

Understanding the Cray X1 System

Samson Cheung, Ph.D.
Halcyon Systems Inc.
NASA Advanced Supercomputing Division
NASA Ames Research Center
cheung@nas.nasa.gov

Abstract

This paper helps the reader understand the characteristics of the Cray X1 vector supercomputer system, and provides hints and information to enable the reader to port codes to the system. It provides a comparison between the basic performance of the X1 platform and other platforms available at NASA Ames Research Center. A set of codes, solving the Laplacian equation with different parallel paradigms, is used to understand some features of the X1 compiler. An example code from the NAS Parallel Benchmarks is used to demonstrate performance optimization on the X1 platform.*

Key Words: Cray X1, High Performance Computing, MPI, OpenMP

Introduction

The Cray vector supercomputer, which dominated in the '80s and early '90s, was replaced by clusters of shared-memory processors (SMPs) in the mid-'90s. Universities across the U.S. use commodity components to build Beowulf systems to achieve high peak speed [1]. The supercomputer center at NASA Ames Research Center had over 2,000 CPUs of SGI Origin ccNUMA machines in 2003. This technology path provides excellent price/performance ratio; however, many application software programs (e.g., Earth Science, validated vectorized CFD codes) sustain only a small fraction of the peak without a major rewrite of the code.

The Japanese Earth Simulator demonstrated the low price/performance ratio by using vector processors connected to high-bandwidth memory and high-performance networking [2]. Several scientific applications sustain a large fraction of its 40 TFLOPS/sec peak performance.

The Cray X1 is a scalable vector system [3]; characterized by high-speed custom vector processors, high-memory bandwidth, and a high-bandwidth interconnect linking the nodes. The efficiency of the processors in the Cray X1 is anticipated to be comparable to the efficiency of the NEC SX-6 processors in the Earth Simulator on many computational science applications.

The Department of Energy recently announced (May 12, 2004) an award of \$25 Million to the Oak Ridge National Lab to lead the effort to install the world's largest computer using X1 technology. [4]

This paper will focus on the system characteristics and application optimization techniques. The hardware will be briefly described, followed by a comparison of the NAS Parallel Benchmarks performance with that of other SGI machines. Finally, it will demonstrate some simple techniques for optimizing the performance of a specific NAS Parallel Benchmark.

* This study is sponsored by NASA Ames Research Center under AMTI Subcontract SK-04N-02.

Cray X1 Hardware Description

The Cray X1 design is the incorporation of the previous Cray parallel vector processing (PVP) systems (such as SV1) and massively parallel-processing (MPP) systems (such as T3E). The Cray X1 is hierarchical in processor, memory, and network design. The basic building block is the multi-streaming processor (MSP), which is capable of 12.8 Gflops for 64-bit operations. Housed in each MSP are two MB of cache shared by four single-streaming processors (SSPs). Each SSP is capable of 3.2 Gflops for 64-bit operations, each with 32-stage, 64-bit floating point vector unit and a two-way super-scalar unit.

The concept of "CPU" is not very clear. From a programmer's point of view, if the application is compiled with the "-Ossp" flag option, each SSP can be viewed as a single CPU; otherwise, an MSP is viewed as a single CPU. Since the X1 processor was developed to be comparable to the NEC SX-6 processor, which has eight vector pipes, one MSP is normally viewed as a CPU.

Four MSPs form a Cray X1 node. Within a node, there are 16 GB (or 32 GB) of flat, shared memory. There are three forms of cache in a Cray X1 system, namely, D-cache, I-cache, and E-cache. Each SSP has a 16 KB scalar data cache (D-cache) and a 16 KB instruction cache (I-cache); they are two-way set associative. Each MSP has a two MB E-cache shared by the four SSPs.

