apply compiler directives to make the code run as fast as possible on the SGI Onyx2 (but still produce correct results!), and possible run it in parallel using automatic parallelization (or manually insert parallelization directives in the code).

Below are some points and questions that could be useful in your optimization process.

- Compile and run the code without optimization to get an initial benchmark of the code. Save the output files to be used to test for correctness.
- Decide upon and implement means to measure time and performance of the application.
- Profile the code. Which subroutines account for most of the CPU time?
- Concentrate on the critical routines.

Q How is memory accessed in the critical loops?

Q Is the code CPU or IO bound?

- `f90 -O3 -pfalist` will produce a listing to a `.list` file with some information about automatic parallelization.
- Try to run with automatic parallelization. Do the critical subroutines parallelize? If not, then change the lines hindering parallelization. Insert directives if necessary.
- Document the speedup of your parallel code. Try with 1, 2, and 3 processors.
- Increase the size of the computational domain, i.e. change the parameters `NX, NY, NZ` in the header file `commona.inc` to 1024, one at a time. Does performance improve?
- To get a shorter benchmark you can reduce the number of timesteps by changing the parameter `NSTOP` in `commona.inc`.

A Useful resources

MIPS R10000 - Superscalar CPU Features

The MIPS R10000, designed to solve many of the performance bottlenecks common to earlier microprocessors, is the CPU used in Origin2000 and Onyx2 systems. KTH's Onyx2 system has CPUs running at clock rate 195 MHz.

The R10000 is a four-way superscalar RISC CPU. "Four-way" means that it can fetch and decode four instructions per clock cycle. "Superscalar" means that it has enough independent, pipelined execution units that it can complete more than one instruction per clock cycle. The R10000 contains:

- A nonblocking load-store unit that manages memory access.
- Two 64-bit integer Arithmetic/Logic Units (ALUs) for address computation and for arithmetic and logical operations on integers.
- A pipelined floating point adder for 32- and 64-bit operands.
- A pipelined floating point multiplier for 32- and 64-bit operands.

The two integer ALUs are not identical. Although both perform add, subtract, and logical operations, one can handle shifts and conditional branch and conditional move instructions, while the other can execute integer multiplies and divides. Similarly, instructions are partitioned between the floating point units. The floating-point adder is responsible for add, subtract, absolute value, negate, round, truncate, ceiling, floor, conversions, and compare operations. The floating-point multiplier carries out multiplication, division, reciprocal, square root, reciprocal square root, and conditional move instructions.

The two floating-point units can be chained together to perform multiply-then-add and multiply-then-subtract operations, which are single instruction codes. These combined operations, often referred to by their operation codes of madd and msub, are designed to speed the execution of code that evaluates polynomials.

From Origin2000 and Onyx2 Performance Tuning and Optimization Guide Document Number 007-3430-002

MIPS R10000 - Cache Architecture

PERFEX - R10000 Counter Event Types

The MIPS R10000 CPU contains two 32-bit hardware counters, each of which can be assigned to count any one of 16 events. The counters can be used in two ways. First, they can be used to tabulate the frequency of events in a particular program, for example, counting all instructions, or counting floating-point instructions.

Table B-1 summarizes the types of events that can be counted by the R10000 CPU in either Counter 0 or Counter 1. The table is ordered by the event type number as used in the R10000 special register that controls event counting. The same event numbers are used by the *perfex* and *ssrun* commands. A detailed discussion of the events follows the table.

Table B-1 : R10000 Countable Events

Event Number	Counter 0 Event	Event Number	Counter 1 Event
0	Cycles	16	Cycles
1	Instructions issued to functional units	17	Instructions graduated
2	Memory data access (load, prefetch, sync, cacheop) issued	18	Memory data loads graduated
3	Memory stores issued	19	Memory data stores graduated
4	Store-conditionals issued	20	Store-conditionals graduated
5	Store-conditionals failed	21	Floating-point instructions graduated
6	Branches decoded	22	Quadwords written back from L1 cache
7	Quadwords written back from L2 cache	23	TLB refill exceptions
8	Correctable ECC errors on L2 cache	24	Branches mispredicted
9	L1 cache misses (instruction)	25	L1 cache misses (data)
10	L2 cache misses (instruction)	26	L2 cache misses (data)
11	L2 cache way mispredicted (instruction)	27	L2 cache way mispredicted (data)
12	External intervention requests	28	External intervention request hits in L2 cache
13	External invalidate requests	29	External invalidate request hits in L2 cache
14	Instructions done (in chip rev 2.x, virtual coherence)	30	Stores, or prefetches with store hint, to CleanExclusive L2 cache blocks
15	Instructions graduated	31	Stores, or prefetches with store hint, to Shared L2 cache blocks

From Origin2000 and Onyx2 Performance Tuning and Optimization Guide Document Number 007-3430-002

SSRUN - Taking Sampled Profiles

Similar to *perfex*, the *ssrun* command executes the subject program and collects information about the run. However, where *perfex* uses the R10000 event counters to collect data, *ssrun* interrupts the program at regular intervals. This is a statistical sampling method. The time base is the independent variable and the program state is the dependent variable. The output describes the program's behavior as a function of the time base.

Understanding Sample Time Bases

The quality of sampling depends on the time base that sets the sampling interval. The more frequent the interruptions, the better the data collected, and the greater the effect on the run time of the program. The available time bases are listed in Table 4-2.

Table 4-2 : SpeedShop Sampling Time Bases

ssrun Option	Time Base	Effect and Use
-usertime	30ms timer	Coarsest resolution; experiment runs quickly and output file is small; some bugs noted in speedshop(1).
-pcsamp[x] -fpcsamp[x]	10 ms timer 1 ms timer	Moderate resolution; functions that cause cache misses or page faults are emphasized. Suffix x for 32-bit counts.
-gi_hwc -fgi_hwc	32771 insts 6553 insts	Fine-grain resolution based on graduated instructions. Emphasizes functions that burn a lot of instructions.
-cy_hwc -fcy_hwc	16411 clocks 3779 clocks	Fine-grain resolution based on elapsed cycles. Emphasizes functions with cache misses and mispredicted branches.
-ic_hwc -fic_hwc	2053 icache miss 419 icache miss	Granularity depends on program behavior. Emphasizes code that doesn't fit in L1 cache.
-isc_hwc -fisc_hwc	131 scache miss 29 scache miss	Granularity depends on program behavior. Emphasizes code that doesn't fit in L2 cache.
-dc_hwc -fdc_hwc	2053 dcache miss 419 dcache miss	Granularity depends on program behavior. Emphasizes code that causes L1 cache data misses.
-dsc_hwc -fdsc_hwc	131 scache miss 29 scache miss	Granularity depends on program behavior. Emphasizes code that causes L2 cache data misses.
-tlb_hwc -ftlb_hwc	257 TLB miss 53 TLB miss	Granularity depends on program behavior. Emphasizes code that causes page faults.
-gfp_hwc -fgfp_hwc	32771 fp insts 6553 fp insts	Granularity depends on program behavior. Emphasizes code that performs heavy FP calculation.
-prof_hwc	user-set	Hardware counter and overflow value from environment variables.

