An Introduction to the Cray X1E

Richard Tran Mills
(with help from Mark Fahey and Trey White)

Scientific Computing Group
National Center for Computational Sciences
Oak Ridge National Laboratory

2006 NCCS Users Meeting
February 14, 2006

This work was performed using resources at the National Center for Computational Sciences at ORNL, which is supported by the U.S. Department of Energy under Contract Number DE-AC05-00OR22725.

Outline

Phoenix is the world's largest X1E system (well, almost)

- 1024 multi-streaming processors
- 2 TB aggregate memory

This talk will introduce Phoenix and briefly discuss:

1. X1E hardware platform
2. Some practicalities:
   - Compiling and linking code
   - Running jobs
   - Debugging
3. Performance analysis and tuning

Part I: X1E hardware platform

- NUMA system consisting of up to 2048 nodes (1024 modules)
- Nodes are logical entities of 4 multi-streaming processors (MSPs)

- Memory shared SMP-style within a node
- Jobs that span nodes behave as on MPP distributed memory system
  - Each node has local memory…
  - …but memory is globally addressable between nodes

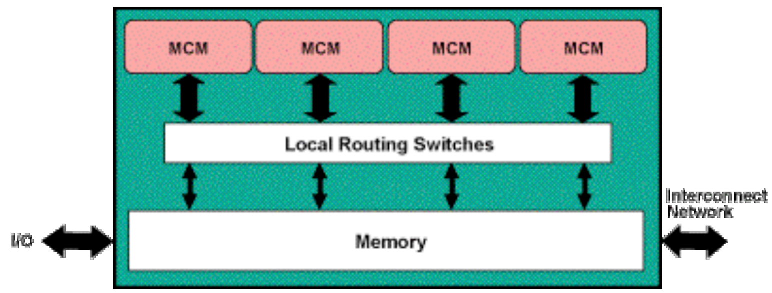
Models of parallelism

The X1E architecture supports several models of parallelism:

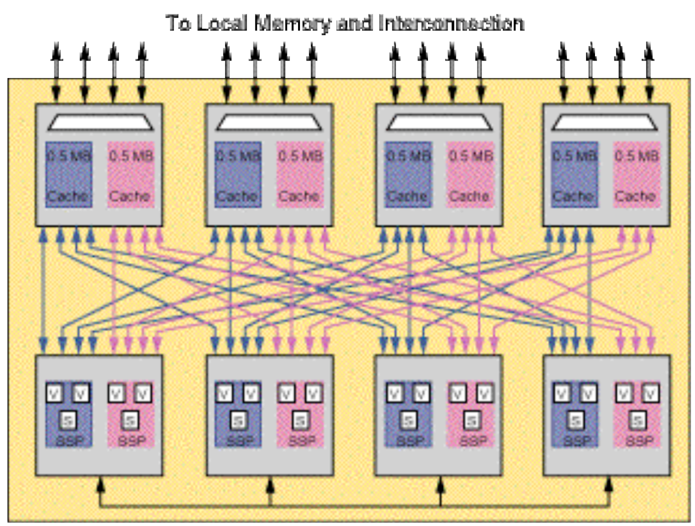
- Two levels of SIMD, loop-level parallelism:
  - Vectorization within SSP
  - Multistreaming within MSP
- OpenMP within node
- Between nodes (or processors)
  - MPI-1 two-sided message passing
  - MPI-2 one-sided communication
  - SHMEM one-sided communication
  - Co-Array Fortran remote memory
  - Direct load/store using pointers

Cray X1E compute module

- Compute module contains 4 multichip modules (MCMs):

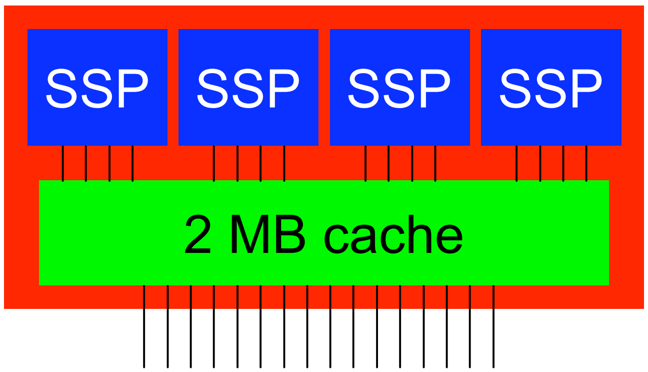


- Each MCM consists of 2 multistreaming processors (MSPs):



Cray X1E multistreaming processor (MSP)

