SX-6 Compare and Contrast

Thomas J. Baring
Arctic Region Supercomputing Center
University of Alaska Fairbanks

ABSTRACT: The Arctic Region Supercomputing Center (ARSC) installed a single 8-CPU Cray SX-6 node, and made it available last August to the broader U.S. HPC community for benchmarking and testing. Our experiences of 6 months suggest that, to those accustomed to traditional Cray PVP systems, the SX-6 architecture and user environments are simultaneously familiar and peculiar. We also note that performance sustained by vectorizable user codes, per SX-6 CPU, has been gratifying while achieving additional speedup from multiple CPUs has proven more elusive.

Introduction

The Arctic Region Supercomputing Center (ARSC), located on the campus of the University of Alaska Fairbanks, supports computational research in science and engineering with emphasis on high latitudes and the Arctic. The center provides high performance computational, visualization, networking and data storage resources for researchers within the Department of Defense and other government agencies and the University of Alaska and other academic institutions.

For a period of two years, to end in June 2004, ARSC and Cray Inc. are jointly managing a Cray SX-6 for their mutual and individual benefit. The system is physically installed in Alaska but Cray personnel located primarily in Mendota Heights, MN provide system administration and operations over a dedicated VPN. ARSC and Cray maintain completely disjoint help desks, account administration procedures and policies, user bases, and front-end workstations (for user authentication and access as well as support of programming environment cross-compilers and GUI tools). ARSC and Cray personnel meet frequently by email and teleconference to discuss issues as they arise and to manage shared elements of the system, such as the structure of the NQS queues, and the operations schedule.

As sole North American distributor of the SX-6, Cray uses the system to benchmark codes, teach classes, and provide early access to prospective clients. ARSC makes the system available for evaluation as a unique service to the wider U.S. HPC community--including individuals who are otherwise non-ARSC users.

This paper attempts to give a vicarious experience of using the SX-6 through descriptions and porting case-studies, using UNICOS and traditional Cray PVP systems as primary points of reference.

SX-6 Architecture Overview

The SX-6 installed at ARSC is a single cabinet, 8 CPU node with 64 GB of symmetric shared memory, and 1.1 TB disk. ARSC-sponsored users have additional access to 420 GB of disk, cross-mounted from ARSC's SGI Octane 2 front-end host.

Each CPU is a single-chip 8-way vector processor. The vector units operate at 500 Mhz, and as multiplication and addition can be chained, the peak performance of a single vector pipe is 1 GFLOPS. Given 8 pipes, the peak theoretical performance per processor is 8 GFLOPS. The scalar units also operate at 500 Mhz and have a theoretical peak of 1 GFLOPS. The 8 vector registers per CPU are 256 elements long and CPU-Memory bandwidth is 32GB/sec/CPU. There is no bit-matrix multiply unit, and the system runs the SUPER-UX operating system, a Unix variant.

SX-6 User Environment

Front-End Host

Cross-compilers and other programming tools, running on an SGI front-end workstation, are well-integrated into the SX-6 user environment. Although "self-compilers" are installed on the SX-6, users are encouraged to compile on the front-end system, and to move their executable to the SX-6 using one of the cross-mounted file systems or the scp command. It's also possible to compile, debug, profile, and run from within the "PSUITE" GUI environment running on the front-end, Similarly, the GUI-based MPI profiling tool, VAMPIR/SX runs on the front-end.

Parallel Programming models:

Parallel programming models are available as follows:

Autotasking	Yes
OpenMP	Yes
MPI	Yes
MPI-2	"almost all"
SHMEM	No
Coarray Fortran	No
UPC	No
HPF	Yes

Table 1: SX-6 Programming Models

Compilers/libraries:

The SX-6 supports the Fortran 90, C, and C++ programming languages. Both 32- and 64-bit IEEE numeric units are supported and programs are compiled in either of two modes: default or wide precision. In the former, Fortran `REAL` and `INTEGER` are all 32-bit, and size of variables declared explicitly with the `KIND` attribute or "*" are honored.

"Wide" precision overrules all explicit size specifications, forcing them and default variables to 64-bit. The user must link the correct version of libraries for which both default and 64-bit version exist. For example, the 64-bit version of libblas.a is libblas_64.a and that of libmpi.a versus libmpiw.a. Use of "wide" precision is a close approximation to traditional UNICOS PVP 64-bit precision.

Fortran Standard Features

In most respects, the set of features provided in the SX-6 f90 compiling system (termed the "overall" compiler) is familiar to Cray users. Here's a partial list of options :

SX-6 f90 option	Cray ftn Comparison
[{-c | -Nc}]	Same
[-Dname[=value]]	Same
[-d [a][C][D][W][w]]	Same (disable)
[-e [a][C][D][W][w]]	Same (enable)
[{-Ep | -EP | -NE}]	CPP control
[{-f0 | -f3 | -f4}]	Fixed/free format
[{-ftrace | -Nftrace}]	Tracing
[-G {global | local}]	taskcommon
[{-g [v] | -Ng}]	Same (debugging)
[-I directory-name]	Same
[-L library-directory-name]	Same
[-l library-name]	Same
[-o object-file-name]	Same
[-p | -Np}]	Profiling
[{-R0 | -R1 | -R2 | -R3 | -R4 | -R5}]	Listing, loopmark, etc.
[-Wa "option-string"]	Same ("as" options)
[-Wc "option -string"]	"cc" options
[-Wf "option -string"]	Same ("f90" options)
[-Wl "option -string"]	Same ("ld" options)
[-Wp "option -string"]	"cpp" options

Table 2: SX-6 f90/Cray ftn options

Optimization/Vectorization/Parallelization

There's divergence between the SX-6 and Cray compilers regarding the number of scalar optimization (termed just "optimization" in the SX-6 world), vectorization, and parallelization options. SX-6 users have greater low-level control over compilation. On Cray systems, here is the general list of 22 features available through "-O":