In general, each time base discovers the program PC most often in the code that consumes the most units of that time base.

Origin2000/Origin200/Onyx2 Quick Reference
Single Processor Tuning

Based on information in

Origin2000 Performance Tuning and Optimizations

http://www.sgi.com/techpubs/lib/makepage.cgi?007-3430-002

Step One: Get the Right Answer

Select options that emphasize tracing and porting over performance. Options are described in *f77*(1) and *cc*(1).

Flag	*Usage*
-n32	Best choice for IRIX and generic sources.
-64	Needed instead of -n32 when memory usage exceeds 2 GB
-trapuv	Help to find code that uses uninitialized variables (see *f77*(1)).
-check_bounds	Check array references for legality.
-g	Preserve symbols for debugging.
-static	Allocate data in heap initialized to zero when foreign code expects this.
-r8 -i8	Default real and integer to 64-bit size when porting from Cray only. Caution: be sure to use explicit declaration of external library functions e.g. INTEGER*4 TIME

Choose a debugger and learn its command set.

Debugger	*Usage*
dbx(1)	Standard UNIX debugger, elaborate command-line interface
cvd(1)	Graphical debug/test environment in WorkShop ProDev package

Do not proceed until you have:

- A working makefile (see *make*(1), and, for parallel makefiles, *pmake*(1) and *smake*(1)).
- Basic sanity-test cases that yield the expected answers.

Step Two: Use Existing Tuned Code

Hardware-tuned math functions are in:

Collection	*Man Page*	*Available Functions*
fastmath	*libfastm*(3M)	Fastest versions of sin, cos, tan, exp, log, pow
complib	*complib*(3F)	FFTs, convolutions, BLAS, LINPACK,EISPACK,LAPACK, sparse solvers, ...
vector intrinsics	*f77*(1), *cc*(1)	vsin, vcos, vtan, vexp, vlog, vsqrt; enable automatic vectorization with -O3, -Ofast, or -LNO:vintr=on

Use *-lcomplib.sgimath -lfastm* in link step in makefile.

Step Three: Find Out Where to Tune

Use *timex*(1) for overview; *perfex*(1) to see code behavior; drill down with *ssru*

Command	*Purpose*
timex *prog args*	Reports real, user, system time. High system time may b I/O, FPEs (see *sigfpe*(3C)), or system calls (see *par*(1)).
perfex -a [-y] *prog args*	Get overview of program behavior; spot major issues.
perfex -e *nn prog args*	Reports R10K counter *nn*, e.g. 26 for cache misses (list co perfex -h). Verify hypotheses, check for changes from pe
ssrun -pcsamp *prog args*	Sample on realtime intervals; show functions using mos
ssrun -cy_hwc *prog args*	Sample on cycles; show functions using most CPU cycle
ssrun -fp_hwc *prog args*	Sample on FLOPS; show functions doing the most arith
ssrun -dsc_hwc *prog args*	Sample on secondary data cache misses
ssrun -prof_hwc *prog args*	Sample on user-specified counter. Set counter with setenv _SPEEDSHOP_HWC_COUNTER_NUMBER <c setenv _SPEEDSHOP_HWC_COUNTER_OVERFLOW
ssrun -fpe *prog args*	Trace floating point exceptions
ssrun -ideal *prog args*	Profile true counts of basic block use; show most-used f

perfex output goes to stderr. ssrun output goes to file *prog.exptype.nnn*, displa *prof*(1). See *speedshop*(1) for more experiments; see *ssrun*(3) for "caliper" API.

Step Four: Find the Optimum Compilation Flags

Don't rely on default options. Turn off debugging flags, try these optimizatio

Option	*Effect and Purpose*
-Ofast=ip27	Macro for compiler group's top picks for Origin platform s
-O3	Enable maximum optimizations (includes LNO).
-mips4	For R10K, R8K, R5K only; use -mips3 to permit use on R4K
-OPT:IEEE_arithmetic=3	See *f77*(1) or *cc*(1) for discussion. Check that answers are st after applying these.
-OPT:roundoff=3	
-IPA=on	Enable inter-procedural analysis. Changes relative times of compile vs. link; see *ipa*(5).
-OPT:alias=<name>	Disambiguate pointer references. Use <name>=disjoint or whenever possible. See *cc*(1).

Step Five: Tune Cache Performance

- Use Loop Nest Optimizations (via -O3 or -Ofast) to enable loop fusion, inte cache blocking array padding and prefetching
- Use stride-1 accesses whenever possible
- Group together data used at the same time
- Use LNO directives to fine-tune its actions (see *f77*(1) and *cc*(1))
- Use larger page sizes to reduce TLB misses

Copyright 1997 Silicon Graphics, Inc. All rights reserved.

SiliconGraphics

Origin2000/Origin200/Onyx2 Quick Reference
Multiprocessor Tuning

Based on information in

Origin2000 Performance Tuning and Optimizations

http://www.sgi.com/techpubs/lib/makepage.cgi?007-3430-002

Step One: Tune Single Processor Performance

See page 1.

Step Two: Parallelize Code

Choose parallization methodology:

- Automatic parallelization: -pfa or -pca (see *MIPSpro Power Fortran 77 Programmer's Guide, IRIS POWER C User's Guide*)
- MP Library directives: -mp (see *MIPSpro Fortran 77 Programmer's Guide* , *C Language Reference Manual*)
- Other libraries: MPI, PVM, pthreads (see *Topics in IRIX Programming*)

Profile code to monitor degree of parallelization:

- Vary number of threads (e.g., for MP library, use setenv MP_SET_NUMTHREADS)
- Measure wall clock time to determine speedup (e.g., use *timex*(1)).
- perfex -a -mp prints all counts for each thread
- SpeedShop generates an output file for each thread

CPU activity may be displayed with *gr_osview*(1) and *top*(1), memory usage with *gmemusage*(1), hardware configuration with *hinv*(1) and *topology*(1).

