X1 Porting Experiences

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

ABSTRACT: *ARSC staff and users have ported a number of codes, both parallel and serial, to a 128 processor Cray X1 that was put into full production at ARSC on November 4, 2003. This paper presents some of the challenges and rewards from these porting experiences, and will describe specific optimization techniques attempted and performance realized.*

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.

ARSC commenced the physical installation of a 128-MSP Cray X1 in May, 2003 and completed it in August. The programming environment has progressed from version 4.3 at original install to 5.2 at the present time, with numerous bug fixes and feature additions.

This paper traces the porting and optimization of several codes for the Cray X1. It presents an overview of the challenges and potential for performance and by example points out some of the difficulties.

User Codes

BH

BH is an ab initio quantum chemistry code for predicting material properties. It has good vector and parallel performance, and has been ported to a number of platforms, notably the user's local cluster, SGI Origins at the Engineer Research and Development Center MSRC (ERDC), and the SX-6 at ARSC. Basic attributes of this code are shown in table 1.

Program	**BH**
Language	Fortran / C
Explicit Parallelism	MPI
Lines of code	~9,500

table 1: Basic attributes of BH

Researchers at ARSC, ERDC, and the research group which uses BH are collaborating on the X1 version. I did the original port, and Sam Cable, of ERDC, and I are working together on optimization and on-going testing.

One of the challenges in porting BH was simply learning to navigate the code and its unique build process. The build process uses a series of scripts and include files to promote portability and to produce a separate executable for each simulation, or experiment. The process is quite powerful, as it hides platform specific code. Briefly, here are the steps to building a new executable:

1) The researcher generates a collection of simulation data files and an appropriate include file ("PARAM") of Fortran parameter definitions for compiling the executable for a specific experiment.
1) The top level build script ("goM"):
    a. launches a second script ("compgen") which generates 9 new scripts containing pre-processor, Fortran, C, and linker command lines.

a. copies platform specific files (36 include, 2 Fortran source, and 4 C source files) to a common source directory.
a. executes "make" against a makefile which is updated manually and separately.

Having been ported to numerous platforms, BH offered several examples to use in generating the X1 platform specific include and source files. A costly mistake, in terms of time and effort, was in basing the X1 LAPACK calls on a much earlier port to "cray" rather than the current port to either "sgi" or "t3e." In particular, the use of one specific include file is shown in this snippet of Fortran source from BH:

```
include(lapacktrf)
     $            (maxpm,maxpm,crmm,maxdim,ipvt,info)
```

The file, lapacktrf must contain "`call cgetrf`" but, based on the old "cray" port, I initially used "`call sgetrf`". The error revealed itself as gradual convergence to an incorrect solution. Table 2 lists the three main porting issues.

Complexity of source structure and build process
Data size issues
Retired system calls and intrinsics

table 2: BH Porting issues

BH requires default 64-bit data types for numerical accuracy, and thus must be compiled with "`-s default64 -dp`". Use of these options, however, caused problems for certain utilities and intrinsics that require(d) 32-bit arguments. Programming environments 5.0 (PE 5.0) and earlier, offer only a default type (i.e., `REAL*4`) version of the `erf` and `erfc` routines. Thus, with `-s default64` in force in PE 5.0, the following treatment was required:

```
REAL(KIND=4) ERFC
…
vr = ERFC(REAL(A=(alphrt*r), KIND=4))/r
```

PEs 5.1 and higher provide a double precision implementation, `DERFC`. The following update to the port provides more consistent coding and results with other platforms:

```
vr = DERFC(alphrt*r)/r
```

Similarly, the I/O utility, `FLUSH`, crashes the program if called with an `INTEGER*8` argument. Thus, with `-s default64`, the argument to `FLUSH` must be explicitly converted to 32-bits. Here are two examples from BH:

```
call flush (7_4)
call flush (INT(A=iounit, KIND=4))
```

Certain system calls and intrinsic routines had to be replaced. For instance, I replaced `itime` with the Fortran 90 intrinsic function, `date_and_time`.

The final phase of porting is optimization. A predecessor of this code was run extensively on Cray platforms, and it inherited some excellent vector properties. However, to capitalize on these properties and achieve good performance, some optimization was required.

Memory resident scratch files	
Vectorization/Streaming	Inline subroutines and functions
	Eliminate dependencies
	Eliminate conditionals
	Manually unwind loops
	Eliminate redundant computations
	Insert Cray Fortran compiler directives

table 3: BH Optimizations

The first profile of BH using `cray_pat` revealed that the top two consumers of time were file read and write. The code uses scratch files as a mechanism for rotating certain large arrays (e.g., cy(i,k,j) = cx(i,j,k)). It writes the arrays, and rereads them with the rotation. The number of scratch files increases with the number of MPI tasks.

Fortunately, the X1 provides adequate memory for these files to be mapped to memory and a simple mechanism for doing so, the run-time `assign` command. Here's a section of the PBS script used to run a large BH experiment, showing the appropriate assignments:

```
export FILENV=\$EVAR
eval $(assign -F mr.scr.ovfl.start_size=72  p:temp%)
eval $(assign -F mr.scr.ovfl.start_size=10342 p:WAVE%)
```

These assignments map all files with names matching the regular expressions "temp.*" and "WAVE.*" to memory, deletes them on closure, allows them to overflow to disk if they grow too large, and starts them off at 72 and 10342 disk blocks (4096 bytes per block), respectively. I wrote a utility program to compute the appropriate `start_size` for these files based on the dimension of the Fortran arrays and a specific experiment's PARAM file. I assume that giving a `start_size` equivalent to the known final size aids performance, but haven't tested this.

When profiled again, file I/O had been reduced to less than 10% of the total run time. Three user subroutines had bubbled to the top of the list in its place.