0, 1, 2, 3
aggress, noaggress
bl, nobl
allfastint, fastint, nofastint
ieeeconform, noieeeconform
inline0, inline1, inline2, inline3
inlinefrom=source
loopalign, noloopalign
msgs, nomsgs
negmsgs, nonegmsgs
modinline, nomodinline
Nointerchange
overindex, nooverindex
pattern, nopattern
recurrence, norecurrence
scalar0, scalar1, scalar2, scalar3
stream0, stream1, stream2, stream3
task0, task1, task2, task3
taskinner, notaskinner
threshold, nothreshold
vector0, vector1, vector2, vector3
vsearch, novsearch
zeroinc, nozeroinc

Table 3: Cray ftn "–O" options:

The highest-level interface to the optimization options through the Cray compilers is "-O[0|1|2|3]", each of which controls a set of lower level options. For instance, on the SV1ex, "-O3" is equivalent to:

```
Recurrence,Scalar=2,Vector=3,VSearch,Task=2
```

SX-6 f90 offers two analogous high-level controls: "-P" for automatic parallelization and "-C" for scalar optimization and automatic vectorization:

```
f90 or sxf90
[-C{debug|ssafe|vsafe|sopt|vopt|hopt}]
[-P{auto | openmp | multi | stack | static}]
```

As in Cray ftn, each high-level option defines a set of what are termed "detailed" options, all of which are accessible through the high-level "-Wf…" option. The difference is simply the number of such options over which the user has control. Presumably, the Cray compilers perform the same types of optimization, but they are largely hidden from the user behind intermediate levels (like "scalar[0-3]"). Here is the complete list of 19 scalar and 39 vectorization/parallelization SX-6 detailed options:

SX-6 f90 detailed options for optimization:

```
[{-ai | -Nai }]
[{-fusion | -Nfusion}]
[{-i[{errchk | noerrchk}] | -Ni}]
[-O [{chg | nochg}]
[{darg | nodarg}]
[{div | nodiv}]
[extendreorder]
[reorderrange=bblock]
[{if | noif }]
[{infomsg | nomsg }]
[{iodo | noiodo}]
[{move | nomovediv | nomove}]
[{overlap | nooverlap}]
[{unroll[={nlevel]| nounroll}]
[{zlpchk | nozlpchk}]
[-prob_generate]
[-prob_use]
[-prob_dir=directory-name ]
[-prob_file=filename]
```

Table 4: SX-6 f90 scalar optimization detailed options

SX-6 f90 detailed options for vectorization and parallelization:

```
[-common {global|local}]
[-moddata {global|local}]
[-ompctl [{condcomp|nocondcomp}] ]
[-pvctl[{altcode={{dep|nodep}|{loopcnt|noloopcnt}}|
noaltcode}]
[{assoc | noassoc}]
[{assume | noassume}]
[{chgpwr}]
[{cncall=routine-name[,routine-name]}]
[{collapse | nocollapse}]
[{compress | nocompress}]
[{divloop |nodivloop}]
[{expand=n | noexpand}]
[{for[=n] | by=n}]
[{fullmsg | infomsg | nomsg}]
[{ifopt | noifopt}]
[{inner | noinner}]
[{listvec | nolistvec}]
[{loopchg |noloopchg}]
[loopcnt=n ]
[{lstval | nolstval}]
[{matmul | matmulblas | nomatmul}]
[{outerstrip | noouterstrip}]
[{outerunroll | noouterunroll}]
[{parcase | noparcase}]
[{parthreshold[=n] | noparthreshold}]
[res={whole | parunit | no}]
[shape=n1[,n2]...]
[{split | nosplit}]
[{vchg | novchg}]
[vecthreshold=n ]
[{verrchk | noverrchk}]
[ vl={fix256 | max512 | fix512}]
[{vlchk | novlchk}]
[{vr64 | vr128 | vr256 | vr512}]
[ vwork={ stack | statick}]
[vworksz=n[M] ] ]
[-reserve=n ]
[-tasklocal {macro|micro}]
[{-v |-Nv}]
```

Table 5: SX-6 f90 detailed vectorization/optimization options

For an analysis of the effects and potential usefulness of the detailed options, see the section on the RIPPLE code, described below. Also, note that optimization of FLAPW (below) relied on access to these options in order to switch the one option off which was preventing use of the overall, "-C vopt" setting.

C/C++

On the SX-6, there remains an apparently demoted, standalone C language compiler. The preferred approach to compiling C code is to use the C++ compiler, giving it the option "c++ … -Xa". The C++ optimization and vectorization/parallelization options are consistent in many respects with the f90 compiler. For instance, the "-C" and "-P" overall options are the same, as is access to detailed options. The set of detailed options is reduced to 9 scalar and 20 vectorization/parallelization.

Run Time Environment

Another contrast appears at run time. While UNICOS seems to prefer commands and compiler options to define features of the run-time environment, SUPER-UX uses a large set of environment variables. Table 6 gives the 25 (of 39 total) variables for which an approximate UNICOS equivalent exists (based on the authors' experience).

SUPER-UX Run-time Parameters	*Functionally similar UNICOS command(S)*
Exception handling	
F_ERRCNT, F_ERRHALT, F_ERRMSG, F_ERROPTn, F_EXPRCW	/etc/cpu
I/O Stats	
F_FILEINF	Procstat, procview
Profiling	
F_FTRACE	f90 -ef, flowtrace
Performance data	
F_PRINFDIFF, F_PROGINF	hpm, ja
I/O control	
F_FMTBUF, F_HSDIR, F_INPUT, F_MEMWAIT, F_NORCW, F_OUTPUT, F_PARTRCW, F_POSITION, F_PROMOTE, F_SETBUF[u], F_SETBUFALIGN[u], F_UFMTADJUST[u], F_UFMTENDIAN, F_UFMTFLOAT1, F_UFMTFLOAT2, F_UFMTIEEE,	assign

Table 6: SX-6 Run Time parameters

Exception handling

The options listed above for controlling run-time handling of exceptions provide an impressive level of detail to the user. For example, F_ERRCNT sets the number of exceptions which will be detected and reported before the program halts. F_ERROPTn controls handling of one

specific error number or range of numbers, and takes the following arguments:

```
    setenv F_ERROPTn n1, n2, alt, err, m, t, a,
cnt
```

The description of this one run-time parameter takes a couple of pages in the SX-6 manual. Sparing you the details (available in ARSC HPC Newsletter issue 264 [1]), here is an example which would instruct the operating system to: not change the behavior for error 274 only, not have a user defined routine handle the error, not trap IO errors, not issue error messages, not trace back, not terminate, and finally, not count the errors in this class:

```
setenv F_ERROPT1 274,274,0,0,2,2,2,2
```

A setting like this could be made for every error number of concern, or none at all.