Step Three: Identify Bottlenecks

- Is load balance OK?
  - e.g., to check balance of floating point operations
  - perfex -a -mp *prog args* | & grep "floating point"
- Are there a lot of secondary cache misses?
  - perfex -a -y *prog args*

If so, false sharing or data placement may be a problem.

- Is there false sharing?
  - Check perfex output to determine if interventions and/or invalidations are a large fraction of secondary cache misses.

Step Four: Fix False Sharing

If false sharing is a problem, use SpeedShop to monitor stores to shared cache lines:

```
setenv _SPEEDSHOP_HWC_COUNTER_NUMBER 31
ssrun -prof_hwc prog args
```

Revise data structures or algorithms to remedy the problem.

Step Five: Tune For Data Placement

For libmp programs:

For well-parallelized libmp programs which generate a lot of secondary ca
but do not scale well, determine sensitivity to data placement. Try as few s
needed to achieve satisfactory performance:

1) Try round-robin page allocation (setenv _DSM_ROUND_ROBIN o
2) Try page migration (setenv _DSM_MIGRATION on) with round-r

If these techniques don't solve scaling problems, the program needs to be m
make sure the data are properly distributed.

3) Make sure data initializations are parallelized. This relies on "first
data are stored in memory of processor which first accesses a pag
4) Ensure proper data placement via C$DISTRIBUTE and C$PAGE_P
5) For fine-grain data distributions (chunks smaller than a page), con
C$DISTRIBUTE_RESHAPE directive.

The *MIPSpro Fortran 77 Programmer's Guide* and *C Language Reference Manu*
the data distribution directives and the following environment variables:

Variable	*Use and Possible Values*	
_DSM_VERBOSE	Set (any value) to get runtime report of memory and thread placement.	no
_DSM_ROUND_ROBIN	Cause pages to be allocated cyclically all memories used by the program.	no
_DSM_MIGRATION	Set ON to enable migration of explicitly placed data, ALL_ON to enable migration of any data,	OI
_DSM_PPM	Processes per memory (node), set to 1 to use only one CPU per memory (but reduce numthreads also).	2
_DSM_FOP	Enable inter-thread barrier synchronization using fetch+op instructions	no
_DSM_MUSTRUN	Set (any value) to lock threads to processors Not recommended in time-sharing environments	no
_DSM_OFF	Set (any value) to disable data distribution	no
PAGESIZE_STACK, _DATA, and _TEXT)	Set virtual page sizes in KB (e.g., 64). May reduce number of TLB faults.	16

For non-libmp programs, use dplace

```
dplace [-place placement_file] [-data_pagesize n-bytes][-stack_pagesize n-
     [-text_pagesize n-bytes] [-migration threshold] [-propagate] [-mustru
     [-v[erbose]] program [program-arguments]
```

MPI 2.0 placement:

```
setenv MPI_NP <n>
mpirun -np $MPI_NP /usr/sbin/dplace [dplace args] ./a.out [args]
```

Scalable placement_file for MPI 2.0

```
memories ($MPI_NP + 1)/2 in topology cube    # set up compact me
threads $MPI_NP + 1                          # number of threads
distribute threads 1:$MPI_NP across memories # ignore lazy thread
```

Copyright 1997 Silicon Graphics, Inc. All rights reserved.

Meaning of prefixes

S - REAL	C - COMPLEX
D - DOUBLE PRECISION	Z - COMPLEX*16 (this may not be supported by all machines)

For the Level 2 BLAS a set of extended-precision routines with the prefixes ES, ED, EC, EZ may also be available.

Level 1 BLAS

In addition to the listed routines there are two further extended-precision dot product routines DQDOTI and DQDOTA.

Level 2 and Level 3 BLAS

Matrix types:

GE - GEneral	GB - General Band	
SY - SYmmetric	SB - Sym. Band	SP - Sum. Packed
HE - HErmitian	HB - Herm. Band	HP - Herm. Packed
TR - TRiangular	TB - Triang. Band	TP - Triang. Packed

Level 2 and Level 3 BLAS Options

Dummy options arguments are declared as CHARACTER*1 and may be passed as character strings.

TRANx	= '**N**o transpose', '**T**ranspose', '**C**onjugate transpose' (X, X^T, X^H)
UPLO	= '**U**pper triangular', '**L**ower triangular'
DIAG	= '**N**on-unit triangular', '**U**nit triangular'
SIDE	= '**L**eft', '**R**ight' (A or op(A) on the left, or A or op(A) on the right)

For real matrices, TRANSx = 'T' and TRANSx = 'C' have the same meaning.
For Hermitian matrices, TRANSx = 'T' is not allowed.
For complex symmetric matrices, TRANSx = 'H' is not allowed.

References

C. Lawson, R. Hanson, D. Kincaid, and F. Krogh, "Basic Linear Algebra Subprograms for Fortran Usage," *ACM Trans. on Math. Soft.* 5 (1979) 308-325

J.J. Dongarra, J. DuCroz, S. Hammarling, and R. Hanson, "An Extended Set of Fortran Basic Linear Algebra Subprograms," *ACM Trans. on Math. Soft.* 14,1 (1988) 1-32

J.J. Dongarra, I. Duff, J. DuCroz, and S. Hammarling, "A Set of Level 3 Basic Linear Algebra Subprograms," *ACM Trans. on Math. Soft.* (1989)

Obtaining the Software via `netlib@ornl.gov`

To receive a copy of the single-precision software, type in a mail message:

```
send sblas from blas
send sblas2 from blas
send sblas3 from blas
```

To receive a copy of the double-precision software, type in a mail message:

```
send dblas from blas
send dblas2 from blas
send dblas3 from blas
```

To receive a copy of the complex single-precision software, type in a mail message:

```
send cblas from blas
send cblas2 from blas
send cblas3 from blas
```

To receive a copy of the complex double-precision software, type in a mail message:

```
send zblas from blas
send zblas2 from blas
send zblas3 from blas
```

Send comments and questions to `lapack@cs.utk.edu` .

