Cray Reveal Webinar: A Tool to Help Porting to Manycore

Heidi Poxon
Technical Lead & Sr. Manager, Performance Tools
Cray Inc.
This presentation may contain forward-looking statements that are based on our current expectations. Forward looking statements may include statements about our financial guidance and expected operating results, our opportunities and future potential, our product development and new product introduction plans, our ability to expand and penetrate our addressable markets and other statements that are not historical facts. These statements are only predictions and actual results may materially vary from those projected. Please refer to Cray's documents filed with the SEC from time to time concerning factors that could affect the Company and these forward-looking statements.
Future Architecture Directions

● Nodes are becoming more parallel
  ● More processors per node
  ● More threads per processor
  ● Vector lengths are getting longer
  ● Memory hierarchy is becoming more complex
  ● Scalar performance is not increasing and will start decreasing

● For the next decade, HPC systems will have the same basic architecture:
  ● Message passing between nodes
  ● Multithreading within the node (pure MPI will not do)
  ● Vectorization at the lowest level (SSE, AVX, GPU, Phi)
Future Application Directions

● Threading on node as well as vectorization is becoming more important – need more parallelism exploited in applications due to increasing number of cores and threads

● Current petascale applications are not structured to take advantage of these architectures
  ● Currently 80-90% of applications use a single level of parallelism
    • MPI or PGAS between cores of the MPP system

● Looking forward, application developers are faced with a significant task in preparing their applications for the future
  ● Codes must be converted to use multiple levels of parallelism
  ● More complex memory hierarchies will require user intervention to achieve good performance
Three Levels of Parallelism Required

1. Developers will continue to use MPI between nodes or sockets

2. Developers must address using a shared memory programming paradigm on the node

3. Developers must vectorize low level looping structures

While there is a potential acceptance of new languages for addressing all levels directly. Most developers cannot afford this approach until they are assured that the new language will be accepted and the generated code is within a reasonable performance range
When to Move to a Hybrid Programming Model

● **When code is network bound**
  - Look at collective time, excluding sync time: this goes up as network becomes a problem
  - Look at point-to-point wait times: if these go up, network may be a problem

● **When MPI starts leveling off**
  - Too much memory used, even if on-node shared communication is available
  - As the number of MPI ranks increases, more off-node communication can result, creating a network injection issue

● **When contention of shared resources increases**

● **When you want to exploit heterogeneous nodes**
Approach to Adding Parallelism

1. Identify key high-level loops
   ● Determine where to add additional levels of parallelism
     ● Assumes MPI application is functioning correctly on X86
     ● Find top serial work-intensive loops (perftools + CCE loop work estimates)

2. Perform parallel analysis, scoping and vectorization
   ● Split loop work among threads
     ● Do parallel analysis and restructuring on targeted high level loops
     ● Use Reveal + CCE for scoping, loopmark and source browsing

3. Add OpenMP layer of parallelism
   ● Insert OpenMP directives (with Reveal directive building assistance)
     ● Run on X86 to verify application and check for performance improvements

4. Analyze performance for further optimizations, specifically vectorization of innermost loops
Challenges

● **Investigate parallelizability of high level looping structures**
  - Often times one level of loop is not enough, must have several parallel loops
    ● Need a large number of loop iterations to feed the GPU threads
  - User must understand which high level DO loops have independent iterations

● Without tools, variable scoping of high level loops is very difficult
  - Loops must be more than independent, their variable usage must adhere to private data local to a thread or global shared across all the threads
  - Independence can be complicated to understand (and even runtime dependent)

● **Investigate vectorizability of lower level DO loops**
The Problem – How Do I Parallelize This Loop?