Performance Analysis

As mentioned above, making the following assignment at run-time causes performance data for the given execution to be dumped:

F_PROGINF=DETAIL

Here's an example of the output from a 4-CPU auto-parallelized run of the FLAPW code, which will be described in more detail below. It's given here to show the range of counters and data provided.

```
    ******  Program Information  ******
 Real Time (sec)       :        719.253574
 User Time (sec)       :       1229.029483
 Sys  Time (sec)       :        120.601631
 Vector Time (sec)     :        219.913104
 Inst. Count           :      92135082895.
 V. Inst. Count        :       9978846506.
 V. Element Count      :    1126941276992.
 FLOP Count            :     636068564467.
 MOPS                  :        983.782350
 MFLOPS                :        517.537271
 MOPS   (concurrent)   :       2773.263985
 MFLOPS (concurrent)   :       1458.927855
 VLEN                  :        112.933020
 V. Op. Ratio (%)      :         93.205160
 Memory Size (MB)      :        144.000000
 Max Concurrent Proc.  :                4.
  Conc. Time(>= 1)(sec):        435.983563
  Conc. Time(>= 2)(sec):        282.813372
  Conc. Time(>= 3)(sec):        281.641796
  Conc. Time(>= 4)(sec):        228.592612
 Event Busy Count      :                0.
 Event Wait (sec)      :          0.000000
 Lock Busy Count       :                0.
 Lock Wait (sec)       :          0.000000
 Barrier Busy Count    :                0.
 Barrier Wait (sec)    :          0.000000
 MIPS                  :         74.965722
 MIPS (concurrent)     :        211.326964
 I-Cache (sec)         :         10.048854
 O-Cache (sec)         :        105.484218
 Bank (sec)            :         10.659311

 Start Time (date)  :  2002/12/10 14:53:10
 End   Time (date)  :  2002/12/10 15:05:09
```

Batch System

The SX-6 batch system is named NQS, and it includes familiar sounding commands from UNICOS, such as "qsub", "qdel", and "qmgr". Under SUPER-UX, there are eight different "qstat" commands: "qstat[a|c|ck|d|f|q|r]". Thus, "qstatc" is roughly equivalent to "qstat –c" on UNICOS. The syntax of qsub scripts is almost equivalent, as are most other aspects of these two varieties of NQS.

Documentation

The philosophy of giving access to details seems to hold true with the online SX-6 manuals. The on-line manuals are exhaustive and well-written.

SX-6 Benchmarking Experiences and Comparisons

ARSC staff members and users have ported several user application codes to the SX-6 as part of our research on this system. In this section, five codes, with performance comparisons as well as porting anecdotes, are presented.

CICE

The Los Alamos Sea Ice Model, CICE version 3.1 was ported to the SX-6 and SV1ex. CICE is engineered to serve as a component of a fully coupled global climate model, but for these tests, was run as a standalone application. It is parallelized using MPI but contains an alternate code path, selected by makefile options, to produce a non-MPI version. For these tests, MPI, single processor, and auto-parallelized versions were used.

An ARSC user provided CICE with 16GB of IEEE formatted binary input data files and restart file. After some effort, attempts to convince the SV1ex version to read the restart file or run in MPI mode were both abandoned. Thus, the only SV1ex data available are single CPU or autotasked, and starting from the 0^{th} iteration as opposed to using the restart file. For performance comparisons to be valid, all SX-6 runs (except that noted in table 7 as having used a restart file) were also made from the 0^{th} iteration.

The following table shows the performance of this code on 1-CPU. All SX-6 data were collected on a dedicated system while all SV1ex data were collected on a lightly to moderately loaded system. One day of ice formation was simulated. The SX-6 run was over 3.5 times faster based on wall-clock time than the SV1ex run.

System	CPU PeakPerf (gflops)	Wall-clock time (sec)	mflops	Percent of CPU PeakPerf	Restart File Used
SV1ex	2.0	10703	158	8%	No
SX6	8.0	2931	670	8%	No
SX6	8.0	1488	1027	13%	Yes

Table 7: CICE performance results, 1-CPU, 1-day simulation runs

As shown in the following graph, the MPI version achieved good but not exemplary parallel speedup on the SX-6. The restart runs had better performance and worse speedup, but this improved with longer runs. This is an advantage to the user, who simulates multiple years, and thus almost invariably uses both restart files and maximum CPU time.

In addition to explicit parallelism, the single CPU version of CICE was auto-parallelized on both platforms, using the following compiler options:

SX-6: `f90 -sx6 -Wf,"-pvctl noverrchk" -P auto -C vopt`

SV1ex: `ftn -Otask3,aggress`

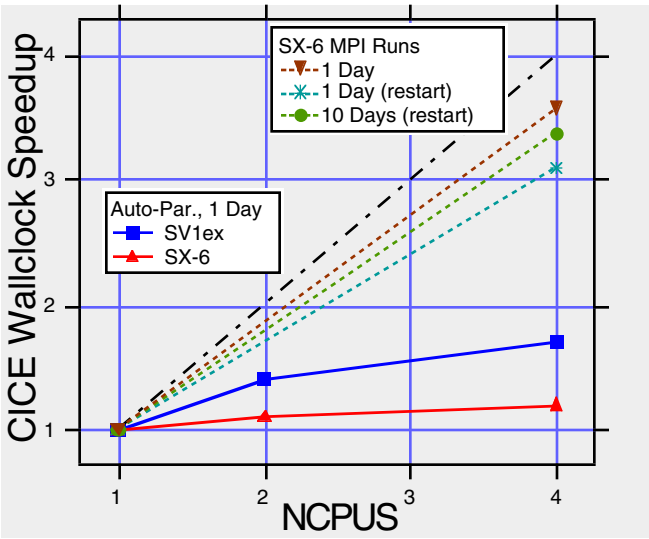

FIGURE 1: CICE speedup

The auto-parallelization results graphed indicate that for this code, SV1ex autotasking was far more successful than SX-6 auto-parallelization at providing speedup. A possible explanation is that a single SX-6 CPU has 4 times the internal capacity of a single SV1ex CPU to exploit parallelism (8 versus 2 vector pipes), and thus the SX-6 compiler may have exhausted most of the code's loop-level parallelism in saturating one CPU, leaving little work for multiple CPUs. In fact, the SV1ex 4-CPU multitasked run takes 6256 seconds of wall-clock time compared with 2931 seconds for the SX-6 1-CPU run, even though the peak theoretical performance of 4 SV1ex CPUs equals that of 1 SX-6 CPU. Single-CPU vectorization on one SX-6 CPU is more successful than the combination of vectorization and autotasking on the SV1ex.