Basic Linear Algebra Subprograms

A Quick Reference G

University of Tennessee
Oak Ridge National Laborator
Numerical Algorithms Group L

May 11, 1997

Level 1 BLAS

Routine	Arguments (dim, scalar, vector, vector, scalars, 5-element array)	Operation	prefixes
SUBROUTINE xROTG	(A, B, C, S)	Generate plane rotation	S, D
SUBROUTINE xROTMG	(D1, D2, A, B, PARAM)	Generate modified plane rotation	S, D
SUBROUTINE xROT	(N, X, INCX, Y, INCY, C, S)	Apply plane rotation	S, D
SUBROUTINE xROTM	(N, X, INCX, Y, INCY, PARAM)	Apply modified plane rotation	S, D
SUBROUTINE xSWAP	(N, X, INCX, Y, INCY)	$x \leftrightarrow y$	S, D, C, Z
SUBROUTINE xSCAL	(N, ALPHA, X, INCX)	$x \leftarrow \alpha x$	S, D, C, Z, CS, ZD
SUBROUTINE xCOPY	(N, X, INCX, Y, INCY)	$y \leftarrow x$	S, D, C, Z
SUBROUTINE xAXPY	(N, ALPHA, X, INCX, Y, INCY)	$y \leftarrow \alpha x + y$	S, D, C, Z
FUNCTION xDOT	(N, X, INCX, Y, INCY)	$dot \leftarrow x^T y$	S, D, DS
FUNCTION xDOTU	(N, X, INCX, Y, INCY)	$dot \leftarrow x^T y$	C, Z
FUNCTION xDOTC	(N, X, INCX, Y, INCY)	$dot \leftarrow x^H y$	C, Z
FUNCTION xxDOT	(N, X, INCX, Y, INCY)	$dot \leftarrow \alpha + x^T y$	SDS
FUNCTION xNRM2	(N, X, INCX)	$nrm2 \leftarrow \|\|x\|\|_2$	S, D, SC, DZ
FUNCTION xASUM	(N, X, INCX)	$asum \leftarrow \|\|re(x)\|\|_1 + \|\|im(x)\|\|_1$	S, D, SC, DZ
FUNCTION IxAMAX	(N, X, INCX)	$amax \leftarrow 1^{st} k \ni \|re(x_k)\| + \|im(x_k)\| = max(\|re(x_i)\| + \|im(x_i)\|)$	S, D, C, Z

Level 2 BLAS

Routine	Arguments (options, dim, b-width, scalar, matrix, vector, scalar, vector)	Operation	prefixes
xGEMV	(TRANS, M, N, ALPHA, A, LDA, X, INCX, BETA, Y, INCY)	$y \leftarrow \alpha Ax + \beta y, y \leftarrow \alpha A^T x + \beta y, y \leftarrow \alpha A^H x + \beta y, A - m \times n$	S, D, C, Z
xGBMV	(TRANS, M, N, KL, KU, ALPHA, A, LDA, X, INCX, BETA, Y, INCY)	$y \leftarrow \alpha Ax + \beta y, y \leftarrow \alpha A^T x + \beta y, y \leftarrow \alpha A^H x + \beta y, A - m \times n$	S, D, C, Z
xHEMV	(UPLO, N, ALPHA, A, LDA, X, INCX, BETA, Y, INCY)	$y \leftarrow \alpha Ax + \beta y$	C, Z
xHBMV	(UPLO, N, K, ALPHA, A, LDA, X, INCX, BETA, Y, INCY)	$y \leftarrow \alpha Ax + \beta y$	C, Z
xHPMV	(UPLO, N, ALPHA, AP, X, INCX, BETA, Y, INCY)	$y \leftarrow \alpha Ax + \beta y$	C, Z
xSYMV	(UPLO, N, ALPHA, A, LDA, X, INCX, BETA, Y, INCY)	$y \leftarrow \alpha Ax + \beta y$	S, D
xSBMV	(UPLO, N, K, ALPHA, A, LDA, X, INCX, BETA, Y, INCY)	$y \leftarrow \alpha Ax + \beta y$	S, D
xSPMV	(UPLO, N, ALPHA, AP, X, INCX, BETA, Y, INCY)	$y \leftarrow \alpha Ax + \beta y$	S, D
xTRMV	(UPLO, TRANS, DIAG, N, A, LDA, X, INCX)	$x \leftarrow Ax, x \leftarrow A^T x, x \leftarrow A^H x$	S, D, C, Z
xTBMV	(UPLO, TRANS, DIAG, N, K, A, LDA, X, INCX)	$x \leftarrow Ax, x \leftarrow A^T x, x \leftarrow A^H x$	S, D, C, Z
xTPMV	(UPLO, TRANS, DIAG, N, AP, X, INCX)	$x \leftarrow Ax, x \leftarrow A^T x, x \leftarrow A^H x$	S, D, C, Z
xTRSV	(UPLO, TRANS, DIAG, N, A, LDA, X, INCX)	$x \leftarrow A^{-1}x, x \leftarrow A^{-T}x, x \leftarrow A^{-H}x$	S, D, C, Z
xTBSV	(UPLO, TRANS, DIAG, N, K, A, LDA, X, INCX)	$x \leftarrow A^{-1}x, x \leftarrow A^{-T}x, x \leftarrow A^{-H}x$	S, D, C, Z
xTPSV	(UPLO, TRANS, DIAG, N, AP, X, INCX)	$x \leftarrow A^{-1}x, x \leftarrow A^{-T}x, x \leftarrow A^{-H}x$	S, D, C, Z