Table 1: Basic Performance Comparison With Other Platforms

Machines	SGI O3K	CPQ SC45	SGI Altix	Cray X1	Linux
Hardware					
Chip	MIPS R14000	Alpha-EV68	Itanium 2	Cray X1	Pentium 4 Xeons
CPU/MHz	512, 600MHz	1392, 1GHz	512, 1.5GHz	512 SSP, 800MHz	512, 2.4GHz
CPU/node	4	4	2	4 MSPs (16 SSPs)	2
1st cache	32 KB	64 KB	32 KB	2MB cache in MSP	Trace cache
2nd cache	8 MB	8 MB	256 KB		512KB
3rd cache	N/A	N/A	6 MB		
Memory/node	1 GB	2 GB	3.8 GB	16GB	1 GB DDR
Communication					
Bandwidth (MB/sec)	229.740	772.135	1878.416	12049.934	154.802 / 229.991
Latency (µsec)	3.3	6.1	1.6	9.2 (MSP) 5.1 (SSP)	17.3 / 12.1
MPI_ALLtoALL	MB/sec/proc	MB/sec/proc	MB/sec/proc	MB/sec/proc	MB/sec/proc
2 CPUs	116.995	276.731	874.000	3025.179	63.565
8 CPUs	105.114	56.148	223.642	2629.884	63.805
16 CPUs	97.540	33.380	131.710	2188.945	60.481
32 CPUs	81.434	28.525	95.533	1499.350	51.139
64 CPUs	64.540	24.211	55.314	803.575	46.140

Table 1 gives comparison of some platforms with X1. The platforms compared were, at one time, available at NASA Ames Research Center where a performance comparison was completed in March

2004. It is observed that although the MPI bandwidth of X1 is very high, the latency is very low (9.2 msec.) across MSPs. The latency is a bit faster (5.1 msec.) across SSPs within an MSP. Nevertheless, if an application is latency-bounded, the performance on the X1 would be penalized. According to Cray engineers, the later version of X1 will be improved.

MSP and SSP

There are two different modes of compiling and running a software application on X1: namely, the MSP mode and the SSP mode. The default is the MSP mode. To use the SSP mode, one has to compile with the flag "-Ossp" (Fortran).

The concept of multi-streaming can be illustrated in the following do-loop:

```
DO I = 1,2000
  A(I) = A(I) * pi
ENDDO
```

Each MSP has four SSP processors. Processor SSP0 works on I=1501-2000, SSP1 works on I=1001-1500, SSP2 works on I=501-1000, and SSP3 works on I=1-500. This do-loop is also called a streaming region. Only one of the four SSP processors is used outside of the streaming regions. So, streaming is compiler auto-parallelization of eligible loops, with the work divided between the four SSPs per MSP. If multiple loops exist, the compiler will automatically apply the streaming to an outer loop when feasible. This is identical to the PARALLEL DO directive in OpenMP. In the MSP mode, the synchronization between SSPs is done on hardware with lower overhead than using the OpenMP directive.

When an application has difficulty scaling to a large number of processors, using the MSP as the principal MPI process provides a more powerful processing node. This may allow scaling to a higher sustained performance level with fewer computational units.

In order to determine if the compiler multi-streams a do-loop, one can include the compiler flag "-rm" for loop marking. After compilation, a text file (loop-mark report) is created. For example,

```
78.  1 M-----<      do j= jst, jend
79.  1 M V---<        do i= istart, iend
80.  1 M V              dumax= max( dumax, abs( du( i, j)))
81.  1 M V              u( i, j)= u( i, j)+ du( i, j)
82.  1 M V--->        enddo
83.  1 M----->        enddo
```

The "M" stands for multi-stream, and the "V" stands for vectorization. The loop-mark report indicates that the compiler performs multi-streaming in the outer loop and vectorization in the inner loop, index "i".

It should be realized that the vectorization happens at the SSP level, whereas the streaming only exists for MSPs. If a code streams well, an MSP is effectively an extremely powerful single processor. If it doesn't stream, then it just wastes three SSPs, and it should be run in SSP mode.

Even running in SSP mode, the code should be vectorized in order to achieve good performance on the X1. As mentioned before, the one X1 node has 16 SSPs; these SSPs share the same 16 GB of memory. If the application does not do well in streaming and OpenMP is desired, one can use a maximum of 16 SSPs for 16 OpenMP processes.

Sample Problem – Laplacian

The physical problem is to solve the Laplace Equation, which is the model equation for diffusive processes such as heat flow and viscous stresses. It makes a good test case for approaches to numerical programming. In two dimensions, the Laplace Equation takes the following forms:

$$\nabla^2 u \equiv \frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} = 0 \quad (1)$$

The set of boundary conditions in this report is shown in the Figure 1. For the sake of simplicity, the initial guess of the solution is $u=0$.

The computational grid is 2,000 by 4,000, with 2,000 grid points in the x -direction and 4,000 points in the y -direction. For parallelization, we use 1-D domain decomposition in the y -axis.