The top routine (excno) contained one critical loop that contains 100 lines of preprocessed source, two called subroutines (another 150 lines), 7 nested smaller loops, and three conditionals. Initially, vectorization was inhibited by the subroutine calls. To inline them, modifications to the build process were required. The modifications 1) preserve previously deleted intermediate source files produced by preprocessor commands, 2) compile the source inline files without inlining them, 3) provide a list of "inlinefrom" files to the `ftn` commands for the desired destination routines. Despite successful inlining, vectorization of the loop was still inhibited. `Ftn` reported the following dependency on the variable, `ecrs`:

```
       !!! igga is a loop invariant.
       do 8100 n=1,nplwv
              if (igga.GT.0) then
                ecrs = …
              else
                !!! ecrs not set
              endif
              if (igga.EQ.1) then
                … = ecrs …
              endif
8100 continue
```

The compiler couldn't divine that the conditions in the "if" statements guarantee that `ecrs` will always be defined prior to use. By initializing the variable (to any value) immediately above the loop, the compiler unconditionally vectorized it. As a side note, the SX-6 vectorizing compiler doesn't vectorize this particular loop either with or without the modification.

With "excno" fully vectorized and streamed, the top routine became "potnl1." The original version contained several loops like the one that follows. Note that"nallkp(n,iks)" is large and thus is the best candidate for vectorization and streaming:

```
  386.  1 2 M 4---------<        do m=1, nallkp(n,iks)
  387.  1 2 M 4                    cfcf=catstf(m,l,j,n)*flqint(m,ml,j,n)*cg(m)
  388.  1 2 M 4
  389.  1 2 M 4 Vs------<           do jj1=1,3
  390.  1 2 M 4 Vs W----<              do jj2=1,3
  391.  1 2 M 4 Vs W W--<                 do jj3=1,3
  392.  1 2 M 4 Vs W W                     ctt(jj1,jj2,jj3)=
  393.  1 2 M 4 Vs W W         $          ctt(jj1,jj2,jj3)+rleng(jj1,m,n)
  394.  1 2 M 4 Vs W W         $          *rleng(jj2,m,n)*rleng(jj3,m,n)*cfcf
  395.  1 2 M 4 Vs W W-->                 enddo
  396.  1 2 M 4 Vs W---->              enddo
  397.  1 2 M 4 Vs------>           enddo

ftn-6254 ftn: VECTOR File = potnl1temp.F, Line = 386
  A loop starting at line 386 was not vectorized because a recurrence was found
  on "CTT" at line 392.
```

Not believing (or understanding) the waring about the recurrence, I tried adding PREFERVECTOR and PREFERSTREAM directives to the loop at line 386, but this was insufficient to persuade `ftn` to vectorize the loop. Adding these directives and then eliminating the inner loop nest by manually unwinding it, however, allowed the compiler to multi-stream and vectorize the loop. This is indicated by the "M" and "V" codes in the loopmark listing for the optimized version:

```
382.  1 2 3           !DIR$ PREFERSTREAM
383.  1 2 3           !DIR$ PREFERVECTOR
384.  1 2 3 MV------<  do m=1, nallkp(n,iks)
385.  1 2 3 MV           cfcf=catstf(m,l,j,n)*flqint(m,ml,j,n)*cg(m)
386.  1 2 3 MV
[ … documention cut …]
399.  1 2 3 MV      ctt(1,1,1)=ctt(1,1,1)+rleng(1,m,n)*rleng(1,m,n)*rleng(1,m,n)*cfcf
400.  1 2 3 MV      ctt(1,1,2)=ctt(1,1,2)+rleng(1,m,n)*rleng(1,m,n)*rleng(2,m,n)*cfcf
401.  1 2 3 MV      ctt(1,1,3)=ctt(1,1,3)+rleng(1,m,n)*rleng(1,m,n)*rleng(3,m,n)*cfcf
402.  1 2 3 MV      ctt(1,2,1)=ctt(1,2,1)+rleng(1,m,n)*rleng(2,m,n)*rleng(1,m,n)*cfcf
403.  1 2 3 MV      ctt(1,2,2)=ctt(1,2,2)+rleng(1,m,n)*rleng(2,m,n)*rleng(2,m,n)*cfcf
```

An alternative to manually unwinding, discovered later, is to apply the directive "UNROLL 3" to each of the loops in the loop nest. Testing showed that `ftn` vectorizes the wrong loop if "UNROLL," is specified without the depth of 3, or if "UNROLL 3" is not applied to all three loops. (For comparison, the SX-6 f90 compiler unwound these loops and vectorized on the `nallkpt` loop without compiler directives or any other changes to this routine.)

The third and final routine eligible for vectorization to bubble up in the profile was "forcnl". Sam Cable of ERDC optimized this routine, enabling streaming, eliminating redundant computations, and improving vectorization. Note the conditionals in the following snippet of code from the original version:

```
500.  1 2 W 4 5----<             do 9401 ml = 1,nl(j)
501.  1 2 W 4 5                    if (ilall(ml,j).eq.0) then
502.  1 2 W 4 5 Vr-<                 do 2010 m=2,nallkp(n,iks)
503.  1 2 W 4 5 Vr                     cpart(m)=catstf(m,l,j,n)
504.  1 2 W 4 5 Vr       $                   *cptwfp(m,nn,n)
505.  1 2 W 4 5 Vr       &                   *flqint(m,ml,j,n)
506.  1 2 W 4 5 Vr                     ct1=ct1+cpart(m)*lpctfx(igx(m,n))
507.  1 2 W 4 5 Vr                     ct2=ct2+cpart(m)*lpctfy(igy(m,n))
508.  1 2 W 4 5 Vr                     ct3=ct3+cpart(m)*lpctfz(igz(m,n))
509.  1 2 W 4 5 Vr->  2010           continue
510.  1 2 W 4 5                    endif
511.  1 2 W 4 5                    if (ilall(ml,j).eq.1) then
512.  1 2 W 4 5 V--<                 do 2011 m=2,nallkp(n,iks)
513.  1 2 W 4 5 V                     cpart(m)=catstf(m,l,j,n)
514.  1 2 W 4 5 V                  $        *cptwfp(m,nn,n)
515.  1 2 W 4 5 V                  &        *flqint(m,ml,j,n)
516.  1 2 W 4 5 V                     cparx2(m)=cpart(m)*rleng(1,m,n)
```