- How do I know this is a good loop to parallelize?
- What prevents me from parallelizing this loop?
- Can I get help building a directive?

```
subroutine sweepz
  ...
  do j = 1, js
    do i = 1, isz
      radius = zxc(i+mypez*isz)
      theta  = zyc(j+mypey*js)
      do m = 1, npez
        do k = 1, ks
          n = k + ks*(m-1) + 6
          r(n) = recv3(1,j,k,i,m)
          p(n) = recv3(2,j,k,i,m)
          u(n) = recv3(3,j,k,i,m)
          v(n) = recv3(4,j,k,i,m)
          f(n) = recv3(5,j,k,i,m)
        enddo
      enddo
    enddo
  enddo
  ...
  call ppmlr
  do k = 1, kmax
    n = k + 6
    xa(n) = zza(k)
    dx(n) = zdz(k)
    xa0(n) = zza(k)
    dx0(n) = zdz(k)
    e(n) = p(n)/(r(n)*gamm)+0.5 & *(u(n)**2+v(n)**2+w(n)**2)
  enddo
  call ppmlr
  ...
enddo
```

```
subroutine ppmlr

  call boundary
  call flatten
  call paraset(nmin-4, nmax+5, para, dx, xa)
  call parabola(nmin-4,nmax+4,para,p,dp,p6,pl,flat)
  call parabola(nmin-4,nmax+4,para,r,dr,r6,rl,flat)
  call parabola(nmin-4,nmax+4,para,u,du,u6,ul,flat)
  call states(pl,ul,rl,p6,u6,r6,dp,du,dr,plft,ulft,rlft,prgh,urgh,rrgh)
  call riemann(nmin-3,nmax+4,gam,prgh,urgh,rrgh,plft,ulft,rlft pmid umid)
  call evolve(umid, pmid)  \* contains more calls
  call remap  \* contains more calls
  call volume(nmin,nmax,ngeom,radius,xa,dx,dvol)
  call remap  \* contains more calls
  return
End
```
Simplifying the Task with Reveal

- Navigate to relevant loops to parallelize
- Identify parallelization and scoping issues
- Get feedback on issues down the call chain (shared reductions, etc.)
- Optionally insert parallel directives into source
- Validate scoping correctness on existing directives
Using Reveal with Performance Statistics

Optionally *create loop statistics using the Cray performance tools to determine which loops have the most work*

- Helps identify high-level serial loops to parallelize
  - Based on runtime analysis, approximates how much work exists within a loop

- Provides the following statistics
  - Min, max and average trip counts
  - Inclusive time spent in loops
  - Number of times a loop was executed
Collecting Loop Work Estimates

- Load PrgEnv-cray module (must use CCE)
- Load perftools module

- Compile **AND** link with `–h profile_generate`
  - `cc –h profile_generate –o my_program my_program.c`

- Instrument binary for tracing
  - `pat_build –w my_program`

- Run application

- Create report with loop statistics
  - `pat_report my_program.xf > loops_report`

*pat_report produces report plus .ap2 file that can be used with Reveal*
Example Report – Inclusive Loop Time

Table 2: Loop Stats by Function (from -hprofile_generate)