The standard finite-difference explicit approach is used to discretize equation (1).

Equation (1) becomes

$$\nabla^2 u \equiv \frac{u_{i+1,j}^n - 2u_{i,j}^{n+1} + u_{i-1,j}^n}{\Delta x^2} + \frac{u_{i,j+1}^n - 2u_{i,j}^{n+1} + u_{i,j-1}^n}{\Delta y^2} + O(\Delta x^2, \Delta y^2) \quad (2)$$

where $u_{i,j}^n$ is the approximated solution of Eqn. (1), after n iterations, at the i^{th} grid point on the x -axis and the j^{th} grid point on the y -axis. The truncation error, $O(\Delta x^2, \Delta y^2)$, is in the order of the square of the grid size. This error is small if the number of grid points is large. Rearranging Eqn. (2), we have an iterative scheme,

$$u_{i,j}^{n+1} = u_{i,j}^n + \Delta u_{i,j}, \text{ where } \Delta u_{i,j} = \frac{u_{i,j+1}^n + u_{i,j-1}^n + u_{i+1,j}^n + u_{i-1,j}^n}{4} - u_{i,j}^n \quad (3)$$

The initial guess, $u_{i,j}^0$, of the solution is set to be zero.

The same iterative scheme in Eqn. (3) is solved by five different computational regimes. Two of these are for serial implementation, and the rest are for parallel implementation.

There are two possible serial approaches in implementing the finite-difference solution to the Laplace Equation described above.

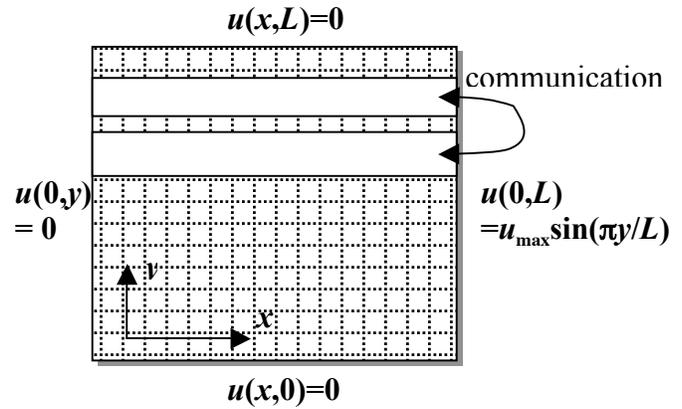


Figure 1. Computational grid and the boundary conditions

The Cache-friendly Approach: `cache.f90`

The $\Delta u_{i,j}$ calculations, the Δu_{\max} comparisons, and the $u_{i,j}$ updates are performed simultaneously. This approach performs well on cache-based microprocessor architectures because it tends to re-use cache. The main do-loop in an iteration process is given below. It is noticed that we jam all the calculations into the inner do-loop.

```

do j= 2, jml
do i= 2, iml
  du( i, j)= 0.25*( uo( i- 1, j)+ uo( i+ 1, j)+    &
                uo( i, j- 1)+ uo( i, j+ 1))- uo( i, j)
  dumax= max( dumax, abs( du( i, j)))
  u( i, j)= uo( i, j)+ du( i, j)
enddo
enddo

```

The Vectorized Approach: `vect.f90`

All $\Delta u_{i,j}$'s are computed, then a Δu_{\max} is found before all $u_{i,j}$'s are updated. This approach should perform well on vector-based architectures, such as the Cray X1 and T90 series. The main do-loops in the iteration process are given below.

```

do j= 2, jml
do i= 2, iml
  du( i, j)= 0.25*( uo( i- 1, j)+ uo( i+ 1, j)+    &
                uo( i, j- 1)+ uo( i, j+ 1))- uo( i, j)
enddo
enddo

do j= 2, jml
do i= 2, iml
  dumax= max( dumax, abs( du( i, j)))
  u( i, j)= uo( i, j)+ du( i, j)
enddo
enddo

```

The Shared Memory Paradigm: `omp.f90`

We parallelize the cache-friendly code (`cache.f90`) that uses OpenMP for multi-threading. The j-loop comprises the bulk of the work in the program and, clearly, should be parallelized. The main do-loop in the iteration process is given below.

```

!$OMP DO REDUCTION( max: dumax)
do j= 2, jml
do i= 2,iml
  du( i, j)= 0.25*( uo( i- 1, j)+ uo( i+ 1, j)+ &
    uo( i, j- 1)+ uo( i, j+ 1))- uo( i, j)
  dumax= max( dumax, abs( du( i, j)))
  u( i, j)= uo( i, j)+ du( i, j)
enddo
enddo
!$OMP END DO

