More on Application profiling and optimization
Agenda

- Optimizing source code and controlling the compiler
- A Tour of the Apprentice2 GUI
- Optimizations for MPI – Rank Reordering
Doesn't the compiler do everything?

- **Not yet...**
  - Standard answer, unchanged for last 50 or so years

- **What does it do**
  - It tries to compile the loops in your application to be as fast as possible
  - Performance depends on reducing memory use and using the best machine instructions (vectorization)
  - This means your code may be significantly transformed

- **What can you do**
  - Work out what you care about (profile)
  - Experiment with alternative source implementations but a lot of expertise is needed here
  - Give the compiler additional information
  - Use compiler output to determine what it is doing and influence it via directives
Loop optimisation techniques

● Most HPC codes are loop-based
  ● Repeatedly process all the elements of an array

● There are various optimization techniques for loops
  ● unrolling/unwinding
  ● stripmining
  ● blocking/tiling

● We are not going to explain HOW to do this manually but it is useful to be aware of these even if you are not going to optimise source

● In many cases, the compiler does these automatically
  ● the material here will help you understand what the compiler did
  ● if necessary, you can then step in to assist the compiler
EXAMPLE 1: Loop unrolling/unwinding

- Unrolling and unwinding are equivalent terms
- Replaces a loop by an equivalent set of statements
  - Removes the overhead of loop control logic
    - incrementing the loop index counter
    - checking if the counter has exceeded the loop bounds
- Most important for small tripcount/low work loops
  - Especially when nested inside other loops
  - Full unwinding requires tripcount to be known at compile time

<table>
<thead>
<tr>
<th>Original code</th>
<th>After unwinding</th>
</tr>
</thead>
<tbody>
<tr>
<td>do i=1,N</td>
<td>a(i) =a(i) + b(i)</td>
</tr>
<tr>
<td>a(i)=a(i) + b(i)</td>
<td>a(i+1)=a(i+1) + b(i+1)</td>
</tr>
<tr>
<td>enddo</td>
<td>a(i+2)=a(i+2) + b(i+2)</td>
</tr>
<tr>
<td>:</td>
<td></td>
</tr>
<tr>
<td></td>
<td>a(N) =a(N) + b(N)</td>
</tr>
</tbody>
</table>
Example 2: Loop blocking/tiling

- **Applied to multi-dimensional loop nests**
  - Two or more loops are stripmined
  - Loop interchange moves the strip loops innermost
- **Most often used to preserve memory locality**

<table>
<thead>
<tr>
<th>Original loopnest</th>
<th>Equivalent explicit code</th>
</tr>
</thead>
</table>
| `do j = 1,Nj      
  do i = 1,Ni      
  !stencil         
  enddo            
  enddo`            | `do jb = 1,Nj,16       
  do ib = 1,Ni,16   
  do j = jb,jb+16-1 
  do i = ib,ib+16-1 
  !stencil         
  enddo            
  enddo            
  enddo            |

- (strictly, upper strip loop limits should be MIN(Nj,jb+16-1) and similar)
Control: Example blocking with Cray Directives

- CCE blocks well, but it sometimes blocks better with help

<table>
<thead>
<tr>
<th>Original loopnest</th>
<th>Loopnest with help</th>
<th>Equivalent explicit code</th>
</tr>
</thead>
<tbody>
<tr>
<td>do k = 1,Nk</td>
<td>!DIR$ BLOCKABLE(j,k)</td>
<td>do kb = 1,Nk,16</td>
</tr>
<tr>
<td>do j = 1,Nj</td>
<td>!dir$ BLOCKINGSIZE(16)</td>
<td>do jb = 1,Nj,20</td>
</tr>
<tr>
<td>do i = 1,Ni</td>
<td>! stencil</td>
<td>do k = kb,kb+16-1</td>
</tr>
<tr>
<td>! stencil</td>
<td></td>
<td>do j = jb,jb+20-1</td>
</tr>
<tr>
<td>enddo</td>
<td></td>
<td>do i = 6, nx-5</td>
</tr>
<tr>
<td>enddo</td>
<td></td>
<td>! stencil</td>
</tr>
<tr>
<td>enddo</td>
<td></td>
<td>enddo</td>
</tr>
<tr>
<td>enddo</td>
<td></td>
<td>enddo</td>
</tr>
</tbody>
</table>