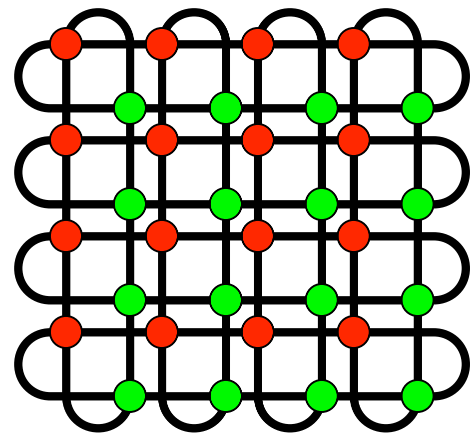
- MSP consists of 4 tightly-coupled single-streaming processors (SSPs)
- Each SSP consists of:
  - One 2-way superscalar processing unit (565 MHz)
  - Two-pipe vector processing unit (1.13 GHz)



Is an MSP one or four processors?

- One!
  - Fast synchronization, shared cache
  - Can be treated as one 8-pipe processor
- Four!
  - Each SSP can operate independently
  - Treat as MPI processor or use OpenMP-like directives

X1E Interconnect



- "Enhanced" 3D torus
  - Modules connected in a 3D torus.
  - One torus dimension is fully connected.
- 12 GB/s measured MPI bandwidth

- Globally addressable memory:
  - Load/store memory on any node
  - Remote memory refs routed through interconnect
  - W/ contiguous nodes, remote address translation possible (System scales w/ number of nodes w/o additional TLB misses)
  - Low latency

Phoenix strengths and weaknesses

- Strengths:
  - Very powerful processors (18 Gflop/sec peak)
  - Low effective latency
    - Vector processors hide local latency
    - Globally addressable memory hides/minimizes global latency
  - Very high memory bandwidth (global and local)
    - Good for stride-1, strided, and random access
- Weaknesses:
  - Scalar processing slow
  - "Some tuning required"
  - Limited memory per MSP

Part II: Practicalities

- Logging into a front end
- Using the Programming Environment
  - Choosing compiler versions
  - Special features of Cray compilers
  - Libraries provided by Cray
- Code-porting issues
- Running jobs
- Debugging

Phoenix front-ends

- Users can ssh into two different front-ends for Phoenix:
  - phoenix.ccs.ornl.gov
    - Compile, load, performance tool commands transparently offloaded to Cray Programming Environment Server (CPES)
    - Some standard tools (e.g., emacs, complete Python) unavailable
  - robin.ccs.ornl.gov
    - Linux cross-compiler box
    - Cross-mounts Phoenix scratch space
    - Can submit and manage Phoenix jobs
    - Much faster than CPES! (5x or more!)

Phoenix front-ends

We recommend working on robin whenever possible.

It is much faster and friendlier!

There are a few cases where you need to use phoenix:

- Using ‘psview’ to display Psched (system scheduler) information
- Using ‘nm’ command to view symbols in binary objects
- autoconf that does not support cross-compilation
  (Note: Python-based configuration needs to be done on robin!)

Cray Programming Environment

Cray Programming Environment provides:

- Fortran compiler 'ftn'
- C compiler 'cc'
- C++ compiler 'CC'
- MPI include files and libraries available by default

- Compiler, MPT versions determined by PrgEnv module version
- `pe-version` tells version of PrgEnv and components
- To load another PE version, do
  `module swap PrgEnv PrgEnv.newversion`
- Best to swap entire PrgEnv, not individual components
  - Possible exception: MPT version

Special Cray compiler flags

Compiler flags are documented completely in man pages.
We outline some Cray-specific ones here.

`-G` options to specify debug level:

- `-g`: Full debugging w/ breakpoints on every executable line.
  - Very slow; all optimizations turned off.
  - Bugs often dissapear!
- `-G1` (ftn) or `-Gp` (CC/cc): block-by-block debugging
  - Multistreaming disabled
- `-G2` (ftn) or `-Gf`: Debugging w/ full optimization
  - In Fortran, only postmortem debugging
  - In C/C++, can set breakpoints at function entry/exit

Special Cray compiler flags

- Executables can be built for three different modes on Phoenix
  - MSP mode (the default)
    - May also want to use `-h gen_private_callee` so that subroutines can be called from multistreamed regions
  - SSP mode (compiler flag: `-h ssp`)
  - "command" mode (compiler flag: `-h command`)
    - Executable can run on service node w/o help from aprun
    - Probably no real need for this now that Robin is available

- Which to use?
  - Lots of loop-level parallelism suggests MSP
  - Very scalable code suggests SSP
- Best to start w/ MSP but try both