```

The Distributed Memory Paradigm: [mpi.f90](#)

We can divide the grid into segments and assign an MPI process to each segment. Each segment also needs to maintain “ghost cells,” which contain solution values at points on the boundaries of neighboring processes; the ghost cells are kept up-to-date by passing messages between processes containing the boundary values.

```

do j= jst, jend
do i= istart, iend
  du( i, j)= 0.25*( uo( i- 1, j)+ uo( i+ 1, j)+ &
    uo( i, j- 1)+ uo( i, j+ 1))- uo( i, j)
  dumax= max( dumax, abs( du( i, j)))
  u( i, j)= uo( i, j)+ du( i, j)
enddo
enddo

! Compute the overall residual
call MPI_REDUCE( dumax, gdumax, 1, MPI_REAL8, MPI_MAX, 0 &
  ,MPI_COMM_WORLD, ierr)

! sending and receiving boundary data (MPI_send/MPI_Receivev...

```

Vectorization

It is a good exercise to compare the performance of the vector code and cache-friendly code on various machines. The following table (Table 2) shows the performance of the vect.90 and cache.f90 codes of the Laplacian solver on three different platforms.

Table 2: Vector code vs. Cache-Friendly code

Machines	SGI O2K	SGI Altix	Cray X1
Compilation	-O3 -r8 -64 -mips4 -r10000	-O3 -r8 -tpp2	-Vaxlib
Compiler	Ver 7.3.1.3m	Ver 8.0	Ver 5.1.0.3
vect.f90	22.8523922 sec.	4.010742 sec.	1.27633 sec.
cache.f90	12.4084873 sec.	5.435039 sec.	64.8442 sec.

It is noted that the vector code is 50 times faster than the cache-friendly code on X1. Thus, vectorization is the key on this platform.

OpenMP Mode

It is interesting to determine the performance of the OpenMP code on the X1 using different (MSP, SSP) modes. The shared memory arena is only available within one node; thus, one cannot use OpenMP across two nodes. The maximum number of OpenMP processors available for use in the MSP mode is four; in the SSP mode it is 16.

Table 3: MSP mode vs. SSP mode on OMP code

Machines	O2K	Altix	X1	
omp.f90			MSP	SSP
2 OMP Proc	58.8336296	22.2773	3.52	9.63
4 OMP Proc	31.7845402	11.1982	1.97	5.43

The MSP mode is faster than the SSP mode because streaming occurs in the outer loop in addition to the OpenMP. In the case of the 4-OMP processor run, there are effectively 16 SSPs working on the loop.

It is difficult to conclude which mode is more efficient than the other. General belief is that streaming is more efficient (less overhead) than OpenMP; therefore one would use the MSP mode rather than OpenMP. However, from the above example, the 4-OMP run in MSP mode does not seem very efficient in utilizing the processors. When considering the scaling of the Origin 2000 and Altix platforms, our simple OpenMP case has almost perfect scaling (factor of two), but the X1 does not. It is believed that the implementation of OpenMP on the X1 leaves room for improvement.

MPI Mode

Suppose an application requires 16 MPI processes. There are two configurations to consider. One can either use 16 SSPs (4 MSPs) or just use 16 MSPs. Unfortunately, there is no rule of thumb to justify what configuration to use. One has to test it both ways. However, it makes sense to use the MSP mode if the code streams well and if MPI communication is a bottleneck.

The following is a demonstration of the performance differences in MSP and SSP modes for the mpi.f90 code. Using the Fortran compiler flag "-rm" to create a loop-mark report (see Figure 2). The outer loop at line 72, represented by "M," is being streamed in the MSP mode; and vectorization (indicated by "V") is performed in the inner loop for both cases. The letter "r" represents unrolling at the compiler level.

The two reports in Figure 2 are identical except the MSP report has multi-streaming, represented by, "M." Our test case is a very nice multi-stream code. On the other hand, it is also very efficient in the MPI communication paradigm using high-bandwidth unblocked MPI send and receive. Figure 3 compares the performance from the MPI code using MSP and SSP modes. The horizontal axis indicates the number of PE (MPI processes). Using 2 PEs in MSP mode would require 8 SSP processors; whereas, using 2 PEs in SSP mode requires 2 SSPs.