- (again, upper limits should be MIN(Nk,kb+16-1) and similar)

- Get the loopmark listing
  - Identifies which loops were blocked
  - Gives the block size the compiler chose
Example 3: Loop interchange

- One of the simplest cache optimisations
  - aim to access consecutive elements of arrays in order
- If multi-dimensional arrays addressed in wrong order
  - causes a lot of cache misses = bad performance
- Order loops in loopnest with fastest innermost
  - Fortran is column-major (LH array index moves fastest)
  - C/C++ is row-major (RH array index moves fastest)

- Compiler may re-order loops automatically (see loopmark)

<table>
<thead>
<tr>
<th>Original loopnest</th>
<th>interchanged code</th>
</tr>
</thead>
<tbody>
<tr>
<td>do i = 1,N&lt;br&gt;do j = 1,N&lt;br&gt;tot = tot + a(i,j)&lt;br&gt;enddo&lt;br&gt;enddo</td>
<td>do j = 1,N&lt;br&gt;do i = 1,N&lt;br&gt;tot = tot + a(i,j)&lt;br&gt;enddo&lt;br&gt;enddo</td>
</tr>
</tbody>
</table>
Optimization for memory access, huge pages

- Various loop transformations we have seen
  - Help with memory access order
  - This makes more efficient use of cache
    - Use as much cache as possible
    - Reuse data when it is in cache

- There is a level beyond cache size to consider
- We have virtual memory pages which map to physical pages
- The OS keeps track of this in hardware (TLB) and software
- As a result we should try to reuse memory within a page
Using hugepages

- Load chosen `craype-hugepages*` module
  - See `module avail craype-hugepages` for list of available options
- Compile as before
- Execute as before, but
  - Make sure this module is also loaded in PBS jobsheet
    - It sets various environment variables

- Which pagesize is best?
  - You should try different settings
  - 2M or 8M are usually most successful on Cray XC systems

- Quick cheat:
  - No need to rebuild to try a different pagesize
  - Can load different hugepages module at runtime
    - Compared to that used at compile-time
  - Compile-time module enables hugepages in the application
  - Runtime module determines the actual size that is used

- See `man intro_hugepages` for more details
Vectorisation

- The most important optimization is for memory access
- Then we can think of optimising computation
- This will be in loops
- Usually only one loop is vectorisable in loopnest
  - And most compilers (not CCE) only consider inner loop
- Optimising compilers will use vector instructions
  - Relies on code being vectorisable
  - Or in a form that the compiler can convert to be vectorisable
    - Some compilers are better at this than others
- Check the compiler output listing and/or assembler listing
  - Look for packed SSE/AVX instructions
Helping vectorisation

● **Is there a good reason for this?**
  ● There is an overhead in setting up vectorisation; maybe it's not worth it
    ● Could you unroll inner (or outer) loop to provide more work?

● **Does the loop have dependencies?**
  ● information carried between iterations
    ● e.g. counter: total = total + a(i)

● **If there are no loop dependencies:**
  ● Tell the compiler that it is safe to vectorise
    ● **IVDEP** directive above loop (CCE, but works with most compilers)
    ● C99: **restrict** keyword (or compile with -hrestrict=a with CCE)
  ● Perhaps the dependencies are between iterations i and i+8
    ● Then it is safe to vectorise with vectors of length 8 or less
    ● Use directive: **IVDEP SAFEVL=8**
  ● see **man ivdep** for more details
Inhibitors to vectorisation