The primary optimization moved all conditional statements from the interior of loops to a single preparatory loop, created a lookup table and temporary storage for precomputed values, and split one large loop into four smaller loops corresponding to the erstwhile conditional statements. Loopmark listings of the optimized code show our friend, "MV":

```
534.  1 2 W 4 MV----<                do m=2, nallkp(n,iks)
535.  1 2 W 4 MV r--<                  do ml = 1, nl(j)
536.  1 2 W 4 MV r                       cpart2(m,ml)=catstf(m,l,j,n)*
537.  1 2 W 4 MV r         $                 cptwfp(m,nn,n)*flqint(m,ml,j,n)
538.  1 2 W 4 MV r-->                    enddo
539.  1 2 W 4 MV---->                   enddo
540.  1 2 W 4 MV----<                   do 2010 m=2,nallkp(n,iks)
541.  1 2 W 4 MV 6--<                     do ml=1, numilall0
542.  1 2 W 4 MV 6                         ct1=ct1+cpart2(m,ilallis0(ml))*
543.  1 2 W 4 MV 6         $                 lpctfx(igx(m,n))
544.  1 2 W 4 MV 6                         ct2=ct2+cpart2(m,ilallis0(ml))*
545.  1 2 W 4 MV 6         $                 lpctfy(igy(m,n))
546.  1 2 W 4 MV 6                         ct3=ct3+cpart2(m,ilallis0(ml))*
547.  1 2 W 4 MV 6         $                 lpctfz(igz(m,n))
548.  1 2 W 4 MV 6-->                     enddo
549.  1 2 W 4 MV----> 2010              continue
550.  1 2 W 4 MV----<                   do 2011 m=2,nallkp(n,iks)
551.  1 2 W 4 MV 6--<                     do ml=1, numilall1
```

```
552.  1 2 W 4 MV 6                        cparx2(m)=cpart2(m,ilallis1(ml))*rleng(1,m,n)
```

Farther down in the routine, in a different section of logic, the lookup table was used again, to eliminate an entire loop. In the following listing, this is the fifth nested loop, marked “M” to indicate that the compiler had multi-streamed it:

```
907.  1 2 W 4 M 6 7------------<            do i=1,3
908.  1 2 W 4 M 6 7 8----------<               do i2=1,3
909.  1 2 W 4 M 6 7 8 9--------<                  do i3=1,3
910.  1 2 W 4 M 6 7 8 9 W------<                     do m=1,3
911.  1 2 W 4 M 6 7 8 9 W W----<                        do m2=1,3
912.  1 2 W 4 M 6 7 8 9 W W W--<                           do m3=1,3
913.  1 2 W 4 M 6 7 8 9 W W W                                cterm=cterm+
914.  1 2 W 4 M 6 7 8 9 W W W       &                          ceinl3(i,i2,i3,k)
915.  1 2 W 4 M 6 7 8 9 W W W       $                          *conjg(cg3ank(
916.  1 2 W 4 M 6 7 8 9 W W W       $                           l,j,m,m2,m3,nn,n))
917.  1 2 W 4 M 6 7 8 9 W W W       $                          *amet(i,m)*
918.  1 2 W 4 M 6 7 8 9 W W W       $                           amet(i2,m2)*
919.  1 2 W 4 M 6 7 8 9 W W W       $                           amet(i3,m3)
920.  1 2 W 4 M 6 7 8 9 W W W-->                           enddo
921.  1 2 W 4 M 6 7 8 9 W W---->                        enddo
922.  1 2 W 4 M 6 7 8 9 W------>                     enddo
923.  1 2 W 4 M 6 7 8 9-------->                  enddo
924.  1 2 W 4 M 6 7 8---------->               enddo
925.  1 2 W 4 M 6 7------------>            enddo
```

With that loop eliminated, the compiler was able to vectorize on the next deeper loop. This vectorization included several deep loop nests with low iteration count, as shown in the optimized version, below:

```
964.  1 2 W 4 V i---------------<           do i=1,3
965.  1 2 W 4 V i i-------------<              do i2=1,3
966.  1 2 W 4 V i i ir----------<                 do i3=1,3
967.  1 2 W 4 V i i ir 9--------<                    do m=1,3
968.  1 2 W 4 V i i ir 9 10-----<                       do m2=1,3
969.  1 2 W 4 V i i ir 9 10 11--<                          do m3=1,3
970.  1 2 W 4 V i i ir 9 10 11                               cterm=cterm+
971.  1 2 W 4 V i i ir 9 10 11         &                        ceinl3(i,i2,i3,k)
972.  1 2 W 4 V i i ir 9 10 11         $                        *conjg(cg3ank(
973.  1 2 W 4 V i i ir 9 10 11         $                        l,j,m,m2,m3,nn,n))
974.  1 2 W 4 V i i ir 9 10 11         $                        *amet(i,m)*
975.  1 2 W 4 V i i ir 9 10 11         $                        amet(i2,m2)*
976.  1 2 W 4 V i i ir 9 10 11         $                           amet(i3,m3)
977.  1 2 W 4 V i i ir 9 10 11-->                          enddo
978.  1 2 W 4 V i i ir 9 10----->                       enddo
979.  1 2 W 4 V i i ir 9-------->                    enddo
980.  1 2 W 4 V i i ir---------->                 enddo
981.  1 2 W 4 V i i------------->              enddo
982.  1 2 W 4 V i--------------->           enddo
```

Performance gains due to the optimizations are shown in table 4 for one important benchmark. Timings on the Cray SX-6 system are shown for comparison. For both the X1 and SX-6 runs, 4 processors were used, and, on both machines, the entire application image fit within a single shared memory node. Differences in the compilers and operating system which revealed themselves in this exercise are: 1) the SX-6 is not configured for convenient mapping of files to memory, 2) the modification to the excno subroutine did not enable vectorization under the SX-6 f90 compiler, 3) the modification to the potnl1 subroutine was irrelevant because the SX-6 f90 compiler successfully vectorized the correct loop in the first place, 4) the changes to forcnl, given last, is the only X1 optimization which affected SX-6 performance. There was no attempt to optimize this code for the SX-6.