Cray LibSci routines

Cray provides a collection of several highly-optimized kernels in LibSci:

Single processor support for:

- Fast Fourier Transform (FFT), convolution, filtering
- BLAS, LAPACK
- Basic Linear Algebra Communication Subprograms (BLACS)
- Sparse direct solvers

- Multiprocessor distributed memory support for
  - FFT routines
  - Scalable LAPACK (ScaLAPACK) routines
  - Basic Linear Algebra Communication Subprograms (BLACS)
- OpenMP versions of all level 3 BLAS and some level 2 BLAS

- Link with `-lsci` normally
  - `-lsci64` of `-sdefault64` routines
  - `-lompsci` for OpenMP support

UT-BATTELLE
Oak Ridge National Laboratory

Common code porting problems

- Beware of old `#ifdef CRAY` directives!
- X1E is very different from old Cray machines:
  - Some library calls may be unsupported, or work differently.
  - Default Fortran data sizes: 32 bit integers, 32 bit reals
  - Use `-sdefault64` to default to 64 bit integers and reals (and to link w/ MPI, BLAS, etc., that assume this)
  - Can also use `-sreal64` to get 32 bit integers, 64 bit reals
- Need to manually check each #ifdef CRAY to see if it makes sense.

- Auto-configuration problems
  - Configure scripts that cannot cross-compile must be run on Phoenix
  - Build systems relying on Python may need to run on Robin

Running jobs

- Phoenix uses PBS for job submission (see http://info.nccs.gov/resources/phoenix/batch)
- Use `aprun` within job script to launch parallel jobs

- Specify number of MSPs in resource list with `mppe=N`. (This also works for SSP jobs -- ask for `mppe=N/4`, `N` the # SSPs)
- Multi-node jobs (>4 MSPs) must request a multiple of 8 MSPs! (Scheduler places jobs on hardware module boundaries)
- Run out of `/tmp/work/$USER` if doing even moderate IO
- Memory limits:
  - Default memory limit is 2 GB per MSP (512 MB per SSP)
  - Request more with `-m` option to `aprun`
  - If requesting more, ask PBS for more MPPE's than aprun will use
  - May also need to increase env variables; see `man 7 memory`

UT-BATTELLE
OAK RIDGE NATIONAL LABORATORY

Debugging: Postmortem

- Set `TRACEBK` to 30 to get automatic traceback when code crashes
- `aprun` needs '`-c core=unlimited`' to generate core files
  DO NOT do this unless running in "`/tmp/work/$USER`"!

- Can view corefiles with gdb or Totalview
  - `phoenix> gdb a.out core`
  - `phoenix> totalview a.out core`

- Traceback gives hints as to what corefiles to look at:

```
Traceback for process 64311(ssp mode) apid 64184.229 on node 7
```

Suggests starting with core file 229.

Debugging: Interactive

- To use Totalview interactively to debug an N process job:
  - `totalview -app "-n N" a.out` [totalview options] [`-a` <program options>]
  - Use `totalviewcli` for command-line
- This can be very useful, but may be slow

- Can also use gdb
  - Fast and responsive
  - Debugging parallel programs difficult (impossible?)
  - Unsupported by Cray

Part III: Performance analysis and tuning

- Always generate performance profile BEFORE code tuning!
  - Routines that are negligible on other systems may be bottlenecks on X1E!

- Basic code tuning steps:
  1. Generate performance profile and identify hotspots.
  2. Examine loopmark listings for hotspots.
  3. Then do some combination of:
     1. Insert compiler directives
     2. Manually unroll loops, switch loop indices, etc.
     3. Rearrange data structures
  4. Return to step 1 and iterate until performance is “good enough”.

Introduction to CrayPAT

Basic steps in using Cray Performance Analysis Toolkit:

1. Generate executable for Phoenix
2. run 'pat_build' to generate instrumented executables
   - `robin> pat_build [options] a.out a.out.inst`
   - Note that object files must be present!
3. submit batch jobs using 'qsub'
   - Run with 'aprun' to generate '.xf' file,
     then run 'pat_report' to generate performance report
   - OR, run with 'pat_run' to run and generate report
     (Provides somewhat simpler interface)
4. Optionally, use Cray Apprentice ('app2') to visualize performance

Types of performance experiments

- Three basic types of performance experiments
  - "Profiling"
    - Simplest experiment; lowest overhead
    - Samples program counter by user and system time
  - Sampling
    - Sample program counter, call stack, HW counters at specified intervals or specific events
  - Tracing
    - At function entry/exit, record performance data, function arguments, return values
    - pat_build must be instructed to instrument specific functions