Therefore, the 2-PE run in MSP mode (diamond line) is using the same amount (8) of SSPs as the 8-PE run in the SSP mode (square line). It is shown that these two runs have the same performance in

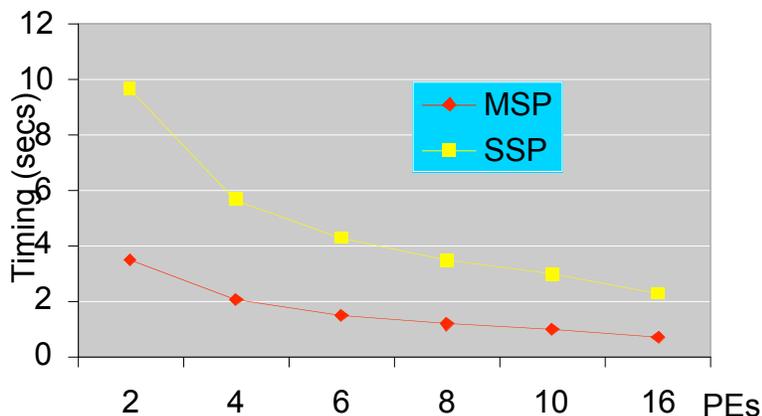


Figure 3. MSP and SSP performance on MPI Laplacian solver.

If one's MPI codes rely on the system for internal buffering of messages, one has to turn it on. This is turned off by default on the X1. To turn it on and/or increase buffer size, one can set the following environment variables:

```
export MPI_BUFFER=1
export MPI_BUFFER_MAX=1000000 (default 0 bytes)
```

Another variable that may be of interest when sending long messages is `MPI_BUFS_PER_PROC`. The default is 16 or 32 pages (1 page = 16 KB).

Data Size Pitfalls

One topic worth discussing in code porting to the X1 is data size. The X1 system is, by default, an IEEE 32-bit system with compiler options and libraries to permit the use of 64-bit data type. To increase the default data size from 32 bits to 64 bits, one can use the Fortran compiler option `-s default64`. Selectively, one can use `-s integer64` and `-s real64` options to change the default data sizes of integers, logical, or real to 64 bits.

It should be noted that the `REAL (KIND=4)` and `REAL` have the same range and precision when the `-s default32` compiler option is enabled (default). The `REAL (KIND=8)` and `REAL` have the same range and precision when the `-s default64` option is enabled.

For example, if the `-s default64` option is used, variables declared as `DOUBLE PRECISION` will be promoted to 128 bits. However, the LibSci scientific library does not support that (please see Table 4 for argument types and sizes for LibSci on Cray). Normally, one would turn off the double precision by using the compiler option `-dp`. That is, compiling with `-s default64 -dp` will promote `REAL` variables to 64 bits and keep those `REAL (KIND=8)` or `DOUBLE PRECISION` variables to 64 bits.

The X1 compiler links the appropriate libraries according to `-s default32` or `-s default64`. Therefore, the MPI libraries provide support only for codes compiled with the `-s default32` and `-s default64` options. In other words, compiling with `-s real64`, and expecting MPI to recognize that the transfer (MPI send/receive) variables are 64-bit would be wrong!

Using `-s default64` will convert `INTEGER` to `INTEGER (KIND=8)`. This will crash certain system calls that require an `INTEGER*4` argument (such as `FLUSH`). In this case, one has to declare the relevant variables explicitly to ensure that the variables will not be promoted to `(KIND=8)` by accident.

Details of the data type information can be found in the Cray X1 User Environment Differences, S-2310-51 Chapter 4. Libraries Differences. [5]

Table 4: Argument types and sizes for LibSci on Cray

Type Description	32-bit Library (default)		64-bit Library	
	Bits	Fortran Specification	Bits	Fortran Specification
single precision real	32	REAL (KIND=4) , REAL*4	64	REAL (KIND=8) , REAL*8
double precision real	64	REAL (KIND=8) , REAL*8	N/A	N/A
single precision complex	64	COMPLEX (KIND=4) , COMPLEX*8	128	COMPLEX (KIND=8) , COMPLEX*16
double precision complex	128	COMPLEX (KIND=8) , COMPLEX*16	N/A	N/A
integer	32	INTEGER (KIND=4) , INTEGER*4	64	INTEGER (KIND=8) , INTEGER*8

NAS Parallel Benchmark

One of the benchmarks in the NAS Parallel Benchmark 2.4 suite [7] is used to demonstrate performance optimization on the X1. Table 5 shows the timing of a subset of the benchmarks. This version of NAS Parallel Benchmark uses MPI.