BH Code Version	Memory Mapped Run	ARSC X1	ARSC SX-6
Original	no	2.8	4.6
Optimized	no	2.5	3.3
Original	yes	1.7	-
Optimized	yes	1.1	-

table 4: BH “BIG” Benchmark wall-clock run time, hours

Note that optimization efforts are not complete. Elimination of the scratch file method for array rotation, additional profiling, and evaluation of MPI performance are next steps. However, the user is pleased with the current level of performance and has commenced production runs.

FLAPW and PFLAPW

These are ab initio quantum chemistry code for predicting material properties. I ported both the latest serial version (FLAPW) and parallel version (PFLAPW) to the X1. Although an earlier serial version has been run extensively on previous ARSC PVP systems, this was the first time I'd seen the parallel version. Both versions have good vector performance, and have been ported to a number of other platforms. Basic attributes are shown in table 5.

Program	**FLAPW**	**PFLAPW**
Language	Fortran	Fortran
Explicit Parallelism	none	MPI
Lines of code	~47,000	~58,000

table 5: Basic attributes of FLAPW

The serial FLAPW ported easily to the X1. Like BH, it requires 64-bit default precision. Simply compiling with `-s default64` worked. Porting PFLAPW was a more challenging, and more important, project.

The greatest challenge was that PFLAPW is parallelized using ScaLAPACK and BLACS, but these libraries are as yet unavailable in the 64-bit version of LibSci. Double precision, 64-bit versions of the ScaLAPACK routines exist in the 32-bit library, but most variables in PFLAPW are declared as default `REAL` or `COMPLEX`. Thus, the problem was to change the declarations from the 32-bit default to 64-bit, mimicking the behavior of `-s default64`, yet still compile with `-s default32` so that the ScaLAPACK library could be used. I arrived at the solution of a custom pre-processor for this code, written in perl. It does the following:

1) promotes function and variable declarations from REAL to DOUBLE PRECISION
1) promotes COMPLEX to COMPLEX*16
1) changes IMPLICIT REAL (A-H,O-Z) to DOUBLE PRECISION (A-H,O-Z)
1) adds IMPLICIT DOUBLE PRECISION (A-H,O-Z) to files lacking any IMPLICIT statement
1) changes all real literals to KIND=8
1) changes all EE notation real literals to double precision
1) replaces intrinsic functions, REAL, CMPLX, SIGN with double precision versions
1) replaces ALOG, AMAX1, AMIN1 with generic equivalents

In its current form, the preprocessor is essentially a search and replace utility with some smarts and sensitivity to context. Perl offers powerful regular expression syntax for pattern matching and is good for performing text manipulations of this sort. The preprocessor is 79 lines of actual code, and is invoked as follows in the PFLAPW makefile:

```
CUSTOMPP=  $(HOME)/preprocX1.prl
[…]
.f.o:
        cd $(@D); \
        $(CUSTOMPP) $(<F) $(*F).p.f; \
        $(CF) -c $(FFLAGS) $(*F).p.f; \
        mv $(*F).p.o $(*F).o; \
        rm  $(*F).p.i $(*F).p.f
```

My hope in writing a custom preprocessor was that it would obviate any changes to the user's source code. Unfortunately, it doesn't handle all legal Fortran syntax and, in the end, I modified two source files (out of 384), to disambiguate confusing, yet valid, Fortran code for the benefit of the preprocessor.

No default 64-bit version of ScaLAPACK / BLACS
Optimization of some files causes program to crash
File I/O issues

table 6: FLAPW / PFLAPW Porting issues

There were two additional porting issues, as noted in table 6. Under PE5.0, optimization of two particular files at level `-O1` or higher caused the program to crash. Fortunately, the files in question contained routines for the parsing of input files and are inconsequential for performance, and PE5.1 fixed the problem. The second problem occurs when the program reads two scratch

files which it has just written itself. For unknown reasons, the problem disappears when the file blocking is switched from the X1 default of f77 to COS, using the following run-time `assign` command:

```
export FILENV=\$EVAR
eval $(assign -F COS u:166)
```

Incidentally, the custom preprocessor proved a useful tool in tracking this bug. I easily modified it to locate all operations on the file in question and insert debugging print statements based on context. I was thus able to convince myself that the file reads were consistent with the writes, and started looking elsewhere for the problem. Commenting out this bit of perl code in the preprocessor, and recompiling, removed the print statements from PFLAPW, all with no modifications to the user's Fortran source.

The following plot shows the run time of my initial PFLAPW X1 port, as compared against ports to six other platforms. (This plot, provided by the user, shows wall-clock time, not of entire runs but of a significant component of a run, matrix diagonalization.)