- Many options for pat_run, pat_report. Too many to list here! See
  - http://info.nccs.gov/resources/phoenix/pat
  - Chapter 2 of Optimizing Applications on Cray X1 Series Systems (available at docs.cray.com)

Performance reports

- I like PAT_RT_EXPERIMENT='samp_cs_time' to profile and sample callstack (Very useful: See where time is spent in calltree)

```
100.0% |    100.0% | 154438 |Total
|-----------------------------------------------
|  25.0% |     25.0% |  38644 |pe.3
||----------------------------------------------
||   7.9% |      7.9% |  12160 |MatSetValuesLocal
|||---------------------------------------------
|||   7.9% |      7.9% |  12159 |matsetvalueslocal_
|||        |           |        | thcjacobian@thc_module_
|||        |           |        |  oursnesjacobian
|||        |           |        |   SNESComputeJacobian
|||        |           |        |    SNESSolve_LS
|||        |           |        |     SNESSolve
|||        |           |        |      snessolve_
|||        |           |        |       pflowgrid_step@pflow_grid_module_
|||        |           |        |        main
|||   0.0% |      7.9% |      1 |DAGetMatrix3d_MPIAIJ
|||        |           |        | DAGetMatrix
|||        |           |        |  dagetmatrix_
|||        |           |        |   pflowgrid_setup@pflow_grid_module_
|||        |           |        |    main
```

UT-BATTELLE
Oak Ridge National Laboratory

Optimization priorities

- Profile should always guide where optimization is done
- In hotspots routines, optimization priorities:
  1. Vectorization (10x or more speedup)
  2. Multistreaming (4x)
  3. Low-latency communication (2x)
  4. Register blocking (< 2x)
  5. Cache blocking (< 2x)

Vectorization

Exploiting fine-grain parallelism with vectorization is #1 priority on X1E!

- One vector instruction == many loop iterations
- Need a large enough number of loop iterations
  - SSP vector register holds 64 doubles
  - More than 64 iterations is ideal (for pipelining, multistreaming)
  - Fewer iterations means lower efficiency
- No procedure calls inside loop
- No loop-carried data dependencies
  - Some exceptions, e.g., reduction operations

Multistreaming

- Multistreaming takes additional advantage of loop-level parallelism.
- Loop iterations divided among 4 SSPs within an MSP
- Usually 2nd most important priority: Up to 4x speedup

- Many of the same considerations as w/ vectorization
- Additional wrinkle: When to stream vs. vectorize?
  - For streamed and vectorized loop nests,
    want to vectorize loop with trip count that results in long vectors
  - May need to help the compiler by telling it
    - what to stream (!dir$ preferstream)
    - what to vectorize (!dir$ prefervector)

What compilers can/can't do

The compiler can do a lot for us:

- Re-arrange loop nests
- Reductions, (un)pack, scatter/gather
- Fuse loops and array statements
- Inline procedures (one level down)
- if statements within loops (Vector masks, some loss of efficiency)

But it cannot do things like:

- Make short vector loops efficient
- Make stride-1 (or -0) scatter/gather efficient
- Know that index arrays don't repeat
- do j = 1, n
  x(i(j)) = x(i(j)) + …
- Effectively inline many levels down

Loopmark listings

- With profile in hand, examine loopmark listings for hotspot routines, to see what the compiler could and couldn't do.

- Loopmark listings show the compiler optimizations applied:
  - What vectorized?
  - What multistreamed?
  - What was unrolled?
  - Why was X not vectorized?

- To obtain:
  - `robin> ftn -rm myprog.f`
  - `robin> cc -hlist=m myprog.c`

Example loopmark listing: Vectorized/streamed

```
 1.           subroutine vectorize1(nx,a,b,c,d)
 2.           real a(nx),b(nx),c(nx),d
 3.
 4.   MVr--< do i = 1, nx
 5.   MVr       c(i) = a(i) * b(i) + d
 6.   MVr--> end do
 7.
 8.           end subroutine

ftn-6005 ftn: SCALAR File = vectorize1.ftn, Line = 4
  A loop starting at line 4 was unrolled 2 times.

ftn-6204 ftn: VECTOR File = vectorize1.ftn, Line = 4
  A loop starting at line 4 was vectorized.

ftn-6601 ftn: STREAM File = vectorize1.ftn, Line = 4
  A loop starting at line 4 was multi-streamed.
```

UT-BATTELLE
Oak Ridge National Laboratory

What if code doesn't vectorize/multistream?