The LU benchmark solves a finite difference discretization of the 3-D compressible Navier-Stokes equations through a block-lower-triangular block-upper-triangular approximate factorization of the original difference scheme. The BT benchmark uses an implicit algorithm to compute a finite-difference solution to the 3-D compressible Navier-Stokes equations. The resulting equations are block-tridiagonal (the fourth order dissipation appears only on the right-hand side, so the left-hand side difference stencil has a width of three blocks). The MG benchmark implements a V-cycle multi-grid algorithm to solve the scalar discrete Poisson equation.

The MG benchmark MFLOPS rate in Table 5 is about the same as shown in Dunigan's evaluation [6]. However, the paper did not give the MFLOPS of the other benchmarks. The comparison for all NAS Parallel Benchmarks can be found in [8]. From Table 5, it is seen that X1 does well only with the MG benchmark, the others do not do well at all.

Table 5: NAS Parallel Benchmarks on various platforms

Machines	SGI O2K/O3K		Altix		Cray X1
	400/800 MHz		1.3GHz	1.5GHz	800 MHz
Compilers and Libraries	MIPSpro.7.3.1.1m mpt.1.4.0.0		Intel Fortran Compiler V7.1		Cray Fortran cftn.4.3
Optimization Flags	-Ofast -64		-O3 -w -ip -tpp2 -stack_temps		-O3 -Ossp
Mop/s Total					
bt.B.4	267	466	1532	1688	664
bt.B.16	1134	1992	5506	6018	2565
bt.B.64	7557	12700	24221	31486	10451
lu.B.4	561	879	3837	4641	1531
lu.B.16	2597	3941	14585	16755	5589
lu.B.64	13947	20656	46684	58370	15390
mg.B.4	381	643	2943	3486	7468
mg.B.16	1435	2432	12210	12773	21747
mg.B.64	6689	15644	35688	46083	47382
bt.C.4	220	417	1117	1559	668
bt.C.16	1065	1811	5555	6214	2645
bt.C.64	5954	10240	21657	26970	10090
lu.C.4	542	845	2647	4447	1441
lu.C.16	2107	3297	15374	17034	5281
lu.C.64	9713	14923	52269	59283	20004
mg.C.4	267	522	2420	2969	9141
mg.C.16	1468	2398	10525	13782	31549
mg.C.64	5599	9664	46806	51590	88587

Performance Optimization

The BT Class C, built with MSP mode, will be used as an example for performance optimization. The original performance is about 180 MFLOPS/process using `-O3` compilation flag. From the table above the MFLOPS/process rate of the BT (in SSP mode for a range of numbers of processors) is about 165; so the original BT does not stream well at all!

To understand where the "hot spot" is, we create an instrumented executable (`bt.C.9_inst`) by using "pat_build" after the executable (`bt.C.9`) is built. The instrumented executable will be used and an instrumented file (`*.xf` file) will be created. A report can be created by "pat_report":

```
prompt> pat_build bt.C.9 bt.C.9_inst
prompt> mpirun -np 9 ./bt.C.9_inst
prompt> pat_report bt.C.9_inst+250973pdt.xf
```

The report shows that routine "binvrchs" takes most of the time and it is called by `x_solve_cell`, `y_solve_cell`, and `z_solve_cell`. From the loop-mark (use flag `-rm`) report of the file `x_solve.f`, the area of interest is the loop (line 439) in routine `x_solve_cell`, see Figure 4. The loop-mark report shows

that this loop is not streamed nor vectorized because subroutines `matvec_sub`, `matmul_sub`, and `binvrchs` are in the loop. Thus, we inline the subroutines by adding directive, `!dir$ inline`, before the call of the routines or adding `!dir$ inlinealways binvrchs` inside the routine `binvrchs`. One can always use the compiler to do the inlining by using the flags:

```
-Oinline5 and/or -Oinlinefrom={x_solve.f,lhsx.f}
```