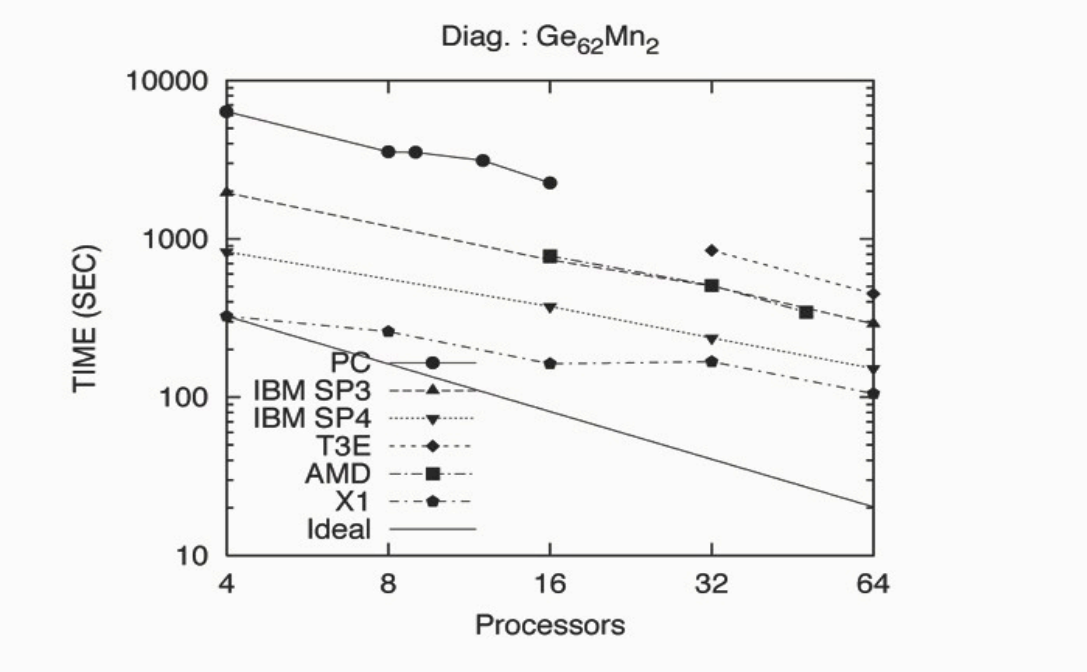

Figure 1: PFLAPW wall clock time for matrix diagonalization

In this plot, two features are immediately apparent. Performance of the X1 port is excellent by comparison, but scaling could stand improvement.

Memory resident scratch files	
Vectorization/Streaming	Inline
	Promote scalars and arrays
	Push loop into subroutine

table 7: FLAPW / PFLAPW Optimizations

Once the correct output was obtained from the basic port, I profiled the code with `cray_pat`. In the largest benchmark to which I had access, about 38% of PFLAPW's time was spent in one user subroutine ("pbe"), and after that, essentially all its time was in MPI routines called from ScaLAPACK routines. The subroutine in question was originally called from within a non-vectorizing loop. Automatic inlining didn't allow the loop to vectorize, and the compiler reported dependencies on variables two and three levels down in the inlined call tree.

Promoting scalar arguments to 1-D arrays, promoting the existing arrays, and adding the length of the arrays as one additional argument allowed me to push the loop inside the pbe subroutine. The following code snippets are from a (condensed) loopmark listing of the caller, showing the essentials of the change. Prior to optimization, the loop in the calling subroutine looked like this:

```
real rx(3,nspd),f(nspd),gw(nspd), pos0(3),
&    ex(2),vx(2),vc(2), exlda(2),vxlda(2),vclda(2), rh3(2)

123 ----<   do 140 k=1,nsp
123           rh3(1)=rhoxc(k,1)            *real(jspins)/2.0
123           rh3(2)=rhoxc(k,min(2,jspins))*real(jspins)/2.0
123
123           call pbe(jspins,rh3,agr(1,k),delta(1,k),dlap(1,k),
123         &          ex,vx,ec,vc,  exlda,vxlda,eclda,vclda)
123 r--<      do 30 js=1,jspins
123 r--> 30     fxc(k,js)= vx(js)+vc(js)
```

After optimization, the same code section looks like this:

```
real, allocatable :: exlda(:,:),vxlda(:,:),vclda(:,:),eclda(:)
real, allocatable :: ex(:,:),vx(:,:),vc(:,:),rh3(:,:),ec(:)
allocate(ex(2,nsp), vx(2,nsp), vc(2,nsp), rh3(2,nsp), ec(nsp))
allocate(exlda(2,nsp), vxlda(2,nsp), vclda(2,nsp), eclda(nsp))

12 MV---<    do k=1,nsp
12 MV          rh3(1,k)=rhoxc(k,1)            *real(jspins)/2.0
12 MV          rh3(2,k)=rhoxc(k,min(2,jspins))*real(jspins)/2.0
12 MV--->    enddo
12
12          call pbe(nsp,jspins,rh3,agr,delta,dlap,
12         &           ex,vx,ec,vc,
12         &           exlda,vxlda,eclda,vclda)
12
12 Vm---<    do k=1,nsp
12 Vm M-<      do js=1,jspins
12 Vm M          fxc(k,js)= vx(js,k)+vc(js,k)
12 Vm M->      enddo
12 Vm--->    enddo
```

As noted earlier, when optimizing an X1 code, it's always gratifying to see "MV" in the loopmark listing, as this indicates that the given loop was unconditionally streamed and vectorized. The real benefit from this change, however, was inside the pbe subroutine, not in the setup shown above.

As mentioned earlier, pbe itself is reasonably long. Each call to the original version performed a long series of scalar calculations using many temporary variables. As a measure of the work in pbe and its inlined subroutines, the Fortran source contains 757 add/subtract operators, 465 multiply, 101 divide, and 40 exponent operators. Pushing the loop into the subroutine, I split it into three smaller loops, but given the complexity, anticipated that additional loop splitting, or even pushing the loop down another level into the inlined routines, would be required for it to vectorize. The `ftn` compiler rose to the challenge, though, and with no further assistance on my part, streamed and vectorized everything in pbe.