Routine	Arguments (options, dim, scalar, vector, vector, matrix)	Operation	prefixes
xGER	(M, N, ALPHA, X, INCX, Y, INCY, A, LDA)	$A \leftarrow \alpha xy^T + A, A - m \times n$	S, D
xGERU	(M, N, ALPHA, X, INCX, Y, INCY, A, LDA)	$A \leftarrow \alpha xy^T + A, A - m \times n$	C, Z
xGERC	(M, N, ALPHA, X, INCX, Y, INCY, A, LDA)	$A \leftarrow \alpha xy^H + A, A - m \times n$	C, Z
xHER	(UPLO, N, ALPHA, X, INCX, A, LDA)	$A \leftarrow \alpha xx^H + A$	C, Z
xHPR	(UPLO, N, ALPHA, X, INCX, AP)	$A \leftarrow \alpha xx^H + A$	C, Z
xHER2	(UPLO, N, ALPHA, X, INCX, Y, INCY, A, LDA)	$A \leftarrow \alpha xy^H + y(\alpha x)^H + A$	C, Z
xHPR2	(UPLO, N, ALPHA, X, INCX, Y, INCY, AP)	$A \leftarrow \alpha xy^H + y(\alpha x)^H + A$	C, Z
xSYR	(UPLO, N, ALPHA, X, INCX, A, LDA)	$A \leftarrow \alpha xx^T + A$	S, D
xSPR	(UPLO, N, ALPHA, X, INCX, AP)	$A \leftarrow \alpha xx^T + A$	S, D
xSYR2	(UPLO, N, ALPHA, X, INCX, Y, INCY, A, LDA)	$A \leftarrow \alpha xy^T + \alpha yx^T + A$	S, D
xSPR2	(UPLO, N, ALPHA, X, INCX, Y, INCY, AP)	$A \leftarrow \alpha xy^T + \alpha yx^T + A$	S, D

Level 3 BLAS

Routine	Arguments (options, dim, scalar, matrix, matrix, scalar, matrix)	Operation	prefixes
xGEMM	(TRANSA, TRANSB, M, N, K, ALPHA, A, LDA, B, LDB, BETA, C, LDC)	$C \leftarrow \alpha op(A)op(B) + \beta C, op(X) = X, X^T, X^H, C - m \times n$	S, D, C, Z
xSYMM	(SIDE, UPLO, M, N, ALPHA, A, LDA, B, LDB, BETA, C, LDC)	$C \leftarrow \alpha AB + \beta C, C \leftarrow \alpha BA + \beta C, C - m \times n, A = A^T$	S, D, C, Z
xHEMM	(SIDE, UPLO, M, N, ALPHA, A, LDA, B, LDB, BETA, C, LDC)	$C \leftarrow \alpha AB + \beta C, C \leftarrow \alpha BA + \beta C, C - m \times n, A = A^H$	C, Z
xSYRK	(UPLO, TRANS, N, K, ALPHA, A, LDA, BETA, C, LDC)	$C \leftarrow \alpha AA^T + \beta C, C \leftarrow \alpha A^T A + \beta C, C - n \times n$	S, D, C, Z
xHERK	(UPLO, TRANS, N, K, ALPHA, A, LDA, BETA, C, LDC)	$C \leftarrow \alpha AA^H + \beta C, C \leftarrow \alpha A^H A + \beta C, C - n \times n$	C, Z
xSYR2K	(UPLO, TRANS, N, K, ALPHA, A, LDA, B, LDB, BETA, C, LDC)	$C \leftarrow \alpha AB^T + \bar{\alpha} BA^T + \beta C, C \leftarrow \alpha A^T B + \bar{\alpha} B^T A + \beta C, C - n \times n$	S, D, C, Z
xHER2K	(UPLO, TRANS, N, K, ALPHA, A, LDA, B, LDB, BETA, C, LDC)	$C \leftarrow \alpha AB^H + \bar{\alpha} BA^H + \beta C, C \leftarrow \alpha A^H B + \bar{\alpha} B^H A + \beta C, C - n \times n$	C, Z
xTRMM	(SIDE, UPLO, TRANSA, DIAG, M, N, ALPHA, A, LDA, B, LDB)	$B \leftarrow \alpha op(A)B, B \leftarrow \alpha B op(A), op(A) = A, A^T, A^H, B - m \times n$	S, D, C, Z
xTRSM	(SIDE, UPLO, TRANSA, DIAG, M, N, ALPHA, A, LDA, B, LDB)	$B \leftarrow \alpha op(A^{-1})B, B \leftarrow \alpha B op(A^{-1}), op(A) = A, A^T, A^H, B - m \times n$	S, D, C, Z

B Source listings of exercises

B.1 PRECISION exercise

! precision.f90

```
!----------------------------------------------------------------------
! The program PRECISION illustrates the impact of double precision
! on the performance of a program on the SGI Onyx2
!
! To compile and link this program on the SGI Onyx use the command:
!
! f90 -o precision -O3 precision.f90
!----------------------------------------------------------------------
PROGRAM PRECISION
IMPLICIT NONE
!
  INTEGER, PARAMETER :: MX=1000, NITER=5
  REAL               :: A(MX,MX)
  REAL               :: X,  Y,  DX,  DY
  DOUBLE PRECISION   :: AD(MX,MX)
  DOUBLE PRECISION   :: XD, YD, DXD, DYD
  DOUBLE PRECISION   :: TIMEF, T0, T1, T2
  INTEGER            :: I, J, ITER
!
  WRITE(*,*)'PRECISION - What Double Precision does ...'
  WRITE(*,*)'------------------------------------------'
  DX  = 0.1E0
  DY  = 0.2E0
  DXD = 0.1D0
  DYD = 0.2D0
!
! Single precision
!
  T0=TIMEF()/1000.0
  DO ITER=1,NITER
    DO J=1,MX
      DO I=1,MX
        X=I*DX
        Y=J*DY
        A(I,J) = (SIN(X)-EXP(Y))/(X**2+Y**2)
      END DO
    END DO
    CALL DUMMY(A)
  END DO
  T1=TIMEF()/1000.0
!
! Double precision
!
  DO ITER=1,NITER
    DO J=1,MX
      DO I=1,MX
        XD=I*DXD
        YD=J*DYD
        AD(I,J) = (SIN(XD)-EXP(YD))/(XD**2+YD**2)
      END DO
    END DO
    CALL DUMMY(AD)
  END DO
  T2=TIMEF()/1000.0
!
  WRITE(*,*) 'Matrix dimension          =', MX
  WRITE(*,*) 'Time with single precision=', T1-T0
  WRITE(*,*) 'Time with double precision=', T2-T1
!
END PROGRAM PRECISION
!-----------------------------------------------------------------------
! DUMMY - prevents compiler from removing all code while optimizing
!-----------------------------------------------------------------------
SUBROUTINE DUMMY( A )
END SUBROUTINE DUMMY ! EOF
```