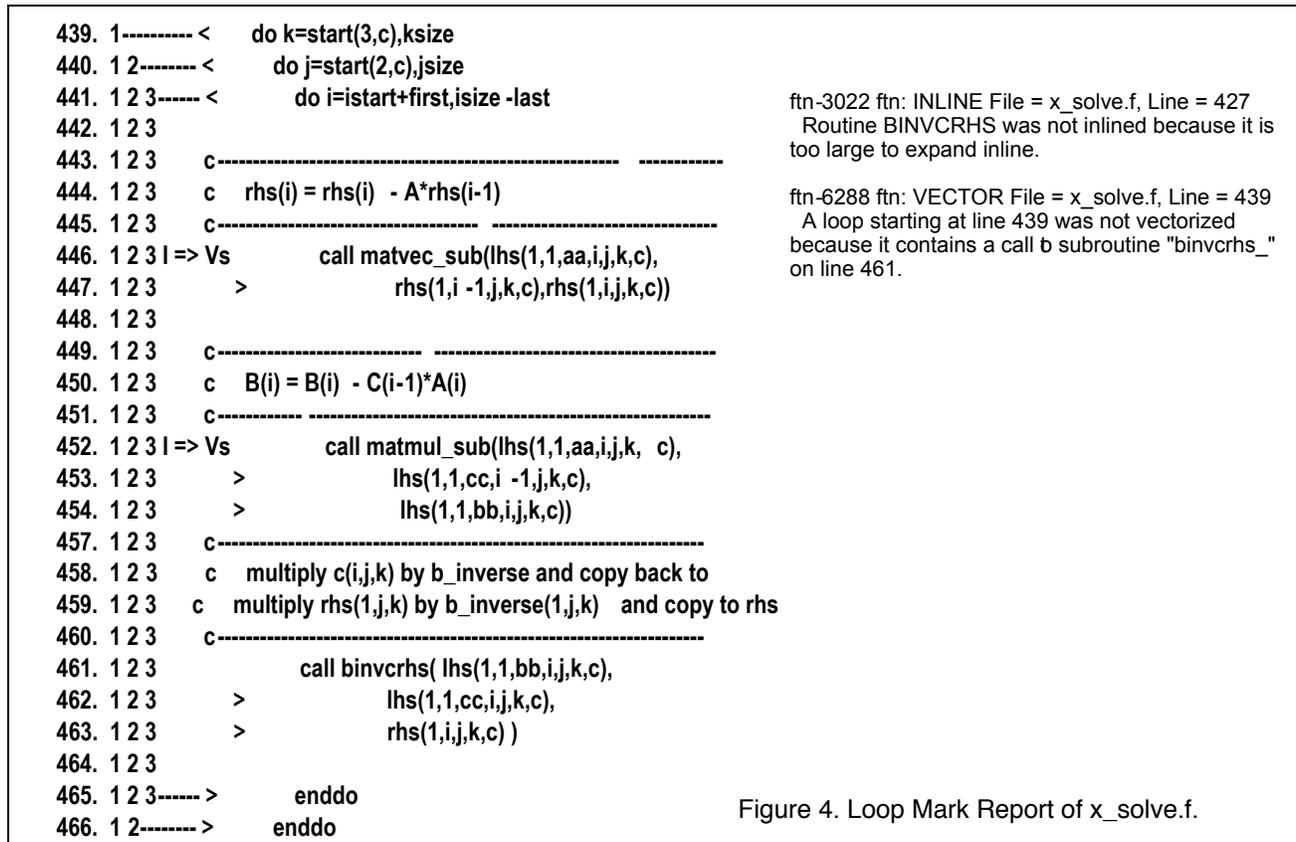


Figure 4. Loop Mark Report of `x_solve.f`.

Since there is dependency in "i" (see line 450 in Fig 4.); we don't get vectorization in "i"; but we can get vectorization in "j". Thus, we move the i-loop outside and force concurrent execution by adding directive `!dir$ concurrent`:

```

do i=istart+first, isize-last
!dir$ concurrent
do k=start(3,c), ksize
!dir$ concurrent
do j=start(2,c), jsize

```

Similar directives should be put in `y_solve_cell` and `z_solve_cell` as well.