- **loop dependencies:**
  - The loop cannot be executed in any order
  - Might be hard to rewrite code to fix this
- **Code is not a loop (do while)**
- **Indirect addressing**
- **Non-vectorisable functions**
- **Unknown loop trip count**
- **Function calls in loop need to be inlined**

- **Check the compiler output to see what it did**
  - CCE: `-hlist=a`
  - Intel: `-vec-report[0..5]`
  - GNU: `-ftree-vectorizer-verbose=5`
CCE directives
Some useful CCE directives

- Compiler directives avoid the need for explicit coding
  - They are compiler-specific but should be ignored as comments by:
    - other compilers
    - the same compiler, if overridden by compiler options
- CCE has a large set of optimisation directives
  - Fortran:  !DIR$ <directive>
  - C/C++:  #pragma _CRI <directive>
    - _CRI optional; include it so compiler warns about unrecognised directives
- Some useful ones are listed on the next few slides
- For more information:
  - man directives
  - man <directive name>
  - Fortran, C/C++ Reference Manuals on docs.cray.com
Selected CCE scalar optimisation directives

● INTERCHANGE (i,j...), NOINTERCHANGE
  ● Specified loops should be interchanged, e.g. (i,j,k) -> (k,j,i)
  ● NOINTERCHANGE directive suppresses loop interchange

● UNROLL [n], NOUNROLL
  ● Specify unrolling of next loop, with optional unroll factor

● BLOCKABLE (i,j...) 
  ● Specified loops can be blocked
  ● NOBLOCKING directive prevents blocking

● BLOCKINGSIZE (n)
  ● Apply blocking factor n to next loop
  ● Use separate BLOCKINGSIZE directives for each loop to be blocked

● FUSION, NOFUSION, NOFISSION
  ● Control loop fusion and fission of specified loop
Selected CCE vectorisation directives (1)

- **IVDEP**
  - Ignore dependencies in the next loop that might inhibit vectorisation

- **NEXTSCALAR**
  - Do not vectorise the next loop

- **PREFERVECTOR**
  - If more than one loop in nest can be vectorised, indicates preference
  - Has the same effect as VECTOR ALWAYS directive

- **NOVECTOR**
  - Disable vectorisation for rest of program unit;
  - reset behaviour with VECTOR directive
Selected CCE vectorisation directives (2)

- **LOOP_INFO** \([\text{min\_trips}(c)] \ [\text{est\_trips}(c)] \ [\text{max\_trips}(c)]\)
  - Provide information on min/mean/max tripcounts for loop

- **PROBABILITY**
  - Indicate probability of a conditional being true
  - May suggest compiler uses gather/scatter methods to vectorise loop

- **PERMUTATION**
  - The specified integer array does not have repeated values
  - Useful for index array used in indirect addressing

- **CONCURRENT**
  - Stronger than IVDEP
    - IVDEP says loop iterations independent in current order
    - CONCURRENT says independent in any order
  - Both CONCURRENT and IVDEP should allow (possible) vectorisation
Concluding remarks

- Compilers are good at optimising code, but not perfect
- If you do nothing else with your code
  - Make sure you address arrays in the "right" order
  - Check the compiler feedback to see it's not doing anything foolish

To go further:
- Understand what the compiler does
  - Look at the compiler feedback in more detail
  - Use profiling and hardware counters to see if these optimisations work
- Help the compiler to understand your code
  - Simpler code is usually a good place to start
  - Use directives to give the compiler more information about your code
- Only start hand-coding optimisations as a last resort

And remember to keep profiling your code
- Optimise the things that take most time
A tour of the Apprentice2 GUI
The Three Stages of Profiling with perftools and CrayPat

1. **Instrumentation**
   - Build executable of an instrumented version of your application

2. **Running your application and Data Collection**
   - Run the instrumented version of your application
   - Transparent collection via CrayPat’s run-time library