B.2 MEMORY exercise

! memory.f90

```
!----------------------------------------------------------------------
! The program MEMORY demonstrates the effect of different memory
! access patterns
!
! To compile and link this program on the SGI Onyx2 use the command:
!
! f90 -o memory -O2 memory.f90
!----------------------------------------------------------------------
PROGRAM MEMORY
IMPLICIT NONE
!
  INTEGER, PARAMETER :: NITER1=20, NITER2=20, NITER3=20, MX=1024, IPAD=1
  REAL               :: A(MX,MX), B(MX+IPAD,MX)
  REAL, PARAMETER    :: MEGA=1024*1024
  DOUBLE PRECISION   :: TIMEF, T0, T1, T2, T3
  INTEGER            :: I, J, ITER
!
  WRITE(*,*)'MEMORY - What memory access patterns means ...'
  WRITE(*,*)'----------------------------------------------'
  CALL DUMMY(A)
  CALL DUMMY(B)
!
! This could cause a lot of cache misses due to bad stride
!
  T0=TIMEF()/1000.0
  DO ITER=1,NITER1
    DO I=1,MX
      DO J=1,MX
        A(I,J) = A(I,J) * 2. + 12.
      END DO
    END DO
  END DO
  CALL DUMMY(A)
!
! Reversing the loops give stride 1
!
  T1=TIMEF()/1000.0
  DO ITER=1,NITER2
    DO J=1,MX
      DO I=1,MX
        A(I,J) = A(I,J) * 2. + 12.
      END DO
    END DO
  END DO
  CALL DUMMY(A)
!
! Padding the first dimension could give better stride
!
  T2=TIMEF()/1000.0
  DO ITER=1,NITER3
    DO I=1,MX
      DO J=1,MX
        B(I,J) = B(I,J) * 2. + 12.
      END DO
    END DO
  END DO
  CALL DUMMY(B)
  T3=TIMEF()/1000.0
!
  WRITE(*,*) 'Matrix dimension                    =', &
       MX
  WRITE(*,*) 'MFlops with stride 1024 memory access=', &
       2*MX*MX*NITER1/MEGA/(T1-T0)
  WRITE(*,*) 'MFlops with stride 1    memory access=', &
       2*MX*MX*NITER2/MEGA/(T2-T1)
  WRITE(*,*) 'MFlops with stride 1025 memory access=', &
       2*MX*MX*NITER3/MEGA/(T3-T2)
  STOP
END PROGRAM MEMORY
!----------------------------------------------------------------------
! DUMMY - prevents compiler from removing all code while optimizing
!----------------------------------------------------------------------
SUBROUTINE DUMMY( A )
END SUBROUTINE DUMMY ! EOF
```

B.3 INLINE exercise

! inline.f90

```
!----------------------------------------------------------------------
! The program INLINE demonstrates what inlining can do to the
! performance of your program.
!
! To compile and link this program on the SGI Onyx2 use the command:
!
! f90 -o inline -O3 -INLINE:never=add_:must=addi_:list=ON inline.f90
!----------------------------------------------------------------------
PROGRAM INLINE
  IMPLICIT NONE
  INTEGER, PARAMETER :: MX=100000, NITER1=10, NITER2=50
  REAL, PARAMETER    :: MEGA=1024.0*1024.0
  REAL               :: A(MX), B(MX)
  DOUBLE PRECISION   :: TIMEF, T0, T1, T2
  INTEGER            :: I, ITER
!
  WRITE(*,*) 'INLINE - What inlining can do ...'
  WRITE(*,*) '---------------------------------'
!
! No inlining
!
  T0 = TIMEF()/1000.0
  DO ITER=1,NITER1
    DO I=1,MX
      CALL ADD( A(I),B(I),A(I) )
    END DO
  END DO
!
! Inlining
!
  T1 = TIMEF()/1000.0
  DO ITER=1,NITER2
    DO I=1,MX
      CALL ADDI( A(I),B(I),A(I) )
    END DO
  END DO
  T2 = TIMEF()/1000.0
  CALL DUMMY(A)
!
  WRITE(*,*) 'Vector length        =',MX
  WRITE(*,*) 'MFlops with no inline=',MX*NITER1/MEGA/(T1-T0)
  WRITE(*,*) 'MFlops with    inline=',MX*NITER2/MEGA/(T2-T1)
  STOP
END PROGRAM INLINE
!-----------------------------------------------------------------------
! ADD - add two reals
!-----------------------------------------------------------------------
SUBROUTINE ADD( A,B,C )
IMPLICIT NONE
!
  REAL, INTENT(IN)  :: A, B
  REAL, INTENT(OUT) :: C
!
  C = A+B
!
END SUBROUTINE ADD
!-----------------------------------------------------------------------
! ADDI - add two reals (will be inlined)
!-----------------------------------------------------------------------
SUBROUTINE ADDI( A,B,C )
IMPLICIT NONE
!
  REAL, INTENT(IN)  :: A, B
  REAL, INTENT(OUT) :: C
!
  C = A+B
!
  END SUBROUTINE ADDI
!-----------------------------------------------------------------------
! DUMMY - prevents compiler from removing all code while optimizing
!-----------------------------------------------------------------------
SUBROUTINE DUMMY( A )
END SUBROUTINE DUMMY ! EOF
```