Here's a snippet of a loopmark listing from inside the original version of the pbe subroutine. It shows the most important inlined call, at line 121:

```
116.                   if(jspins.eq.1) then
117.                     ww=0.0
118.                   else
119.                     ww=(agr(1)**2-agr(2)**2-zet*agr(3)**2)/(rho*rho*twoksg**2)
120.                   endif
121.  I I I I-<>       call CORPBE(RS,ZET,T,UU,VV,WW,1,lpot,ec,vcup,vcdn,
122.                  1                   H,DVCUP,DVCDN)
123.                   eclsd=ec
```

Here's the equivalent section from the vectorized version:

```
147.  MV                  if(jspins.eq.1) then
148.  MV                    ww=0.0
149.  MV                  else
150.  MV                    ww=(agr(1,K)**2-agr(2,K)**2-
151.  MV             &          zet*agr(3,K)**2)/(rho(K)*rho(K)*twoksg**2)
152.  MV                  endif
153.  MV I                call CORPBE(RS,ZET,T,UU,VV,WW,1,lpot,ec,vcup,vcdn,
```

```
154.  MV                1                       H,DVCUP,DVCDN)
```

Unfortunately, the benchmark I used to profile PFLAPW was weighted heavily toward a specific phase of the computation, and although the optimization of pbe improved its performance by 20%, the improvement for the user's benchmark, shown in the following plot, is less than 10%.

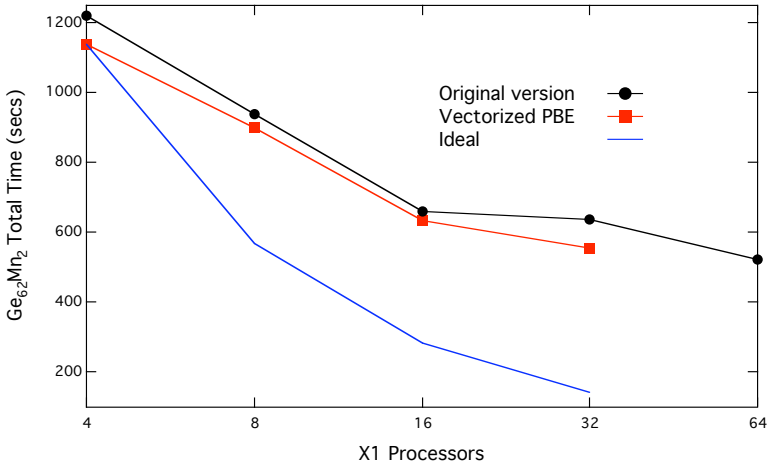

Figure 2: PFLAPW total run-time

Further work on PFLAPW will include profiling with a larger benchmark, and addressing the scaling issue. The vector version of PBE and its calling routines swapped into the serial version of the code, FLAPW, without any modifications.

Gasoline

This is an n-body galaxy model, which has been ported to several MPP systems and logged significant time on ARSC's T3E. It has not been previously run on vector systems.

Program	**gasoline**
Language	C
Explicit Parallelism	MPI / SHMEM
Lines of code	~38,000

table 8: Basic attributes of gasoline

The end user ported gasoline without difficulty to the X1, using only one platform specific include file (used on the T3E) and with expected X1 modifications to the cc command line options.

none

table 9: Gasoline Porting issues

Initial performance was discouraging, but two simple optimizations gave significant improvement. Considerable effort has gone into Gasoline to create a load-balanced, efficient parallel MPI application, and the user found that SSP mode is very nearly four times faster than MSP mode. Perhaps it's a stretch, but I've called this the first "optimization" in the following table:

SSP Mode	
Vectorization/Streaming	Collapse loops, increase vector length
	Eliminate recursive call
	Inline erf/erfc

table 10: Gasoline Optimizations

Given the speedup provided by switching to SSP mode, it became interesting to seek additional optimizations. Timers embedded in the code identified the following loop nest, as shown by the C compiler loopmark listing output, as an unexpectedly slow section:

```
54.  1 2------<       for (ix=-nEwReps;ix<=nEwReps;++ix) {
55.  1 2                      bInHolex = (ix >= -nReps && ix <= nReps);
```

```
 56.  1 2                    dxo = dx + ix*L;
 57.  1 2 3----<             for(iy=-nEwReps;iy<=nEwReps;++iy) {
 58.  1 2 3                          bInHolexy = (bInHolex && iy >= -nReps && iy <= nReps);
 59.  1 2 3                          dyo = dy + iy*L;
 60.  1 2 3 4--<                     for(iz=-nEwReps;iz<=nEwReps;++iz) {
 61.  1 2 3 4                                bInHole = (bInHolexy && iz >= -nReps && iz <= nReps);
 […]
100.  1 2 3 4                                         if (bInHole) gam[0] = -erf(alpha*r);
101.  1 2 3 4                                         else gam[0] = erfc(alpha*r);
```

Vectorization of the innermost loop was inhibited by the `erf` and `erfc` function calls at lines 100 and 101. Automatic inlining to eliminate the function calls was slightly more problematical that usual. I replaced the LibSci version with source code obtained, simply enough, from gasoline distribution itself. In this coding for `erfc`, however, it makes a recursive call, which renders inlining is impossible. Fortunately, the recursion in `erfc` is guaranteed to halt at a depth of two, so it was possible to have `erfc` call a renamed copy of itself ("erfc_inner"). Unlike `ftn`, which offers the inlinefrom compiler option, `cc` doesn't allow inlining between source files. It was thus necessary to include these routines in the same file. With these changes, `cc` defied the odds, inlined this three deep call tree, and vectorized the inner loop, shown above at line 60.