3. **Analysis: Sampling / Tracing**
   - Interpret and visualize data using post-mortem tools:
     1. *pat_report*: a command line tool for generating text reports
     2. *Cray Apprentice2*: a graphical performance analysis tool
     3. *Reveal*: graphical performance analysis and code restructuring tool
Profile Visualization with Cray Apprentice2
Cray Apprentice2

● **Features:**
  ● Call graph profile
  ● Communication statistics
  ● Time-line view
    ● Communication
    ● I/O
  ● Activity view
  ● Pair-wise communication statistics
  ● Text reports
  ● Source code mapping

● **Helps identify:**
  ● Load imbalance
  ● Excessive communication
  ● Network contention
  ● Excessive serialization
  ● I/O Problems
To use Cray Apprentice²

● **You can run app2 on the login nodes:**
  ● You need an X session
    ● `ssh -X <system name>`
    ● and software to catch X windows on your local machine
  ● You need app2 in your path
    ● `module load perftools-base`
  ● The *.ap2 file contains the information (produced by pat_report)
    ● `app2 data_file_name.ap2`
    ● or you can load the ap2 file from the GUI

● **There is also a client version of app2**
  ● You can run this on your local machine
  ● Contact your site administrator for details on how to install this
  ● Then just need to copy the *.ap2 file to this machine
Cray Apprentice²
Statistics Overview: Pie Chart

1. Data tab: shows the name of the data file currently displayed
2. Report toolbar: show the reports that can be displayed for the data currently selected
3. Report tabs: show the reports
4. On many reports, the total duration of the experiment is shown as a graduated bar at the bottom of the window
5. Change view from pie chart to bar graph
6. Help menu

Note that report toolbar ONLY what you have decide to collected with pat_build
Statistics Overview: Bar Graph
## Function Profile View