- Last slide showed perfectly vectorized/multistreamed loop
- When this doesn't happen, try (in order of difficulty):
  1. Using compiler option flags:
     - `-h aggress` to attempt more aggressive loop optimizations
  2. Using compiler directives to give compiler hints:
     - !dir$ in Fortran, #pragma _CRI in C/C++
     - e.g. !dir$ concurrent to assert loop is free of dependencies
  3. Rewriting code
     - Simple stuff: switching loop indices, fusing loops, etc.
     - Complicated stuff: Rewriting data structures,
       choosing more vectorizable algorithms (extreme case)

Example loopmark: partial vectorization

Here, indirect addressing prevents compiler from knowing if index collisions occur:

```
 6.   Vp----<                    DO i = 1,n
 7.   VP r-<>                          e(ix1(i)) = e(ix1(i)) - a(i)
 8.   VP---->                    END DO
 9.
10.                                          end
```

```
f90-6371 f90: VECTOR File = gs-2.f, Line = 6
  A vectorized loop contains potential conflicts due to indirect
  addressing at line 7, causing less efficient code to be generated.

f90-6204 f90: VECTOR File = gs-2.f, Line = 6
  A loop starting at line 6 was vectorized.
```

UT-BATTELLE
OAK RIDGE NATIONAL LABORATORY

Example loopmark: Using 'concurrent' directive

Fix by using `!dir$ concurrent` to assert that loop has no vector dependencies:

```
  6.          !dir$ concurrent
  7.   MV--<        DO i = 1, n
  8.   MV             e(ix1(i)) = e(ix1(i)) - a(i)
  9.   MV-->        END DO
 10.
 11.               end
```

```
f90-6203 f90: VECTOR File = gs-2.f, Line = 7
  A loop starting at line 7 was vectorized because an IVDEP
  or CONCURRENT compiler directive was specified.

f90-6203 f90: STREAM File = gs-2.f, Line = 7
  A loop starting at line 7 was streamed because an IVDEP
  or CONCURRENT compiler directive was specified.
```

Example loopmark: IO within loop

Here, IO needs to be moved outside of a loop by the programmer:

```
 1.          subroutine io1(nx,a,b,c)
 2.          real a(nx),b(nx),c(nx)
 3.
 4.          open(8,file='c_array',access='direct', &
 5.               form='formatted',status='replace')
 6.    1--<  do i = 1, nx
 7.    1       c(i) = a(i) * b(i)
 8.    1       write(8,'(1x,f12.4)',rec=i) c(i)
 9.    1-->  end do
10.
11.          end subroutine

ftn-6286 ftn: VECTOR File = io1.ftn, Line = 6
  A loop starting at line 6 was not vectorized because it contains
  input/output operations at line 8.

ftn-6709 ftn: STREAM File = io1.ftn, Line = 6
  A loop starting at line 6 was not multi-streamed because it contains
  input/output operations.
```

Example loopmark: IO moved outside

The problem is fixed by manually segmenting the loop:

```
 1.           subroutine io2(nx,a,b,c)
 2.           real a(nx),b(nx),c(nx)
 3.
 4.           open(8,file='c_array',access='direct', &
 5.                form='formatted',status='replace')
 6.
 7.   MVr--< do i = 1, nx
 8.   MVr       c(i) = a(i) * b(i)
 9.   MVr--> end do
10.
11.           write(8,'(1x,f12.4)',rec=i) (c(i),i=1,nx)
12.
13.           end subroutine

ftn-6005 ftn: SCALAR File = io2.ftn, Line = 7
  A loop starting at line 7 was unrolled 2 times.

ftn-6204 ftn: VECTOR File = io2.ftn, Line = 7
  A loop starting at line 7 was vectorized.

ftn-6601 ftn: STREAM File = io2.ftn, Line = 7
  A loop starting at line 7 was multi-streamed.
```

Other optimization priorities

Haven't discussed other priorities:

- Communication latency can be reduced by
  - Strategic use of Co-Array Fortran
  - Use of SHMEM, UPC, or MPI-2
  - Remote load-store using pointers
  - See docs.cray.com. Or see the Cray folks at this meeting!

- Register blocking, cache blocking
  - Standard techniques covered in many sources
  - E.g., O'Reilly *High Performance Computing* by Severance and Dowd

Where to go for more help

- Much of the information discussed here can be found at
  http://info.nccs.gov/resources/phoenix

- Many more documents available at http://docs.cray.com
  - Cray X1 Series System Overview
  - Migrating Applications to the Cray X1 Series Systems
  - Optimizing Applications on Cray Series Systems
    (Some of my examples came from here)
  - Cray Fortran, C/C++ reference manuals

- Attend the Cray tutorial/workshop this Wednesday.

- Email help@nccs.gov