<table>
<thead>
<tr>
<th>Loop Incl</th>
<th>Loop Hit</th>
<th>Loop Trips</th>
<th>Loop Trips</th>
<th>Loop Trips</th>
<th>Function=/&gt;.LOOP[.]</th>
</tr>
</thead>
<tbody>
<tr>
<td>8.995914</td>
<td>100</td>
<td>25</td>
<td>0</td>
<td>25</td>
<td>sweepy_.LOOP.1.1i.33</td>
</tr>
<tr>
<td>8.995604</td>
<td>2500</td>
<td>25</td>
<td>0</td>
<td>25</td>
<td>sweepy_.LOOP.2.1i.34</td>
</tr>
<tr>
<td>8.894750</td>
<td>50</td>
<td>25</td>
<td>0</td>
<td>25</td>
<td>sweepz_.LOOP.05.1i.49</td>
</tr>
<tr>
<td>8.894637</td>
<td>1250</td>
<td>25</td>
<td>0</td>
<td>25</td>
<td>sweepz_.LOOP.06.1i.50</td>
</tr>
<tr>
<td>4.420629</td>
<td>50</td>
<td>25</td>
<td>0</td>
<td>25</td>
<td>sweepx2_.LOOP.1.1i.29</td>
</tr>
<tr>
<td>4.420536</td>
<td>1250</td>
<td>25</td>
<td>0</td>
<td>25</td>
<td>sweepx2_.LOOP.2.1i.30</td>
</tr>
<tr>
<td>4.387534</td>
<td>50</td>
<td>25</td>
<td>0</td>
<td>25</td>
<td>sweepx1_.LOOP.1.1i.29</td>
</tr>
<tr>
<td>4.387457</td>
<td>1250</td>
<td>25</td>
<td>0</td>
<td>25</td>
<td>sweepx1_.LOOP.2.1i.30</td>
</tr>
<tr>
<td>2.523214</td>
<td>187500</td>
<td>107</td>
<td>0</td>
<td>107</td>
<td>riemann_.LOOP.2.1i.63</td>
</tr>
<tr>
<td>1.541299</td>
<td>20062500</td>
<td>12</td>
<td>0</td>
<td>12</td>
<td>riemann_.LOOP.3.1i.64</td>
</tr>
<tr>
<td>0.863656</td>
<td>1687500</td>
<td>104</td>
<td>0</td>
<td>108</td>
<td>parabola_.LOOP.6.1i.67</td>
</tr>
</tbody>
</table>
How to Use Reveal

- Generate a program library for your application with CCE
  - `> cc -h pl=himeno.pl -hwp himeno.c`
  - `> ftn -h pl=vhone.pl -hwp file1.f90`

- Launch Reveal
  - `> module load perftools`
  - Use with compiler information only (no need to run program):
    - `> reveal vhone.pl`
  - Use with compiler + loop work estimates (include performance data)
    - `> reveal vhone.pl vhone_loops.ap2`
Browse Source and Compiler Optimizations
Access integrated message ‘explain’ support by right clicking on message.
Navigate Code via Compiler Messages

Choose “Compiler Messages” view to access message filtering

Default filter: Loops that didn’t vectorize. Can select other filters.
View Pseudo Code for Inlined Functions

- Inlined call sites marked
- Expand to see pseudo code
- Search code with Ctrl-F
Add Performance Data to Find Top Loops
View Loops through Call Chain

Loop instances

Loop traceback
Scope Top Time Consuming Loops
Include All Loops as Initial Candidates
Include All Loops as Initial Candidates (2)
Apply Filter to Select Only Top Loops
View Scoping Results

Right click on loop to add it to list of loops to scope
Reveal Gives Feedback on Scoping Results

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Scope</th>
<th>Info</th>
</tr>
</thead>
<tbody>
<tr>
<td>ar@parabola_</td>
<td>Scalar</td>
<td>Unresolved</td>
<td>Possible recurrence involving this object.</td>
</tr>
<tr>
<td>da@parabola_</td>
<td>Scalar</td>
<td>Unresolved</td>
<td>Possible recurrence involving this object.</td>
</tr>
<tr>
<td>delta@remap_</td>
<td>Scalar</td>
<td>Unresolved</td>
<td>Possible recurrence involving this object.</td>
</tr>
<tr>
<td>dvol</td>
<td>Array</td>
<td>Unresolved</td>
<td>Possible recurrence involving this object.</td>
</tr>
<tr>
<td>dx</td>
<td>Array</td>
<td>Unresolved</td>
<td>Possible recurrence involving this object.</td>
</tr>
<tr>
<td>dx0</td>
<td>Array</td>
<td>Unresolved</td>
<td>Possible recurrence involving this object.</td>
</tr>
</tbody>
</table>

Variable from inlining – hover over ‘I’ to see what symbol means

See where variable came from (@function_name)
Reveal Points Out Parallelization Issues

Reveal identifies shared reductions down the call chain.
Generate Directive

Reveal generates example OpenMP directive
Optionally Insert Directive Into Source
Reveal Inserts Directive Into Source