B.4 PARALLEL exercise

! parallel.f90

```
!----------------------------------------------------------------------
! The program PARALLEL demonstrates the speed up of parallelizable
! loops using autotasking.
!
! To compile and link this program on the SGI Onyx2 use the command:
!
! f90 -o parallel          -O1 parallel.f90 -lcomplib.sgimath
! f90 -o parallel -mp      -O3 parallel.f90 -lcomplib.sgimath_mp -lmp
! f90 -o parallel -pfalist -O3 parallel.f90 -lcomplib.sgimath_mp -lmp
!
!----------------------------------------------------------------------
PROGRAM PARALLEL
IMPLICIT NONE
!
  INTEGER, PARAMETER :: MX=200, MEGA=1024*1024, NITER=10
  REAL               :: A(MX,MX),B(MX,MX),C(MX,MX)
  REAL               :: SECOND
  DOUBLE PRECISION   :: TIMEF, CPUTIME1, CPUTIME2
  DOUBLE PRECISION   :: REALTIME, REALTIME1, REALTIME2
  INTEGER            :: ITER
!
  WRITE(*,*) 'PARALLEL - What multitasking does ...'
  WRITE(*,*) '-------------------------------------'
  CALL RANDOM_NUMBER(B)
  CALL RANDOM_NUMBER(C)
  REALTIME1 = TIMEF()
  CPUTIME1  = SECOND()
  DO ITER=1,NITER
    CALL MATMULT(A,B,C,MX)
!    CALL MATMULT2(A,B,C,MX)
!  Fortran 90
!        A=MATMUL(B,C)
!  BLAS
!        CALL SGEMM('N','N',MX,MX,MX,1.0,B,MX,C,MX,0.0,A,MX)
  END DO
  CALL DUMMY(A)
  CPUTIME2=SECOND()
  REALTIME2=TIMEF()
  REALTIME=(REALTIME2-REALTIME1)/1000.0
  WRITE(*,*) 'MATMULT: N = ',MX
  WRITE(*,*) '       CPU = ',CPUTIME2-CPUTIME1
  WRITE(*,*) '   Elapsed = ',REALTIME
  WRITE(*,*) '    MFlops = ',2*MX*MX*MX/MEGA/REALTIME*NITER
  STOP
!
END PROGRAM PARALLEL
!---------------------------------------------------------------
SUBROUTINE MATMULT(A,B,C,N)
IMPLICIT NONE
!
  INTEGER, INTENT(IN)  :: N
  REAL,    INTENT(OUT) :: A(N,N)
  REAL,    INTENT(IN)  :: B(N,N), C(N,N)
!
  INTEGER              :: I, J, K
!
! Multiply A=B*C using index order K/J/I
!
!$OMP PARALLEL DO SHARED(N, A) PRIVATE(K,I)
  DO K=1,N
    DO I=1,N
      A(I,K)=0.0
    END DO
  END DO
!$OMP END PARALLEL DO
!
!$OMP PARALLEL DO SHARED(N, A, B, C) PRIVATE(K,J,I)
  DO K=1,N
    DO J=1,N
      DO I=1,N
        A(I,K)=A(I,K)+B(I,J)*C(J,K)
      END DO
    END DO
  END DO
!$OMP END PARALLEL DO
!
END SUBROUTINE MATMULT
```

```
!-------------------------------------------------------------
SUBROUTINE MATMULT2(A,B,C,N)
IMPLICIT NONE
!
  INTEGER, INTENT(IN)  :: N
  REAL,    INTENT(OUT) :: A(N,N)
  REAL,    INTENT(IN)  :: B(N,N),C(N,N)
!
  INTEGER              :: I, J, K
!
! Multiply A=B*C using index order I/J/K (inner product)
!
!$OMP PARALLEL DO SHARED(N, A, B, C) PRIVATE(I,J,K)
  DO I=1,N
    DO J=1,N
      A(I,J)=0.0
      DO K=1,N
        A(I,J)=A(I,J)+B(I,K)*C(K,J)
      END DO
    END DO
  END DO
!$OMP END PARALLEL DO
!
END SUBROUTINE MATMULT2
!---------------------------------------------------------------------
! DUMMY - prevents compiler from removing all code while optimizing
!---------------------------------------------------------------------
SUBROUTINE DUMMY( A )
END SUBROUTINE DUMMY ! EOF !
```

B.5 LINPACK-LAPACK exercise

! linpack-lapack.f90

```
!--------------------------------------------------------------------
! The program LINPACK_LAPACK demonstrates the difference in performance
! between LINPACK and LAPACK
!
! To compile and link this program on the SGI Onyx2 use the command:
!
! f90 -o linpack-lapack -O3 linpack-lapack.f90 -lcomplib.sgimath
! f90 -o linpack-lapack -O3 linpack-lapack.f90 -lcomplib.sgimath_mp -lmp
!--------------------------------------------------------------------
PROGRAM LINPACK_LAPACK
IMPLICIT NONE
!
  INTEGER, PARAMETER :: N=500
  INTEGER            :: IPVT(N)
  DOUBLE PRECISION   :: A(N,N)
  DOUBLE PRECISION   :: TIMEF, T0, T1
  INTEGER            :: NITER, ITER, INFO
!
  WRITE(*,*) 'LINPACK-LAPACK - Compare LINPACK and LAPACK'
  WRITE(*,*) '-----------------------------------'
  WRITE(*,*) 'Factorization of matrix of order', N
  CALL RANDOM_NUMBER(A)
  NITER=10
  T0=TIMEF()
  DO ITER=1,NITER
    CALL DGEFA(A,N,N,IPVT,INFO)
  END DO
  T1=TIMEF()
  WRITE(*,*) 'Time using LINPACK (s)', (T1-T0)/1000.0
!
  NITER=10
  T0=TIMEF()
  DO ITER=1,NITER
    WRITE(*,*) 'Find and put in LAPACK_EQUIVALENT_TO_DGEFA(A,N,N,IPVT,INFO)'
  END DO
  T1=TIMEF()
  WRITE(*,*) 'Time using LAPACK  (s)', (T1-T0)/1000.0
!
END PROGRAM LINPACK_LAPACK ! EOF !
```

B.6 SCALE exercise

! scale.f90

```
!-----------------------------------------------------------------------
! The program SCALE demonstrates the importance of loop-ordering
!
! To compile and link this program on the SGI Onyx2 use the command:
!
! f90 -o scale -O2 scale.f90
!-----------------------------------------------------------------------
PROGRAM SCALE
IMPLICIT NONE
!
  INTEGER, PARAMETER :: M=1000,N=500
  INTEGER, PARAMETER :: NITER1=20, NITER2=20
  DOUBLE PRECISION   :: D(N),A(M,N)
  DOUBLE PRECISION   :: TIMEF, T0, T1, T2
  INTEGER            :: ITER
!
  WRITE(*,*) 'SCALE - the importance of loop ordering ...'
  WRITE(*,*) '-------------------------------------------'
  CALL RANDOM_NUMBER(A)
  CALL RANDOM_NUMBER(D)
  CALL SCALE1(M,N,A,D)
  T0=TIMEF()
  DO ITER=1,NITER1
    CALL SCALE1(M,N,A,D)
  END DO
  T1=TIMEF()
  WRITE(*,*) 'Standard loop took  ', (T1-T0)/NITER1,' millisec'
  T1=TIMEF()
  DO ITER=1,NITER2
    CALL SCALE2(M,N,A,D)
  END DO
  T2=TIMEF()
  WRITE(*,*) 'Reordered loop took ', (T2-T1)/NITER2,' millisec'
!
END PROGRAM SCALE
!
SUBROUTINE SCALE1(M,N,A,D)
IMPLICIT NONE
!
  INTEGER, INTENT(IN)             :: N, M
  DOUBLE PRECISION, INTENT(INOUT) :: A(M,N)
  DOUBLE PRECISION, INTENT(IN)    :: D(N)
!
  INTEGER                         :: I, J
!
  DO I=1,M
    DO J=1,N
      A(I,J)=A(I,J)*COS(D(J))
    END DO
  END DO
!
END SUBROUTINE SCALE1
!
SUBROUTINE SCALE2(M,N,A,D)
IMPLICIT NONE
!
  INTEGER, INTENT(IN)             :: N, M
  DOUBLE PRECISION, INTENT(INOUT) :: A(M,N)
  DOUBLE PRECISION, INTENT(IN)    :: D(N)
!
  INTEGER                         :: I, J
!
  DO J=1,N
    DO I=1,M
      A(I,J)=A(I,J)*COS(D(J))
    END DO
  END DO
!
END SUBROUTINE SCALE2 ! EOF
```