### Table View

<table>
<thead>
<tr>
<th>Time</th>
<th>Percent</th>
<th>H/Hb</th>
<th>Collide</th>
<th>Imbalance</th>
<th>Potential</th>
<th>Function</th>
<th>Line</th>
<th>File</th>
</tr>
</thead>
<tbody>
<tr>
<td>124.176511</td>
<td>0.15</td>
<td>576</td>
<td>1</td>
<td>5.83</td>
<td></td>
<td>sweep_</td>
<td>0.15</td>
<td>/usr/local/apps/sweep3d/sweep.f</td>
</tr>
<tr>
<td>40.211774</td>
<td>0.25</td>
<td>48</td>
<td>1</td>
<td>48.25</td>
<td></td>
<td>mpi_recev_</td>
<td>0.25</td>
<td>/usr/local/apps/sweep3d/sweep.f</td>
</tr>
<tr>
<td>16.319527</td>
<td>0.30</td>
<td>1536</td>
<td>3</td>
<td>56.00</td>
<td></td>
<td>exit</td>
<td>0.30</td>
<td>/usr/local/apps/sweep3d/sweep.f</td>
</tr>
<tr>
<td>6.173298</td>
<td>0.12</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
<td>mpi_barrier</td>
<td>0.12</td>
<td>/usr/local/apps/sweep3d/sweep.f</td>
</tr>
<tr>
<td>2.750376</td>
<td>0.01</td>
<td>116000</td>
<td>1</td>
<td>17.58</td>
<td></td>
<td>mpi_send</td>
<td>0.01</td>
<td>/usr/local/apps/sweep3d/sweep.f</td>
</tr>
<tr>
<td>2.500288</td>
<td>0.00</td>
<td>576</td>
<td>1</td>
<td>2.62</td>
<td></td>
<td>source</td>
<td>0.00</td>
<td>/usr/local/apps/sweep3d/sweep.f</td>
</tr>
<tr>
<td>1.994620</td>
<td>0.00</td>
<td>144</td>
<td>1</td>
<td>2.59</td>
<td></td>
<td>mpi_bcast</td>
<td>0.00</td>
<td>/usr/local/apps/sweep3d/sweep.f</td>
</tr>
<tr>
<td>0.897399</td>
<td>0.00</td>
<td>192</td>
<td>2</td>
<td>2.17</td>
<td></td>
<td>mpi_send</td>
<td>0.00</td>
<td>/usr/local/apps/sweep3d/sweep.f</td>
</tr>
<tr>
<td>0.410231</td>
<td>0.00</td>
<td>1</td>
<td>1</td>
<td>2.99</td>
<td></td>
<td>flux_err</td>
<td>0.00</td>
<td>/usr/local/apps/sweep3d/sweep.f</td>
</tr>
<tr>
<td>0.382130</td>
<td>0.00</td>
<td>1</td>
<td>1</td>
<td>10.98</td>
<td></td>
<td>mpi_real</td>
<td>0.00</td>
<td>/usr/local/apps/sweep3d/sweep.f</td>
</tr>
<tr>
<td>0.237772</td>
<td>0.07</td>
<td>192</td>
<td>1</td>
<td>95.76</td>
<td></td>
<td>ftime</td>
<td>0.07</td>
<td>/usr/local/apps/sweep3d/sweep.f</td>
</tr>
<tr>
<td>0.185037</td>
<td>0.39</td>
<td>116000</td>
<td>2</td>
<td>17.30</td>
<td></td>
<td>mpi_send</td>
<td>0.39</td>
<td>/usr/local/apps/sweep3d/sweep.f</td>
</tr>
<tr>
<td>0.178712</td>
<td>0.04</td>
<td>48</td>
<td>1</td>
<td>4.95</td>
<td></td>
<td>mpi_initiate</td>
<td>0.04</td>
<td>/usr/local/apps/sweep3d/sweep.f</td>
</tr>
<tr>
<td>0.093047</td>
<td>0.00</td>
<td>48</td>
<td>1</td>
<td>4.50</td>
<td></td>
<td>mpi_init</td>
<td>0.00</td>
<td>/usr/local/apps/sweep3d/sweep.f</td>
</tr>
<tr>
<td>0.091371</td>
<td>0.02</td>
<td>48</td>
<td>1</td>
<td>23.94</td>
<td></td>
<td>mpi_recev</td>
<td>0.02</td>
<td>/usr/local/apps/sweep3d/sweep.f</td>
</tr>
<tr>
<td>0.023356</td>
<td>0.01</td>
<td>1</td>
<td>1</td>
<td>8.00</td>
<td></td>
<td>fxnic</td>
<td>0.01</td>
<td>/usr/local/apps/sweep3d/sweep.f</td>
</tr>
<tr>
<td>0.016962</td>
<td>0.01</td>
<td>68</td>
<td>1</td>
<td>28.14</td>
<td></td>
<td>getc</td>
<td>0.01</td>
<td>/usr/local/apps/sweep3d/sweep.f</td>
</tr>
<tr>
<td>0.003104</td>
<td>0.00</td>
<td>4892</td>
<td>2</td>
<td>20.14</td>
<td></td>
<td>octopi</td>
<td>0.00</td>
<td>/usr/local/apps/sweep3d/sweep.f</td>
</tr>
<tr>
<td>0.002457</td>
<td>0.00</td>
<td>576</td>
<td>1</td>
<td>18.29</td>
<td></td>
<td>global_real_init</td>
<td>0.00</td>
<td>/usr/local/apps/sweep3d/sweep.f</td>
</tr>
<tr>
<td>0.002085</td>
<td>0.00</td>
<td>48</td>
<td>1</td>
<td>69.55</td>
<td></td>
<td>MAIN</td>
<td>0.00</td>
<td>/usr/local/apps/sweep3d/sweep.f</td>
</tr>
<tr>
<td>0.001536</td>
<td>0.00</td>
<td>576</td>
<td>1</td>
<td>39.10</td>
<td></td>
<td>global_int_sum</td>
<td>0.00</td>
<td>/usr/local/apps/sweep3d/sweep.f</td>
</tr>
<tr>
<td>0.001383</td>
<td>0.00</td>
<td>48</td>
<td>1</td>
<td>10.23</td>
<td></td>
<td>inner_auto</td>
<td>0.00</td>
<td>/usr/local/apps/sweep3d/sweep.f</td>
</tr>
<tr>
<td>0.000682</td>
<td>0.00</td>
<td>48</td>
<td>1</td>
<td>97.24</td>
<td></td>
<td>task_init</td>
<td>0.00</td>
<td>/usr/local/apps/sweep3d/sweep.f</td>
</tr>
<tr>
<td>0.000739</td>
<td>0.00</td>
<td>384</td>
<td>2</td>
<td>27.37</td>
<td></td>
<td>global_real_sum</td>
<td>0.00</td>
<td>/usr/local/apps/sweep3d/sweep.f</td>
</tr>
<tr>
<td>0.000682</td>
<td>0.00</td>
<td>2</td>
<td>1</td>
<td>7.54</td>
<td></td>
<td>fopen</td>
<td>0.00</td>
<td>/usr/local/apps/sweep3d/sweep.f</td>
</tr>
<tr>
<td>0.000499</td>
<td>0.00</td>
<td>48</td>
<td>1</td>
<td>7.54</td>
<td></td>
<td>initnbc</td>
<td>0.00</td>
<td>/usr/local/apps/sweep3d/sweep.f</td>
</tr>
</tbody>
</table>
Load Balance View (Aggregated from Overview)