Initial vectorization improved performance only negligibly. This was no surprise, knowing that the value of `nEwReps` in the benchmark case, and typical production runs, is three, and thus that the vector length of the inner loop is only seven. The three loops are not perfectly nested, so the compiler was unable to collapse them automatically. By manually collapsing them, the vector length was increased to 7**3, or 343, performance of this subroutine was improved by a factor of about three, and performance of the overall code was improved by 50%. The following snippet of code shows this modification:

```
241.  1                      ixStride = (2*nEwReps + 1)*(2*nEwReps + 1);
242.  1                      iyStride = 2*nEwReps + 1;
243.  1 V----<               for(tmp_n = 0; tmp_n < ixStride*iyStride; tmp_n++) {
244.  1 V                            ix = tmp_n/ixStride - nEwReps;
245.  1 V                            iy = (tmp_n%ixStride)/iyStride - nEwReps;
246.  1 V                            iz = tmp_n%iyStride - nEwReps;
247.  1 V                            bInHolex = (ix >= -nReps && ix <= nReps);
248.  1 V                            dxo = dx + ix*L;
249.  1 V                                   bInHolexy = (bInHolex && iy >= -nReps && iy <= nReps);
250.  1 V                                   dyo = dy + iy*L;
251.  1 V
```

Gasoline has an internal method for estimating performance. The computation rate in GFLOPS is derived from the number of operations performed in a significant computational kernel divided by the time spent in that kernel added to the time spent perrforming related non-computational communication and other bookkeeping tasks. According to this method, the optimized code runs on the X1 at 4.0 GFLOPS on 16 SSPs, or 7.8% of peak theoretical performance. On the T3E it runs at 4.6 GFLOPS on 64 PEs, or 7.9% of peak. The researcher is using this X1 port to migrate his work from the T3E to the X1.

Rosetta

This is a ab initio protein structure prediction code which has been ported to a wide variety of platforms. It has the basic code attributes as shown in table11.

Program	**rosetta**
Language	Fortran
Explicit Parallelism	none
Lines of code	43,300

table 11: Basic attributes of rosetta

Porting rosetta to the X1 was straightforward. Compiling with `-e0` to initialize local stack variables eliminated a floating point error. Only one code modification was required. A call to `SYSTEM` failed because, on the X1, this utility is not available via a Fortran API. I provided a C interface to the system call and a Fortran interface to the C interface at which point the code compiled and ran correctly.

Missing SYSTEM utility call interface

table 12: Rosetta Porting issues

Initial performance comparisons between runs on the X1 and ARSC's IBM p690 regatta server were discouraging. The benchmark provided by the user completed in 74 seconds on the p690 but 30 minutes on the X1. A profile using `cray_pat` revealed that about 45% of the time was spent in a function, `count_pair_position` which is called at four different locations. An example is shown at line 369 in the following snippet of code:

```
358.  1 2 3-----------------<          do jatom=1,5
359.  1 2 3                              type1=Eatom_type(jatom,j)
360.  1 2 3                              xpos1 = Eposition(1,jatom,j)
361.  1 2 3                              ypos1 = Eposition(2,jatom,j)
362.  1 2 3                              zpos1 = Eposition(3,jatom,j)
363.  1 2 3
364.  1 2 3                              if (type1.ne.0) then
367.  1 2 3 4--------------<               do iatom=1,5 !all backbone atoms in residue
368.  1 2 3 4                                type2=Eatom_type(iatom,i)
369.  1 2 3 4 V I I I I I-<>                 if (count_pair_position(i,type2,res(i),
370.  1 2 3 4                     #                        j,type1,res(j))) then
```

The `count_pair_position` function is inherently scalar. For a single pair of atoms in the protein, it tests several properties to assess whether the pair should be included in the scoring of the current folding of the protein. It and the functions inlined below it, evaluate over 20 conditional statements while performing little computation. One loop in the inlined call tree does vectorize, as shown by the "V" on line 369, but it is a short search loop. The function call itself is included in a pair of 5 iteration, nested loops which would provide scant work, even if they could be vectorized. Improving the performance of this code on the X1 would require a major rewrite. Given that it ran initially on the p690 at about 25x the X1 speed, it seemed unnecessary to pursue X1 optimization further.

FREEH and DYSON

These first of these codes is used for exploring the superconducting properties of ferromagnetic materials, the second for electrodynamics studies of photonic crystals.

Program	**freeh**	**dyson**
Language	Fortran 95	Fortran
Explicit Parallelism	none	none
Lines of code	255	290

table 13: Basic attributes of freeh and dyson

The users of these short, serial codes successfully ported them to the X1 without assistance.

none

table 14: freeh and dyson porting issues

A `cray_pat` profile of freeh revealed that over 95% of its time is spent in the LAPACK eigenvalue/eienvector routine, `SSPEV`, which the code links from Cray's scientific libraries. This routine is called in an iterative loop against a small, 100x100 array. I rejected the idea of optimizing the code by replacing the LAPACK library call it with a parallel equivalent from the ScaLAPACK library. The ScaLAPACK user guide recommends minimum array dimensions of 1000x1000 per processor, after data decomposition. Freeh would have been starting with an array of only 100x100. Given that the vast majority of the work is in this single LAPACK call, I didn't pursue other optimizations. The performance of freeh, as reported by the user of the code, is approximately 1.3x faster on the IBM p690.

Optimizing dyson was another story. Modest changes took this code from approximately 98% scalar to 99.8% vector.

Vectorization/Streaming	Eliminate conditionals in loops
	Eliminate dependency

table 15: Dyson Optimizations

From a simple loopmark listing of this short code, it was clear that the following loop dominated total execution time. This loop updates all elements of the array "g" (see lines 193 and 196), except for one row and column, with a dependency on that row and column:

```
191.  1 Vp----<   do i = 1, nxy ; do j = 1, n_epsilon
192.  1 Vp          if(i /= km(order) .and. j /= order) then
193.  1 Vp              g(i, j) = g(i, j) + wcda*g(i, order)*eps(order)*g(km(order), j)
```

```
  194.  1 Vp          endif
  195.  1 Vp          if(i == km(order) .and. j == order) then
  196.  1 Vp              g(i, j) = (z1/wcda)/(1. - z1*eps(order))
  197.  1 Vp          endif
  198.  1 Vp---->   enddo ; enddo

ftn-6213 ftn: VECTOR File = dy.f90, Line = 191
  A loop starting at line 191 was conditionally vectorized.

ftn-6751 ftn: STREAM File = dy.f90, Line = 191
  A loop starting at line 191 was not multi-streamed because a recurrence was
  found on "G" at line 193.
```

The loopmark listing symbol "Vp," indicating conditional vectorization, is a bit misleading as it implies some benefit from vectorization. `pat_hwpc` reports that this original version of the code (running a small benchmark) computes only 2% of its total operations in vector mode, and that its overall computation rate is 37 MFLOPS. A vector instruction may have been issued, but this is certainly not vector performance.