! Directive inserted by Cray Reveal. May be incomplete.
!$OMP  parallel do default(none) &
!$OMP&  unresolved (dvol,dx,dx0,e,f,flat,p,para,q,r,radius,svel,u,v,w, &
!$OMP&     xa,xa0) &
!$OMP&  private (i,j,k,m,n,$$_n,delp2,delp1,shock,temp2,old_flat, &
!$OMP&    onemfl,hdt,sinx0,gamfac1,gamfac2,dtheta,deltx,fractn, &
!$OMP&    ekin) &
!$OMP&  shared  (gamm,isy,iks,mpey,ndim,ngeomy,nlefty,npey,nrighty, &
!$OMP&      recv1,send2,zdy,zxc,zya)
do k = 1, ks
  do i = 1, isy
    radius = zxc(i+mpey*isy)

    ! Put state variables into 1D arrays, padding with 6 ghost zones
    do m = 1, npey
      do j = 1, js
        n = j + js*(m-1) + 6
        r(n) = recv1(1,k,j,i,m)
        p(n) = recv1(2,k,j,i,m)
        u(n) = recv1(4,k,j,i,m)
        v(n) = recv1(5,k,j,i,m)
        w(n) = recv1(3,k,j,i,m)
        f(n) = recv1(6,k,j,i,m)
      enddo
    enddo
  endo

  do j = 1, jmax
    n = j + 6
    ...

Reveal generates OpenMP directive with illegal clause marking variables that need addressing
Resolve Private Array Concerns for dvol, etc.

From file vh1mods.f90:

```fortran
! module sweeps
!=======================================================================
! Data structures used in 1D sweeps, dimensioned maxsweep  (set in sweepsize.mod)
!=======================================================================

use sweepsize

integer :: nmin, nmax, ngeom, nleft, nright
real, dimension(maxsweep) :: r, p, e, q, u, v, w  ! fluid variables
real, dimension(maxsweep) :: xa, xa0, dx, dx0, dvol  ! coordinate values
real, dimension(maxsweep) :: f, flat  ! flattening parameter
real, dimension(maxsweep,5) :: para  ! parabolic interpolation coefficients
real :: radius, theta, stheta

!$omp threadprivate(dvol,dx,dx0,e,f,flat,p,para,q,r,radius,theta,stheta,u,v,w,xa,xa0)

For OpenMP these need to be made task_private
```
Resolve Shared Reductions

Original

\[
\begin{align*}
\text{hdt} & = 0.5 \cdot dt \\
\text{do } n = n\text{min}-4, \ n\text{max}+4 \\
\quad & \text{Cdtdx} \ (n) = \sqrt{\text{gam} \cdot p(n)/r(n)}/(\text{dx}(n) \cdot \text{radius}) \\
\quad & \text{svel} = \max(\text{svel}, \text{Cdtdx}(n)) \\
\quad & \text{Cdtdx} \ (n) = \text{Cdtdx}(n) \cdot \text{hdt} \\
\quad & \text{fCdtdx}(n) = 1. - \text{fourthd} \cdot \text{Cdtdx}(n) \\
\text{enddo}
\end{align*}
\]

Restructured – One Approach

\[
\begin{align*}
\text{hdt} & = 0.5 \cdot dt \\
\text{!$omp \ critical$
\text{do } n = n\text{min}-4, \ n\text{max}+4 \\
\quad & \text{Cdtdx} \ (n) = \sqrt{\text{gam} \cdot p(n)/r(n)}/(\text{dx}(n) \cdot \text{radius}) \\
\quad & \text{svel} = \max(\text{svel}, \text{Cdtdx}(n)) \\
\quad & \text{Cdtdx} \ (n) = \text{Cdtdx}(n) \cdot \text{hdt} \\
\quad & \text{fCdtdx}(n) = 1. - \text{fourthd} \cdot \text{Cdtdx}(n) \\
\text{enddo} \\
\text{!$omp \ end \ critical$
\end{align*}
\]