By clicking on a given function, we can show the breakdown per each PE.

[Diagram showing load balance view with bars indicating min, max, and avg values for each PE, and a note about ±1 std deviation.]
Call Tree View

Load balance overview:
Height ⇔ Max time
Middle bar ⇔ Average time
Lower bar ⇔ Min time
Yellow represents imbalance time

Width ⇔ inclusive time
Height ⇔ exclusive time
Filtered nodes or sub tree
DUH Button: Provides hints for performance tuning
Zoom

Function List
Call Tree View – Function List

Right mouse click: Node menu e.g., hide/unhide children

Right mouse click: View menu: e.g., Filter

Sort options
% Time, Time, Imbalance % Imbalance time

Function List off
Call Tree Visualization
Discrete Unit of Help (DUH Button)
Source Mapping from Call Graph view
Complementary performance data available in one place

Most reports easily accessible
  - using drop-down menu for easy navigation

Can easily generate new views of performance data

Provides mechanism for more in depth explanation of data presented
Example of pat_report Tables in Cray Apprentice2

<table>
<thead>
<tr>
<th>Time</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.00</td>
<td>0.00</td>
</tr>
<tr>
<td>49.30</td>
<td>98.59</td>
</tr>
<tr>
<td>147.89</td>
<td>197.18</td>
</tr>
</tbody>
</table>

New text table icon
Right click for table generation options
Generating New pat_report Tables

- Profile
- Custom...
- Source
- Calltree
- Callers
- Show Notes
- Show All PE's
- Show HWPC
- Use Thresholds

Panel Actions
Panel Help
Reduce ap2 file information

● Sometimes the amount of data in ap2 file can be large
  ● Very long-running applications
  ● Applications running on a large number of PEs

● The app2 command supports two options to help
  ● --limit and --limit_per_pe
  ● Restrict the amount of data being read in from the ap2 file
  ● use K, M, and G abbreviations for kilo, mega, and giga

● --limit sets a global limit on data size.