The approach I used to optimize the loop was to note that the first conditional statement in the loop blocks updates to the row and column on which all other updates are based. Thus, it was possible to save the particular column and row to temporary arrays which I then used to eliminate the loop dependencies and, by copying the arrays back on exit from the loop, to eliminated the conditionals. "M" and "V" are our friends, and they're out in force in the vectorized version:

```
  206. 1 Vw V M-<>  G_KM_ORDER_J(:) = g(km(order),:)  ! Save
  207. 1 V M----<>  G_I_ORDER(:) = g(:,order)         ! Save
  208. 1
  209. 1            ! Almost all the work in the program takes place in this loop.
  210. 1 Vm------<  do i = 1, nxy
  211. 1 Vm Mr---<    do j = 1, n_epsilon
  212. 1 Vm Mr          g(i, j) = g(i, j) + wcda*G_I_ORDER(i)*eps(order)*G_KM_ORDER_J(j)
  213. 1 Vm Mr--->    enddo
  214. 1 Vm------>  enddo
  215. 1
  216. 1 V M----<>  g(:, order) = G_I_ORDER(:)         ! Restore
  217. 1 Vw V M-<>  g(km(order), :) = G_KM_ORDER_J(:)  ! Restore
  218. 1
  219. 1            !
  220. 1            ! This completely replaces the second "if" block.
  221. 1            !
  222. 1            i = km(order)
  223. 1            j = order
  224. 1            g(i, j) = (z1/wcda)/(1. - z1*eps(order))
```

The `pat_hwpc` tool reports that this vector version runs at 3820 MFLOPS or about 30% of peak theoretical performance. Switching the longer, `nxy` loop to the inside using the `INTERCHANGE` compiler directive, as shown in the following snippet, actually degraded performance to 3378 MFLOPS.

```
  210.  1            !DIR$ INTERCHANGE (j,i)
  211.  1 iV-----<   do i = 1, nxy
  212.  1 iV iM--<     do j = 1, n_epsilon
  213.  1 iV iM          g(i, j) = g(i, j) + wcda*G_I_ORDER(i)*eps(order)*G_KM_ORDER_J(j)
  214.  1 iV iM-->     enddo
  215.  1 iV----->   enddo
```

However, the compiler treated manually interchanged loops differently, unrolling the shorter, `n_epsilon` loop four times, as shown here. This provided the best performance for this code at 4223 MFLOPS or 35% of peak:

```
  211.  1 Mr-----<   do j = 1, n_epsilon
  212.  1 Mr V---<     do i = 1, nxy
  213.  1 Mr V           g(i, j) = g(i, j) + wcda*G_I_ORDER(i)*eps(order)*G_KM_ORDER_J(j)
  214.  1 Mr V--->     enddo
  215.  1 Mr----->   enddo
```

Total wall-clock time for the different versions, running a production problem, compared against times on an IBM P655+ processor and SX-6 CPU are given in the following table.

Dyson Version	**ARSC X1**	**SX-6**	**ARSC p655+**
original	43501	21773	833
vectorized	161	329	822

table 16: Dyson timings (seconds)

The original version was too slow to be useful on the X1 or SX-6, but gave good performance on the IBM. This serves as a reminder that the vector systems are not general purpose, but rather, specialized for vector codes. The comparison between the X1 and SX-6 reflects the SX-6's better scalar performace and the X1's better vector performance. That a speedup of 270x was realized on the X1 with such a minor change demonstrates the importance of performance analysis and optimization for all codes running on the X1.

Sumary Comments

Ten months of experience on the Cray X1 has revealed some common themes.

1) The most predictable porting problems have come from data size and file I/O issues. When approaching a new code, these are the first issue to consider.
2) As the X1 libraries, tools, and compilers have matured, some problems have been solved by either bug fixes or addition of features. This trend is likely to taper off over time as fewer problems remain.
3) The Cray compilers vectorize and stream surprisingly large and complicated blocks of code. It is interesting to compare their handling of various loops with the SX-6 vectorizing compilers, and to realize that each compiler has its strengths. One can't assume that loops which vectorize on one system will necessarily do so on the other.
4) SSP mode significantly outperforms MSP mode for at least one MPI code in serious production at ARSC. Every code should probably be tested in both modes.
5) Loopmark listing for C codes has arrived!
6) Here are two lessons to learn when approaching a new code. First, communicate frequently with the user to avoid making incorrect assumptions and repeating the mistakes of others, and second, be certain to obtain and use benchmarks or test cases which are representative of actual production work. Optimizing the wrong bottleneck is a waste of time.
7) The X1 is not a general purpose machine, it is specialized for codes with good vector characteristics. The right codes will achieve extraordinary performance on the X1.
8) Every code ported to the X1, even if the port goes smoothly, should be evaluated for vector and streaming efficiency by someone with specific X1 knowledge and experience. Of the user codes mentioned in this paper, three benefited significantly by minor non-structural, non-algorithmic modifications. In fact, the X1 would have been rejected for production runs of dyson and gasoline had these codes not been analysed and tuned.

Acknowledgements

Sam Cable, Ilya Grinberg, Andrew Rappe, Joo-Hyoung Lee, John Elson, Klaus Halterman, Thomas Quinn, Richard Bonneau, John Levesque, and Ed Kornkven. This work was supported in part by a grant of HPC resources from the Arctic Region Supercomputing Center at the University of Alaska Fairbanks as part of the Department of Defense High Performance Computing Modernization Program.

The Author

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