For OpenMP need to have a critical region around setting of svel
Resolve Shared Reductions (Continued)

Original

```fortran
hdt = 0.5*dt
do n = nmin-4, nmax+4
   Ctdtnx (n) = sqrt(gam*p(n)/r(n))/(dx(n)*radius)
   svel = max(svel,Ctdtx(n))
   Ctdtx (n) = Ctdtx(n)*hdt
   fCtdtx(n) = 1. - fourthd*Ctdtx(n)
enddo
```

Restructured – Better Approach

```fortran
hdt = 0.5*dt
Svel0 = 0.0
do n = nmin-4, nmax+4
   Ctdtnx (n) = sqrt(gam*p(n)/r(n))/(dx(n)*radius)
   svel0(n) = max(svel(n),Ctdtnx(n))
   Ctdtx (n) = Ctdtx(n)*hdt
   fCtdtx(n) = 1. - fourthd*Ctdtx(n)
Enddo
!$omp critical
Do n = nmin-4, nmax +4
   Svel = max(svel0(n),svel)
Enddo
!$omp end critical
```
Use Reveal to Validate User Inserted Directives

![Reveal screen shot](image.png)

User inserted directive with mis-scoped variable 'n'
VH1 – Astrophysics Code

- VH1 is written with high level loops and complex decision processes.
- Ported to hybrid MPI + OpenMP using Reveal
- Reveal was able to identify
  - storage conflicts
  - private variables in modules
  - reductions down the call chain that require critical regions
- Scoping was performed in seconds where it would have taken weeks to get correct without Reveal
S3D - Structured Cartesian Mesh Flow Solver

● S3D, a pure MPI program, was converted to a hybrid multi-core application suited for a multi-core node with or without an accelerator.

● When the work was started, Reveal did not exist.

● Once Reveal was available, it was instrumental in identifying bugs in the scoping of extremely large loops (3000 lines of Fortran).

● There are both OpenMP and OpenACC versions of S3D that run well on both OpenMP systems and on the Titan Cray XK7 machine at Oak Ridge National Laboratory.
Summary

- Reveal can be used to simplify the task of adding OpenMP to MPI programs

- Can be used as a stepping stone for codes targeted for nodes with higher core counts and as the first step in adding OpenACC to applications to for execution on GPUs
Legal Disclaimer

Information in this document is provided in connection with Cray Inc. products. No license, express or implied, to any intellectual property rights is granted by this document.

Cray Inc. may make changes to specifications and product descriptions at any time, without notice.

All products, dates and figures specified are preliminary based on current expectations, and are subject to change without notice.

Cray hardware and software products may contain design defects or errors known as errata, which may cause the product to deviate from published specifications. Current characterized errata are available on request.

Cray uses codenames internally to identify products that are in development and not yet publically announced for release. Customers and other third parties are not authorized by Cray Inc. to use codenames in advertising, promotion or marketing and any use of Cray Inc. internal codenames is at the sole risk of the user.

Performance tests and ratings are measured using specific systems and/or components and reflect the approximate performance of Cray Inc. products as measured by those tests. Any difference in system hardware or software design or configuration may affect actual performance.

The following are trademarks of Cray Inc. and are registered in the United States and other countries: CRAY and design, SONEXION, URIKA, and YARCDATA. The following are trademarks of Cray Inc.: ACE, APPRENTICE2, CHAPEL, CLUSTER CONNECT, CRAYPAT, CRAYPORT, ECOPHLEX, LIBSCI, NODEKARE, THREADSTORM. The following system family marks, and associated model number marks, are trademarks of Cray Inc.: CS, CX, XC, XE, XK, XMT, and XT. The registered trademark LINUX is used pursuant to a sublicense from LMI, the exclusive licensee of Linus Torvalds, owner of the mark on a worldwide basis. Other trademarks used in this document are the property of their respective owners.

Copyright 2014 Cray Inc.