● --limit_per_pe sets limit per PE
  ● --limit_per_pe generally preferable (not always, but generally)
    ● preserves full parallelism in analysis

● Example: first 3M data items
  ● app2 --limit 3M data_file_name.ap2 &
Timeline views with Cray Apprentice²
Tracing

- **Show tracing results (Time Live View)**
  - Information broken out by communication type (read, write, barrier, and so on)

- **Only true function calls can be traced**
  - Functions that are inlined by the compiler or that have local scope in a compilation unit **cannot** be traced

- **Enabled with pat_build -g, -u, -T or -w options**

- **Full trace (sequence of events) enabled by setting**
  
  ```
  Pat_RT_SUMMARY=0
  ```
Time Line View (Sweep3D)
Time Line View (Zoom)

User Functions, MPI & SHMEM Line

I/O Line
Time Line View (Fine Grain Zoom)
Other Cray Apprentice2 Reports

- **Environment reports**
  - Provide general information about the conditions under which the data file currently being examined was created

- **Traffic Report**
  - Shows internal PE-to-PE traffic over time. The information is broken down by comm. type (read, write, barrier etc.)

- **I/O Rates Report**
  - Table listing quantitative information about program's I/O usage.
    - Look for I/O activities that have low average rates and high data volumes; this may indicate that the file should be moved to a different file system.

- **Hardware reports**
  - Available only if hardware counter information was captured

- **Full description at:** [http://docs.cray.com/books/S-2376-63/S-2376-63.pdf](http://docs.cray.com/books/S-2376-63/S-2376-63.pdf)
Compiler feedback and variable scoping with Reveal
● For an OpenMP port a developer has to understand the scoping of the variables, i.e. whether variables are shared or private.

● **Reveal** is Cray’s next-generation integrated performance analysis and code optimization tool.
  
  ● Source code navigation using whole program analysis (data provided by the Cray compilation environment.)
  
  ● Coupling with performance data collected during execution by CrayPAT. Understand which high level serial loops could benefit from parallelism.
  
  ● Enhanced loop mark listing functionality.
  
  ● Dependency information for targeted loops
  
  ● Assist users optimize code by providing variable scoping feedback and suggested compile directives.
Input to Reveal

- Recompile to generate program library
- Performance data from a separate loop timing trace experiment
- Launch Reveal

```bash
$> module load perftools
$> ftn -O3 -hpl=my_program.pl -c my_program_file1.f90
$> ftn -O3 -hpl=my_program.pl -c my_program_file1.f90
$> reveal my_program.pl my_program.ap2 &
```

- You can omit the *.ap2 and inspect only compiler feedback.
- Note that the profile_generate option disables most automatic compiler optimizations, which is why Cray recommends generating this data separately from generating the program_library file.
Visualize CCE’s Loopmark with Performance Profile
Visualize CCE’s Loopmark with Performance Profile (2)

Integrated message ‘explain support’
View Pseudo Code for Inlined Functions

- Inlined call sites marked
- Expand to see pseudo code
Scoping Assistance – Review Scoping Results

- Loops with scoping information are highlighted – red needs user assistance.
- Parallelization inhibitor messages are provided to assist user with analysis.
- User addresses parallelization issues for unresolved variables.
Scoping Assistance – User Resolves Issues

Use Reveal’s OpenMP parallelization tips

Click on variable to view all occurrences in loop
Scoping Assistance – Generate Directive

Reveal generates example OpenMP directive.
Optimisations for MPI
Rank Reordering

- Sometimes an MPI application is not well balanced

- The MPI library can reorder the ranks at runtime based on the setting of MPICH_RANK_REORDER_METHOD
Rank Placement

- Start with a list of nodes to run on

0: Round-robin placement
  - Sequential ranks are allocated one per node in sequence
  - Placement starts again on first node if we reach the last node

1: SMP-style placement (default)
  - Sequential ranks fill up each node in turn
  - Only then move on to the next node

2: Folded rank placement
  - Similar to round-robin placement
  - except each pass over node list is in the opposite direction