Here are some lessons learned about the SX-6 in porting this code. In the first of three data size problems, the "-ew" flag to force all storage to 64 bits is not necessary for this code. It uses Fortran 90 "kind" attributes extensively, and failed when they were overridden by "-ew." Second, on the SX-6, unlike the Crays, the default unit size for direct access file record lengths is 8-bytes. Thus, opening a file with OPEN (... RECL=16...), for instance, implies that each record is 8*16 (rather than 16) bytes long, and READ statements failed by reading past the ends of files. This was corrected by making this setting at run-time:

```
export F_RECLUNIT=BYTE
```

Third, run-time ""floating-point data overflow" errors occurred on the SX-6 because several functions were declared as default real type, which is 32-bits,

```
real function ice_global_real_minval(nc,work)
```

while the function return values were assigned to 64-bit variables,

```
real (kind=dbl_kind) :: amin
amin = ice_global_real_minval(1,timerw)
```

The correction was straightforward:

```
real (kind=dbl_kind) function    &
ice_global_real_minval(nc,work)
```

ROMS

Kate Hedstrom of ARSC ported the Regional Ocean Model System (ROMS), OpenMP version 1.9, to the SX-6 and compared performance against an IBM Regatta Power4 server. Performance results for 1-CPU runs are given in the following tables for two ocean configurations. The configuration of scientific interest to the user, shown first, has 512x64x30 grid points and full physics including vertical mixing. The user reports that this version does not perform well because computations in the vertical dimension do not vectorize, and this scalar component comes to dominate execution time. (The performance tool used on the Regatta, "hpmcount" reports Mflip/s – million floating point instructions per second – rather than MFLOPS).

System	CPU PeakPerf (GFLOPS)	Wall-clock time (sec)	Mflips or Mflops	Percent of CPU PeakPerf
Regatta	5.2	1444	439	8.4%
SX6	8.0	1439	405	5.0%

Table 8: ROMS performance result, 1-CPU runs, full-physics configuration

The user also ran two idealized problems with specified mixing (Table 9) which do vectorize well. These were run on both 160x160 and a 400x400 grids, and the larger grid was not run on the Regatta.

System	CPU PeakPerf (GFLOPS)	Wall-clock time (sec)	Mflips or Mflops	Percent of CPU PeakPerf	Problem Size
Regatta	5.2	496	625	12%	160x160
SX6	8.0	110	2240	28%	160x160
SX6	8.0	438	3509	44%	400x400

Table 9: ROMS performance result, 1-CPU runs, Test Configuration

Parallelism in this implementation of ROMS is coded explicitly with OpenMP, and as the full physics and 160x160 idealized problem results shown here demonstrate, the code scales reasonably on both systems. These runs were made on lightly loaded, but non-dedicated systems.

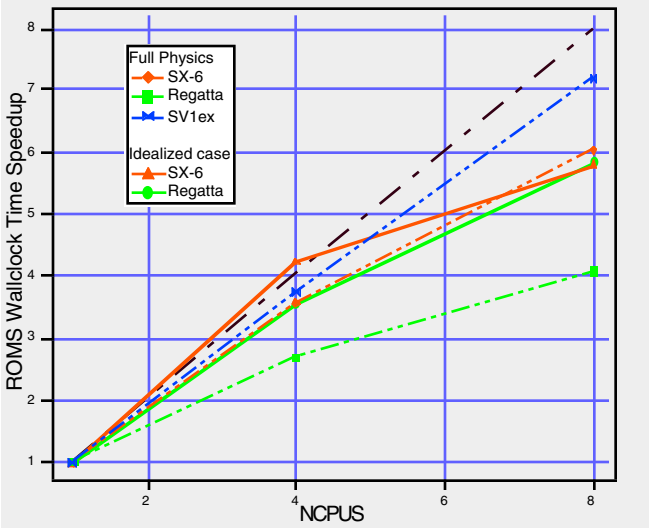

Figure 2: ROMS speedup, 160x160 case

Further work could be done to improve scalability. In particular, ROMS allows the data to be distributed across processor grids of any dimension. Arrays in the above 4-CPU runs were decomposed in a 1x4 arrangement, the 8-CPU runs in a 1x8 arrangement, but further tests on the Regatta showed that 8-CPUs in a 2x4 arrangement increased speedup on that system from 4.0 in the 1x8 arrangement to 5.6.

The user reported one difficulty porting the code to the SX-6. The following OpenMP error in the code ("SINGLE" worksharing constructs appeared outside a parallel region) did not cause problems on the Regatta or other systems, but lead to incorrect results on the SX-6. By deleting the "SINGLE" directives, the user obtained correct runs on the SX-6.

```
!
C$OMP SINGLE
        call get_data
C$OMP END SINGLE
!
! - other serial work -
!
C$OMP PARALLEL DO
C$OMP+          PRIVATE(thread,subs,tile),
C$OMP+          SHARED(numthreads,lock)
        ! - work -
C$OMP END PARALLEL DO
!
! - other serial work -
!
C$OMP SINGLE
        call output
C$OMP END SINGLE
```

FLAPW

FLAPW (Full Potential Linearized Augmented Plane Wave Method) is a materials structure optimization and molecular dynamics code. Runs are compared against the SV1ex. Performance results for 1-CPU runs are given for a test case which reasonably approximates the user's actual production runs.