The next routines of interest indicated from the `pat_report` are `lhsx.f`, `lhsy.f`, and `lhsz.f`; they are called by the `x_solve.f`, `y_solve.f`, and `z_solve.f`, respectively. From the loop-mark report of `x_solve.f`:

```
ftn-3021 ftn: INLINE File = x_solve.f, Line = 54
Routine LHSX was not inlined because the compiler was unable to locate
the routine to expand it inline.
```

So, we can inline the routine by putting the directive "`!dir$ inlinealways lhsx`" inside `lhsx.f`. Similarly, we can inline `lhsy.f` and `lhsz.f`. However, the loop-mark report in `lhsx.f` shows that there is no streaming in one of the loops, the reason is :

```
ftn-6755 ftn: STREAM File = lhsx.f, Line = 24
A loop starting at line 24 was not multi-streamed because a recurrence
was found on "TMP1" between lines 28 and 121.
```

Here are the two occurrences of the variable `tmp1` :

```
24. 1-----<      do k = start(3,c), cell_size(3,c)-end(3,c)-1
25. 1 2-----<      do j = start(2,c), cell_size(2,c)-end(2,c)-1
26. 1 2 Vs--<      do i = start(1,c)-1, isize + 1
27. 1 2 Vs
28. 1 2 Vs          tmp1 = 1.0d+00 / u(1,i,j,k,c)
29. 1 2 Vs          tmp2 = tmp1 * tmp1
30. 1 2 Vs          tmp3 = tmp1 * tmp2
```

and

```
119. 1 2 Vs--<      do i = start(1,c), isize
120. 1 2 Vs
121. 1 2 Vs          tmp1 = dt * tx1
122. 1 2 Vs          tmp2 = dt * tx2
```

Although these two `tmp1`'s are not related, the compiler cannot distinguish if they are recurrent or independent. One can solve this problem by renaming the second `tmp1`. Then, the loop will be streamed. After recompile and rerun the benchmark, the new performance number is 1235.67 MFLOPS/process using the compilation flags,
`-O3 -Oinline5 -Oinlinefrom={x_solve.f, lhsx.f}`.

The performance number indicates that we are heading to the right direction. Repeat the "pat_report" profiling, it is found that the `x_solve.f` is still the most time consuming file. The loop in Fig.4 becomes the loop in Figure 5.

```
439. m-----<      do i=start+first,isize-last
440. m          !dir$ concurrent
441. m M-----<      do k=start(3,c),ksize
442. m M          !dir$ concurrent
443. m M 3-----<      do j=start(2,c),jsize
444. m M 3
...
448. m M 3 I      call matvec_sub(lhs(1,1,aa,i,j,k,c),
449. m M 3      >      rhs(1,i-1,j,k,c),rhs(1,i,j,k,c))
450. m M 3
```

ftn-6294 ftn: VECTOR File = x_solve.f, Line = 443
A loop starting at line 443 was not vectorized
because a better candidate was found at line 448.

Figure 5. New Loop Mark Report of `x_solve.f`. The J-loop is not vectorized.

The loop-mark report indicates that the loop starting at line 443 is not vectorized because a better candidate was found at line 448. It turns out that the compiler vectorizes the loops in routines `matvec_sub` and `matmul_sub`. Those loops, however, is only of size 5:

Acknowledgement:

The author greatly appreciates to Dr. Johnny Chang's (NASA Ames Research Center) help on the BT Benchmark, and the User Support Group at the Artic Region Supercomputer Center (ARSC) for the Cray X1 account.

Reference:

- [1] <http://www.beowulf.org/beowulf/projects.html>
- [2] Gordon Bell prizes at the SC2002 Conference:
http://access.ncsa.uiuc.edu/Releases/02Releases/11.07.02_SC2002_Gor.html
- [3] Cray Inc., <http://www.cray.com/products/systems/x1/>
- [4] http://www.ornl.gov/info/press_releases/get_press_release.cfm?ReleaseNumber=mr20040512-00
- [5] <http://www.cray.com/cgi-bin/swpubs/craydoc30/craydoc.cgi>
- [6] Dunigan T.H. et al. "Early Evaluation of the Cray X1", SuperComputing 2003, Nov. 15-21, 2003, Phoenix, AZ, USA.
- [7] Bailey, D.H. et al. "The NAS Parallel Benchmarks 2.0, Tech. Rep. NAS-95-010, NASA Ames Research Center, Moffett Field, CA, 1995.
- [8] http://www.halcionsystems.com/php/employees/cheung/PROJECTS/NAS_Projects/X1_NPB.html