3: Custom ordering
  - The location of each rank in turn is specified in a list

Examples of these are shown on the next slide
  - For a simplified example of four cores per node
0: Round Robin Placement

```
Node 0
0 12
24 36

Node 1
1 13
25 37

Node 2
2 14
26 38

Node 3
3 15
27 39

Node 4
4 16
28 40

Node 5
5 17
29 41

Node 6
6 18
30 42

Node 7
7 19
31 43

Node 8
8 20
32 44

Node 9
9 21
33 45

Node 10
10 22
34 46

Node 11
11 23
35 47
```
1: SMP Placement (default)
2: Folded Placement
3: Custom Example

- **MPICH_RANK_REORDER=3** enables this
- **Ordering comes from file** `MPICH_RANK_ORDER`
  - comma separated ordered list
  - can optionally be condensed into hyphenated ranges
  - all ranks should be included in the list once and only once
- **Nodes are filled up SMP-style**
  - but not with sequential rank numbers
  - instead, take ranks sequentially from the `MPICH_RANK_ORDER` list

```
MPICH_RANK_ORDER: 0,1,4,5,2,3,6-9,12,13,10,11,14,15
```
MPI grid detection:

There appears to be point-to-point MPI communication in a 20 X 16 grid pattern. The 27.5% of the total execution time spent in MPI functions might be reduced with a rank order that maximizes communication between ranks on the same node. The effect of several rank orders is estimated below.

A file named MPICH_RANK_ORDER.Grid was generated along with this report and contains usage instructions and the Custom rank order from the following table.

<table>
<thead>
<tr>
<th>Rank Order</th>
<th>On-Node Bytes/PE</th>
<th>On-Node Bytes/PE% of Total</th>
<th>MPICH_RANK_REORDER_METHOD</th>
</tr>
</thead>
<tbody>
<tr>
<td>Custom</td>
<td>8.092e+09</td>
<td>75.00%</td>
<td>3</td>
</tr>
<tr>
<td>SMP</td>
<td>4.580e+09</td>
<td>42.45%</td>
<td>1</td>
</tr>
<tr>
<td>Fold</td>
<td>2.290e+08</td>
<td>2.12%</td>
<td>2</td>
</tr>
<tr>
<td>RoundRobin</td>
<td>0.000e+00</td>
<td>0.00%</td>
<td>0</td>
</tr>
</tbody>
</table>

When testing this the time went only down to 348 from 360 seconds, but approach might become important when scaling higher.
Further information from CrayPat

Metric-Based Rank Order:

When the use of a shared resource like memory bandwidth is unbalanced across nodes, total execution time may be reduced with a rank order that improves the balance. The metric used here for resource usage is: USER Time

For each node, the metric values for the ranks on that node are summed. The maximum and average value of those sums are shown below for both the current rank order and a Custom rank order that seeks to reduce the maximum value.

A file named MPICH_RANK_ORDER.USER_Time was generated along with this report and contains usage instructions and the Custom rank order from the following table.

<table>
<thead>
<tr>
<th>Rank Order</th>
<th>Maximum Value</th>
<th>Average Value</th>
<th>Max:Ave</th>
<th>Reduction in Max</th>
<th>Ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>Custom</td>
<td>3.491e+03</td>
<td>3.393e+03</td>
<td>1.03</td>
<td>8.77%</td>
<td></td>
</tr>
<tr>
<td>Current</td>
<td>3.827e+03</td>
<td>3.393e+03</td>
<td>1.13</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Rank reordering

- **Easy to experiment with**
  - defaults at least should be tested with every application…
  - CrayPat can help generate the reorder file

- **When might rank reordering be useful?**
  - If point-to-point communication consumes a significant fraction of program time and a load imbalance detected
    - e.g. for nearest-neighbour exchanges (see next slide)
  - Also shown to help for collectives (alltoall) on subcommunicators
  - Spread out I/O servers across nodes
  - If there is a good use case for exploiting the Intel hyperthreads

- **Have used this for I/O servers (NEMO) and radiation colocation (IFS)**