System	CPU PeakPerf (GFLOPS)	CPU time (sec)	Mflops	Percent of CPU PeakPerf
SV1ex	2.0	1033	624	31%
SX6	8.0	329	1940	24%

Table 10: FLAPW Performance Results, 1-CPU

FLAPW is not parallelized explicitly, but lends itself well to autotasking on the SV1ex, where the user is active. Without making any effort to modify the code, auto-parallelization was performed with the following compiler options, and parallel blas library:

SV1ex: `ftn -Otask2`
SX-6: `f90 -ew -P auto -Wf"-O nodarg" -L/SX/usr/lib -llapack_64 -lparblas_64`

The following graph of speedup shows the results. Wall-clock time from dedicated runs is unavailable, so MFLOPS were used as the basis. The assumption that a code which lent itself to auto-parallelization on the SV1ex would do so on the SX-6, without modification, was proven false with FLAPW. But again, note that one SX-6 CPU has the internal architecture to support 4 times the low-level parallelism that can be exploited in an SV1ex CPU.

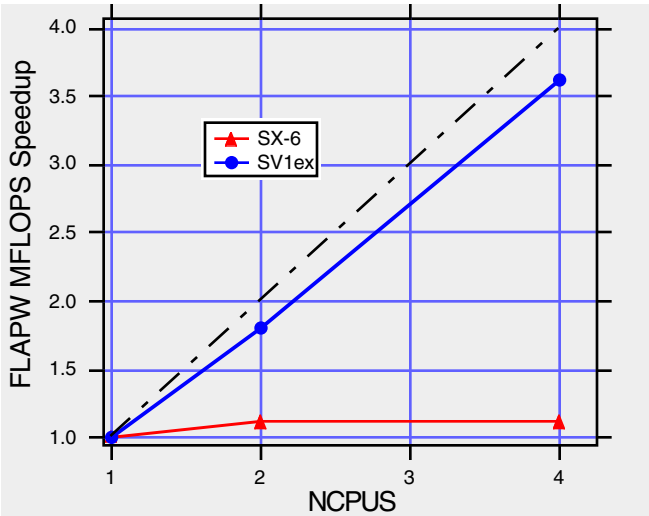

Figure 3: FLAPW speedup

Porting FLAPW from the SV1ex to the SX-6 required both code modifications and tweaking of compiler and linker options. One notable code change was required because the code was compiled using the SX-6 "-ew" option to mimic the 64-bit SV1ex data types. "Wide" is a powerful option, and forces, not just the default types to 64-bits, but also those sized explicitly. This made several generic interfaces, such as "s_gemv" and "d_gemv," below, indistinguishable based on their arguments. The real and double precision variables become identical as 64-bit reals, and one of the two subroutines had to be eliminated from the interface.

```
interface gemv
#ifndef SX6
         module  procedure
z_gemv,c_gemv,d_gemv,s_gemv
#else
         module  procedure c_gemv,s_gemv
#endif

#ifndef SX6
         subroutine
d_gemv(tr,m,n,alph,a,lda,x,incx,beta,y,incy)
         use kinds
         character*1 tr
         integer(kind=ikind) m,n,lda,ldb,incx,incy
         double precision :: a(:,:),x(:),alph,beta
         double precision :: y(:)

         ! - work -
         end subroutine d_gemv
#endif

         subroutine
s_gemv(tr,m,n,alph,a,lda,x,incx,beta,y,incy)
         use kinds
         character*1 tr
         integer(kind=ikind) m,n,lda,ldb,incx,incy
         real :: a(:,:),x(:),alph,beta
         real :: y(:)
         ! - work -
end subroutine d_gemv
```

Another requirement compiling with –ew is to explicitly link with the correct, 64-bit versions of the system libraries: -llapack_64 -lblas_64. Also note that the libraries are not searched cyclicly by default, so they must either be listed in the correct order (as shown above) or the default over-ridden (-Wl"-hlib_cyclic").

No code modifications were made for performance, but getting good single-CPU performance using just the compiler options was an interesting exercise. When compiled with the default optimization options, the code promptly crashed. Recompiled with "ssafe", the code ran correctly, but at a boring 1231 MFLOPS, only twice the SV1ex rate.

An effort was made to approach a higher level of compiler optimization, which started by noting all possible "compile mode" or "-C" settings. Here they are, in order of increasing optimization ("vopt" is the default):

```
-C{debug|ssafe|vsafe|sopt|vopt|hopt}
```

The next step was to replace "ssafe" with its equivalent set of "detailed" options:

```
-Wf"-O nochg nodarg nodiv noiodo nomove
overlap nounroll"  \
-Wf"-pvctl nocollapse nomatmul noouterunroll"
\
-Wf"-Nv"
```

Most SX-6 optimizations can be explicitly specified as either on or off, for instance "unroll" or "nounroll". The "ssafe" detailed options all have the effect of turning an optimization *off*. Eliminating them thus turns optimizations back on. The above list was tested in what approximated a binary search. It was ultimately discovered that only "nodarg" ("dummy arguments are not subject to optimization") was required for the code to run correctly. When compiled as follows:

```
-Wf"-O nodarg"
```

the code ran correctly at 1940 MFLOPS. In this case, user control over detailed options proved quite beneficial.

Tsunami

This is the University of Alaska Fairbank's tsunami modeling code. It is a nested-grid finite element model parallelized using OpenMP, tested by Andrew Lee and Tom Logan of ARSC. The original code scaled poorly on multiple SX-6 CPUs, and an effort was made to improve this. The resulting modification dramatically improved both SX-6 scalability and wall-clock time. Comparison runs were made against the original version running on the SV1ex, and results for runs of both the "original" and "modified" versions on one SX-6 CPU are given in table 11:

System	CPU PeakPerf (GFLOPS)	Wall-clock time (sec)	MFLOPS	Percent of CPU PeakPerf	Code version
SV1ex	2.0	834	297	15%	Original
SX6	8.0	303	909	11%	Original
SX6	8.0	170	1756	22%	Modified

Table 11: Tsunami performance, 1-CPU

The original and modified versions run 2.7 and 4.9 times faster, respectively, on an SX-6 CPU, compared with the originial version on one SV1ex CPU.