C About the Finite Difference Time Domain Method

The Finite Difference Time Domain (FDTD) method for electromagnetic computations, introduced by Yee in 1966 is described briefly below. A modern reference is Taflove's book from 1995 [2].

FDTD is based on a discretization of Maxwell's curl equations for linear media,

$$\mu \frac{\partial H}{\partial t} + \nabla \times E = 0 \quad (1)$$

$$\epsilon \frac{\partial E}{\partial t} - \nabla \times H = -J \quad (2)$$

where E is the electric field, H the magnetic field, J the source current density, μ the magnetic permeability and ϵ the electric permittivity. The spatial and time derivatives are approximated with central differences. The form of the resulting difference equations depend on the actual coordinate system where the Cartesian one is the most commonly used. The Cartesian grid is based on a unit cell. Different components of the electric field are given in a staggered fashion, and so are the components of the magnetic field. Also, the magnetic field points are staggered between the electric field points that are located at the center of the rectangular cell faces (or vice versa), see Figure 1. Time is discretized with equidistant steps and the E and H fields are separated in time with one half of the time step. The algorithm uses a leapfrog scheme in time and space to follow the evolution of the electric and magnetic fields in the computational space. The resulting finite difference equations are explicit and result in a fast and, in its basic form, robust algorithm for electromagnetic field computations. To calculate a given component of the H-field, only four neighboring E-field values are needed (and vice versa for the E-fields). This means that in a parallel version of the code, where the computational space is divided into blocks and the electromagnetic fields in each block are computed on different processors, only the fields at the block boundaries have to be communicated to other blocks.

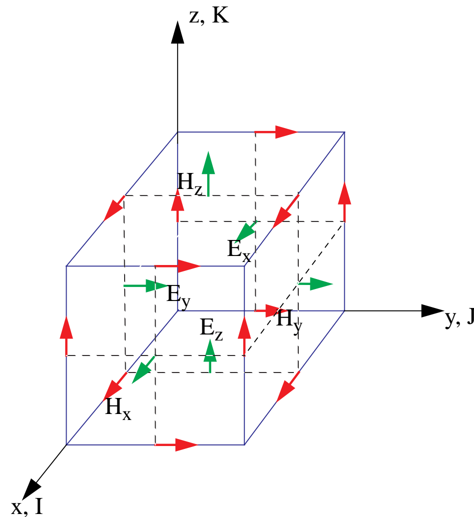

Figure 1: Yee FDTD unit cell

The cell size and the size of the computational space should be chosen with respect to the object and its curvature and the maximum frequency. In practice it is a trade off between those factors, the size of the computer memory and the CPU time. To achieve acceptable accuracy, the cell size should be smaller than the wavelength divided by a factor p, where p depends on the object and the physical quantity to be calculated. The factor p should not be smaller than 10 and might in some cases have to be as large as 30. Most of the problems that are studied are for unbounded domains, since there are no natural boundaries such as in the case of a metallic cavity. However, the size of computer memories are limited so the computational space has to be bounded. This

must be done in a way such that outgoing waves are absorbed by the outer boundary, thereby simulating an infinite space. The boundary can not simply be terminated because the fields at the boundary will require information about the fields outside the boundary. This is achieved by applying boundary conditions referred to as absorbing or radiating boundary conditions. It can been shown that by factorizing the partial differential operator L for a wave equation $LU = 0$ as $L^+L^-U = 0$, where U is a scalar field component, one type of absorbing boundary conditions can be derived. Applying L^+ or L^- to U at the boundary absorbs the wave. This can, however, not be implemented exactly in the finite difference scheme. One first order approximation is decribed by Mur. Essentially L^+ and L^- are expanded in Taylor series where only the first two terms are kept and the derivatives are approximated with finite differences. The basic physical picture is that the fields just outside the boundary are calculated by extrapolation from the fields inside the boundary at earlier times. These fields are then used to calculate the values of the fields at the boundary at later times. A new, so far promising, type of absorbing boundary conditions are the Perfectly Matched Layers (PML) boundary conditions.

The excitation is usually an incident plane wave or a current induced by a voltage or current source in the interior of the object or attached to the exterior. An incident wave is created by sources on a surface enclosing the object. The sources are commonly referred to as Huygens sources and the surface as a Huygens surface. The underlying principle is the equivalence principle. It states that the fields within an enclosed surface, not containing the source, can be exactly reproduced without the source, if the tangential electric and/or magnetic current densities are known on the surface. The method almost completely confines the incident wave to the region inside the Huygens surface while the surface is still transparent to outgoing waves.

The time step is limited by the CFL (Courant-Friedrich-Levy) condition for numerical stability,

$$\Delta t \quad < \quad \frac{1}{c\sqrt{\frac{1}{(\Delta x)^2} + \frac{1}{(\Delta y)^2} + \frac{1}{(\Delta z)^2}}},$$

where c is the wave velocity of the medium and Δx, Δy and Δz are the cell sizes in the x-, y-, and z-direction, respectively.

References

[1] *The Finite Difference Time Domain Method for Electromagnetics*, K. Kunz and R. Luebbers, CRC Press, 1993

[2] *Computational Electromagnetics: The Finite-Difference Time-Domain Method*, A. Taflove, Artech House, Boston, MA, 1995