Parallelism is explicit in this code, using OpenMP directives. While SV1ex parallel speedup of the original version is moderate, SX-6 speedup is dismal, as shown in Figure 4. Although CPU time is not graphed, each additional SX-6 CPU consumes about as much additional CPU-time as the first. Thus, 8 SX-6 CPUs consume about 8 times the CPU time of 1 CPU, while on the SV1ex, 8 CPUs consume a bit more than 1 CPU.

The modified version on the SX-6 dramatically improves both wall-clock time and speedup. As described below, this non-functional on the SV1ex, so it doesn't appear in the graph.

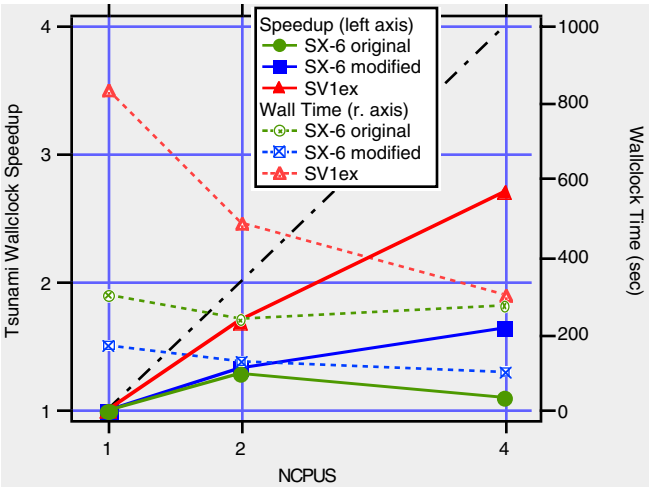

Figure 4: Tsunami speedup and wall-clock time

To improve scaling on the SX-6, the "modified" version eliminates one subroutine (DEPTHS) which did nothing more than take a multi-component user derived type variable and call a work subroutine (DEPTHS_INNER), breaking the components of the derived type apart as individual array and scalar variables. The original DEPTHS subroutine makes this call:

```
 call DEPTHS_INNER(g%mj,g%mk,r,T,
*     g%D,g%DU,g%DV,g%HM,g%SLO,g%UFRIC,g%VFRIC,
*     g%umask,g%vmask,g%DEF)
```

The original DEPTHS_INNER interface looks like this:

```
 subroutine DEPTHS_INNER(mj,mk,r,T,
*     D,DU,DV,HM,SLO,UFRIC,VFRIC,
*     umask,vmask,DEF)
```

And the original DEPTHS_INNER subroutine contains loops like the following, which vectorize on the SV1ex and SX-6:

```
do j=2,mj
  if(umask(j,mk).eq.1.)then
    DU(j,mk)=0.5*(D(j,mk)+D(j-1,mk))
    if(DU(j,mk).lt.500.) DU(j,mk)=500.
    UFRIC(j,mk)=r*T/DU(j,mk)
  end if
end do
```

On this original rendition on the SX-6, profiling reveals that a huge amount of time is consumed in call overhead to DEPTHS_INNER, apparently in creating duplicate copies of all the derived-type component arrays, and this work increases linearly as CPUs are added at run-time. As the speedup curve for the original code on the SX-6 shows, this eliminates benefits from adding additional CPUs. On the SV1ex, it appears that, as one would expect in pass-by-reference Fortran, the addresses of the component arrays are passed without array duplication, and there is no excessive call overhead.

The modified version eliminates DEPTHS_INNER, and simply passes the undeconstructed derived type variable, "g," to DEPTHS, like this:

```
subroutine DEPTHS(g, r, T)
use grid_params
```

and the loops are rewritten as follows:

```
do j=2,g%mj
  if(g%umask(j,g%mk).eq.1.)then
    g%DU(j,g%mk)=0.5*(g%D(j,g%mk)+g%D(j-1,g%mk))
    if(g%DU(j,g%mk).lt.500.) g%DU(j,g%mk)=500.
    g%UFRIC(j,g%mk)=r*T/g%DU(j,g%mk)
  end if
end do
```

On the SV1ex, the `ftn` compiler does not vectorize loops (even when !DIR$ IVDEP is applied) which, due to the ambiguity of pointer arrays, may or may not contain data dependencies. Thus, this second form of the loop doesn't vectorize, and the compiler explains that there is a "possible recurrence." This conservative approach assures correctness (but unvectorized performance is of course, abysmal.). The SX-6 compiler, on the other hand, does vectorize the above loop, and in this case, is rewarded for taking the risk, because the different arrays are indeed disjoint in memory and the output is correct.

RIPPLE

RIPPLE [2] is a program developed at Los Alamos National Laboratory to model incompressible fluid flows with surface tension on free surfaces. Prof. Zygmunt Kowalik and Juan Horrillo at the University of Alaska

Fairbanks are adapting it to the study of tsunamis and their "run-up" on shore. It was originally developed on a Cray running the CTSS operating system and was ported by the developers to run on Unix workstations. Horrillo and Ed Kornkven of ARSC ported it to the SX-6 where it is being modified to a scale appropriate to tsunami research.

RIPPLE maintains a rectilinear mesh upon which fluid volume and forces are calculated over a user-defined number of time steps. Consequently, the amount of calculation for a given simulation will depend on the size of the grid and the number of time steps. At this writing, the model is still being studied and calibrated with relatively small grids. For the timings that follow, a 362x58 grid was used over 1000 time steps which result in execution times on the order of 100 seconds. These runs were shortened by a factor of 10 for the sake of turnaround in our experiments. Single CPU times, with default optimization, are compared against the SV1ex, and, as shown here, the SX-6 CPU run is about 3.9 times faster than the SV1ex run.

System	CPU PeakPerf (GFLOPS)	Real time (sec)	Mflops	Percent of CPU PeakPerf
SV1ex	2.0	480	181	9%
SX6	8.0	123	813	10%

Table 12: RIPPLE Performance Comparisons, 1-CPU

Much of our work, below, on the SX-6 concerned compiler optimizations, thus, we present more detailed comparisons between the SX-6 and SV1ex. The code was compiled and executed several times, each time attempting to use comparable compile options. The resulting execution times are as follows:

Compile Event	**SV1ex Options**	**SX-6 Options**	**SV1ex Exec Time (sec)**	**SX-6 Exec Time (sec)**	**SX-6 Speedup (x)**
Default optimizations	Default optimization (-O2), default inlining, -dp	Default optimization, default inlining, -ew	480.56	123.13	3.90
Inlining, default optimization	-inline2	-pi nest=2 line=200	434.19	115.25	3.77
No inlining, max optimization	-O3	-Chopt	461.25	115.21	4.00
Inlining, max optimization	-inline2, -O3	pi nest=2 line=200, -Chopt	442.22	102.22	4.33
Run time profiling, default optimization	–ef –lperf	–ftrace	1673.88	424.50	3.94

Table 13: Detailed performance comparisons, 1-CPU

As the table shows, the SX-6 runs RIPPLE about four times faster than the SV1ex.

As our final comparison against the SV1ex, we obtained the execution-time profiles of the two platforms. It is a common practice when optimizing a program to do a trace of the code in order to locate the most computationally expensive subroutines, functions, or blocks. As expected, the traces for a program like RIPPLE, which is deterministic, were very similar for the two vector architectures.

In porting RIPPLE to the SX-6, extensive exploration of the many compiler options was made. One goal of this test was to determine how much "tinkering" with compiler options might be necessary to achieve optimal performance. The following results show that the high-level –C compiler options, in particular, the default –Cvopt option, do an excellent job of optimizing the code. The following table presents the differences between the –C alternatives:

Compiler Options	**Compile Time (sec)**	**Avg Execution Time (sec)**	**% of -Cvopt**
-Chopt	50.26	84.93	89.77
-Cvopt	46.68	94.61	100.00
-Cvsafe	31.81	146.70	155.05
-Csopt	28.06	636.97	673.26
-Cssafe	27.28	682.40	721.28
-Cdebug	22.60	2664.81	2816.62

Table 14: SX-6 f90 high-level compiler options compared

Note that RIPPLE spends about 91% of its time in vector instructions and 9% in scalar (according to the "vector percent" run-time performance output from F_PROGINF=DETAIL). Consequently, options that quash vectorization (-Csopt, -Cssafe, and -Cdebug) will perform poorly. -Chopt performs the best, but uses some aggressive optimizations that do not suit all applications. We have seen applications that abort under -Chopt but not under -Cvopt.

The second compiler option test involved the SX-6 "detailed options," combining switches and looking for variances in the execution times. The "control" compile option was –Cvopt. Consequently, the analysis is looking for differences due to adding and subracting options from that (already high) optimization level. The differences found were all negative; i.e., all the options that made a significant difference to the execution time lengthened, rather than shortened, execution time. The significant options found are shown in table 15. Note that these all remove an optimization function from vopt (e.g., "nodarg", as seen in the FLAPW section, indicates "no" "darg" – no dummy argument optimization). There was no option included in "hopt" which improved the performance of "vopt."

Option Tested	Default for -Cvopt	Significance (Fα)	Est. % Added to Wall-clock time
-O darg	√		
-O nodarg		α=.01	69%
-O div	√		
-O nodiv		α=.01	3%
-O extendreorder	√		
-O reorderrange=bblock		α=.05	8%
-pvctl altcode	√		
-pvctl noaltcode		α=.01	32%
-pvctl expand=4	√		
-pvctl expand=12			
-pvctl noexpand		α=.01	2-3%

Table 15: Detailed options with significant effect when added to -Cvopt

Options that were tested but yielded no significant change in execution time are: –O {chg, nochg}, -O {move, nomovediv, nomove}, -O {overlap, nooverlap}, -O {unroll, nounroll}, –O {fusion, Nfusion}, -O {if, noif}, {-ai, -Nai}, -pvctl {ifopt, noifopt}, -pvctl {inner, noinner}, -pvctl {assoc, noassoc}, -pvctl {collapse, nocollapse}, –pvctl {assume, noassume}, -pvctl {loopchg, noloopchg}, -pvctl {loopfusion, noloopfusion}, {-G, -NG}, -pvctl {outerunroll, noouterunroll}, –pvctl {listvec, nolistvec}, -pvctl {vchg, novchg}, -pvctl vwork= {static, stack}.

The SX-6 port itself was relatively straightforward. The complicated make scripts that were supplied were abandoned in favor of one generated by *fmgen*, a makefile generator, which was modified to fit the directory structure of the RIPPLE source. After a few source changes to remove some features that were particular to the 32-bit workstation version, the program compiled. The only other notable part of the build process was the decision to make use of the *–ew* compiler option which, as discussed earlier, forces 64-bit storage. We note that the SX-6 appears to be very well outfitted to host users of older Cray machines. For example, it supplied some very old Cray routines that RIPPLE uses which were not even found on ARSC's SV1ex when RIPPLE was ported to that machine. The main differences between the SX-6 porting environment and that of the SV1ex are summarized in the following table:

Feature Differences	SV1ex	SX-6
fdate(), etime(), hostnm() functions (presumably from CTSS days)	Had to add these missing functions	Supplied
kill(), ismin(), ismax(), second() functions	Had to rename or remove to avoid conflict with existing function names	No conflict
Data sizes	Compiled with "–dp" to disable double precision arithmetic	Compiled with –ew to force 64-bit data values
Double functions dmin1(), dmax1(), dabs(), dsqrt()	Replaced with generic equivalents to avoid type errors	Used as-is
IOTTY	Unknown to system	By default represents "stdout" unit

Table 16: SV1ex and SX-6 porting difference with RIPPLE

Summary

Objective Measures

The SX-6 experience can be evaluated subjectively, but first a recap of objective measures. As noted earlier, the SX-6 f90 compiler offers the user 58 optimization options (versus 22 for the SV1ex), 39 run-time parameters (versus several execution control command for the SV1ex), and, of course, each CPU is basically 4-times as strong, in vector pipes, vector register size, and peak theoretical GFLOPS as one SV1ex CPU, and has about 9 times the per CPU memory bandwidth,

Speedup of the tested codes is summarized in figure 5. In this graph, circular markers designate the OpenMP codes, inverted triangular markers, the auto-parallelized codes, and a square marker, the MPI code. There are four metrics of speedup plotted, in columns. The first, labeled "SV1ex-1p" shows the speedup for each code running on 1 SX-6 CPU versus 1 SV1ex CPU (speedup computed as $walltime^{SV1ex}$ / $walltime^{SX-6}$). All codes but two appear in the range of 3 to 4 times faster on the SX-6. A factor of 4 speedup might be expected given that these are fundamentally similar architectures and the SX-6 processor offers 4 times the performance of the SV1ex.

As shown, however, the ROMS idealized model was almost 11 times faster on the SX-6 than on the SV1ex. We believe that this speedup is a function of memory bandwidth. The computational intensity (defined as number of numerical operations per memory access) of this model is about one, as measured on the SX-6, and thus, its performance is limited by the lower of the processor performance or memory bandwidth. As previously noted, the SX-6 provides 9 times the memory bandwidth of the SV1ex. We conclude that this code is memory bandwidth limited, and thus its performance is degraded relative to the CPU capability on the SV1ex, but that on the SX-6, this roadblock is removed.

Tsunami also obtained greater than the 4 times speedup predicted by the SX-6 CPU alone. The Tsunami value is a comparison between the modified code running on the SX-6 and the original on the SV1ex. Although Tsunami is the sole code specifically optimized for the SX-6, we believe the comparison is valid because the modification didn't affect the computational loops of the code, but only the structure of subroutine calls. A comparison of profile outputs reveals that the SV1ex does not suffer the excessive call overhead which was removed for the SX-6 in the modification. As with the idealized ROMS case, Tsunami is memory bandwidth limited on the SV1ex, as described in [3], and suggested by its "vector percentage" of over 99% on the SX-6, as measured under F_PROGINF. The worst code in the SX-6 versus SV1ex single CPU comparison is the original version of Tsunami. As noted above, the SX-6 creates excessive call overhead on this code, and thus the SV1ex compares very well.

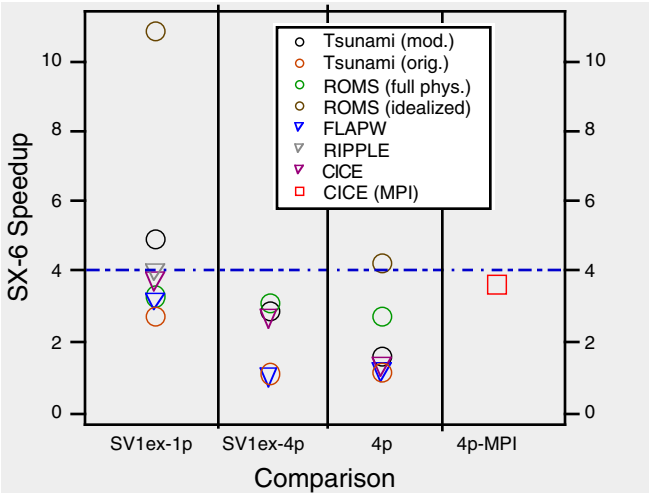

Figure 5: SX-6 wall-clock speedup

The second metric, "SV1ex-4p," shows the speedup going from 4 SV1ex CPUs to 4 SX-6 CPUs, under autotasking, auto-parallelization, or OpenMP. The third metric, labeled "4p" shows the speedup in going from 1 to 4 SX-6 CPUs, using auto-parallelization or OpenMP. The fourth metric, "4p-MPI" shows speedup going from 1 to 4 SX-6 CPUs under MPI.

Under the shared memory methods, the codes generally scale worse on the SX-6 than on the SV1ex, and speedup from 1 to 4 SX-6 CPUs is less than ideal. Two explanations present themselves, first, that SV1ex autotasking is simply better than SX-6 auto-parallelization. Second, the previously mentioned idea that vectorization alone on the SX-6 extracts most of the low-level parallelism inherent in these codes, just to keep one of the 8-pipe CPUs busy. The good speedup of the MPI version of CICE, especially when compared with the auto-parallelized version of the same code, suggests that sufficient work exists in this code to be processed efficiently in parallel, but that on the SX-6, the programmer must extract it explicitly, at a larger granularity than that available to the compiler.

Subjective Measures

This paper is largely a series of case-studies, and thus it seems appropriate to conclude with some subjective comments by the users themselves.

Jim Long reports that the SX-6 is "*really* nice", and he wishes he had a single-CPU desk side server of his own. Kate Hedstrom reports that she likes the picky compiler and the information provided by F_PROGINF=DETAIL, and adds: "I thought it was interesting how many options are controlled by environment variables. I didn't use it enough to decide if I liked that, though." The author appreciates the extensive documentation, access to detailed compiler options, performance, and cross-compilers.

References

[1]: Halting (Or Not) on Numerical Exceptions, G. Robinson, ARSC, ARSC HPC Users Newsletter, http://www.arsc.edu/support/news/HPCnews.shtml, issue 264, Feb 21, 2003

[2]: Douglas B. Kothe, Raymond C. Mjolsness, Martin D. Torrey, "RIPPLE: A Computer Program for Incompressible Flows with Free Surfaces", Los Alamos National Laboratory, LA-12007-MS, April 1991.

[3]: SV1ex Memory Upgrade Gives Greatest Boost to User Performance, T. Baring, ARSC, Proceedings of the Cray User Group, May 2002

Acknowledgements

Jacqui Warren (Cray Inc.), Claudia David (Cray Inc.), John Metzner (Cray Inc.), Enrique Lopez-Pineda (Cray Inc.), Nick Chepurniy (Cray Inc.), Dave Parks (NEC Solutions America, Inc.), Elena Suleimani (University of Alaska Fairbanks, Geophysical Institute), Cathrine Stampfl (Northwestern University), Wieslaw Maslowski (Naval Postgraduate School)), Ed Kornkven (ARSC), Tom Logan (ARSC), Kate Hedstrom (ARSC), Andrew Lee (ARSC)), Guy Robinson (ARSC), Jim Long (ARSC).

Author

Tom Baring is a Vector Specialist at the Arctic Region Supercomputing Center, and may be reached at baring@arsc.edu.