Tutorial A: Introduction to Parallel Tools.

Nicholas Allsopp
Cray Application Support at HLRS

Stuttgart, 11th October 2012
Using the installed tools on Hermit

- Several software packages are provided and supported by Cray
  - Programming Environment
  - Optimized Libraries
  - Profiler
  - ATP
- Others are provided by HLRS
  - Intel compiler
  - DDT
  - Threadspotter, …

- This tutorial is about using the Cray tools, especially the profiler CrayPat
Workflow for scaling your application

1. Identify Application and Science Worthy Problem
2. Get application ported to the XE6
3. Get your application to run
4. Instrument and run the application
5. Examine the Results
   a) Is there load imbalance?
   b) Is computation > 50% of the runtime?
   c) Is communication > 50% of the runtime?
   d) Is I/O > 50% of the runtime?

Repeat as needed
1st Step: Identify Application and Science Worthy Problem

- Formulate the problem
- The problem identified should make good science sense
  - No publicity stunts that are not of interest
- It should be a production style problem
  - Weak scaling
    - Finer grid as processors increase
    - Fixed amount of work when processors increase
  - Strong scaling
    - Fixed problem size as processors increase
    - Less and less work for each processor as processors increase

Think Bigger
Weak vs. Strong scaling

![Graph showing weak vs. strong scaling for BQCD on a lattice with 48, 48, 48, 96 elements at 8192 cores using CG only.](image)
2nd Step: Porting your application

To make porting easy, Cray's Programming Environment is based on modules:

- You decide which environment by selecting the PrgEnv-X module X=cray, pgi, intel, gnu
- By selecting the PE, you get all needed setup done for you: Include paths, Lib paths, Path to compilers, tools … There is no need to know where something is installed
- The default loaded modules at HLRS includes:
  - PrgEnv-cray (MPI, ACML, LIBSCI (Blas, Scalapack, …), ATP, CCE)
  - xtpe-interlagos
  - HLRS created modules for the workspace mechanism and mail system setup for the queuing system

There are other Cray supported modules available:

- Setup: craype-<machine> including support for GPU,
- Performance: hugepages, fftw v2 and v3, trilinos, PETsC ..
- IO: Netcdf, hdf5, iobuf …
- Profiling: perftools, papi, ddt
2nd Step: Porting your application

- **HLRS provided modules (selection):**
  - MPI for CCM mode: hpmpi
  - ISV: abaqus, lstc
  - Libs: Metis, ParMetis, PARPACK
  - Tools: inspector, periscope, scalasca, tau, threadspotter, vampir

- **CRA Y-HLRS provided modules (selection)**
  - Applications: CP2K, CPMD
  - Tools: cmake (with support for XE6)

- **Software is added on Hermit as a need is identified**
2nd Step : PrgEnv, Easy to use

- After your module setup, you use compiler wrappers to compile and link your code
  - ftn for Fortran
  - cc for C
  - CC for C++
- The box shows a very easy example
- Notice: No need for compiler flags showing where MPI is located

This stays true for all modules. As soon a module is loaded, the compiler wrappers will find the headers, modules and libs.
3rd Step: Run your code

```bash
#!/bin/bash
PBS -l mppwidth=1024
PBS -l mppnppn=16
PBS -l mppdepth=2
PBS -l feature=mem64gb
PBS –N jobname
PBS -l walltime=5:0:0

cd $PBS_O_WORKDIR
export OMP_NUM_THREADS=1

export MPICH_VERSION_DISPLAY=1
export MPICH_CPUMASK_DISPLAY=1
export MPICH_ENV_DISPLAY=1

module load craype-hugepages2M

aprun –n 1024 –N 16 –d $OMP_NUM_THREADS ./a.out
```
3rd Step: Bug hunting

- Most likely you will need some tools for finding the location of bugs while developing your application.

- Debugger: At HLRS the DDT debugger from Allinea is installed
  - The Totalview debugger is also supported by Cray. You might find it on other systems.
    Not covered in this talk.

- The Cray tools ATP (Abnormal Termination Processing).
  - allows you to find the location of the problem by displaying tracebacks of your application.
  - It can also be used to monitor your application while running (deadlock finding).
ATP

- **To use:**
  - Load the module atp (default on Hermit)
  - Link your application
  - Set the environment variable ATP_ENABLED=1
  - Run your application

- If your code crashes, you will find a file named atpMergedBT.dot in your directory. You can view it with `statview` after the module stat is loaded. This replaces having <ntasks> core dumps

- You can also issue a “STAT <PID>” to get a current traceback snapshot of your running application
Part of ‘statview’ window of wrf
4th step: Instrument the application

- **Instrument the application**
  - Run the production case
    - Run long enough that the initialization does not use > 1% of the time
    - Run with normal I/O
  - Use Craypat’s APA
    - First gather sampling for line number profile
    - Second gather instrumentation (-g mpi,io)
      - Hardware counters
      - MPI message passing information
      - I/O information

- `> load module`
- `> make`
- `> pat_build -O apa a.out` Execute
- `> pat_report *.xf`
- `> pat_build -O *.apa` Execute
CrayPat is really a cover all tool

It can collect information about almost everything and the user still have full control => it is quite complex
The 3 basic man pages (pat, pat_build, pat_report) have over 3000 lines

To help the user to get started fast, it supports an “automatic program analysis” (apa) process.
- Not only is this a fast way of getting started, for most users the provided information is what they want

In addition it also has a graphical interface for analyzing the data called app2 (Cray Apprentice2)
CrayPat: basic information

- CrayPat supports two categories of performance analysis experiments:
  - tracing experiments, which count some event such as the number of times a specific system call is executed, and
  - asynchronous (sampling) experiments, which capture values at specified time intervals or when a specified counter overflows. Tracing is more accurate, sampling has less overhead

- Typically, after using
  - `pat_build` to instrument a program, users set environment variables to control runtime data collection and run the instrumented program, and then use either
  - `pat_report` or `app2` (Cray Apprentice2) to view the resulting report.
CrayPat : Using APA (2 steps profiling)

- First load the module: `module load perftools`
- Secondly: Recompile and relinking your application
  - This will add some debugging information to the object files

- Build the instrumented version: `pat_build -O apa ./a.out`
  - This will create a new executable using sampling for the data collection
- Run the instrumented version, when finished you will get some *.xf files or a new directory containing the *.xf files
  - Output location depends on number of tasks you run with.
- Run `pat_report` on the *.xf files. This will create 2 files:
  - A *.ap2 file. It contains all the information from the *.xf files and is CrayPat version independent. You can now delete the *.xf files
  - A *.apa file with information for the next step (see next slide)

- Reinstrument the executable for tracing using the ap2 file
  - `pat_build -O <file.apa>`
  - This will create an instrumented executable using tracing for the collection, but only on the relevant parts (based on the sampling analyze)
- Run the tracing executable
- Use `pat_report` on the newly created *.xf files
CrayPat : Sampling analysis output (1/3)

CrayPat/X: Version 5.3.0 Revision 8530 (xf 8240) 12/09/11 10:11:21

Number of PEs (MPI ranks): 320

Numbers of PEs per Node: 32  PEs on each of 10  Nodes

Numbers of Threads per PE: 1

Number of Cores per Socket: 16

Execution start time: Fri Aug 17 11:25:07 2012

System type and speed: x86_64 2300 MHz

Current path to data file:
/univ_1/ws1/ws/hpcander-wrf-0/WRFV3/run/wrf.exe+pat+20975-939s.ap2 (RTS)
Notes for table 1:

Table option:
- O samp_profile

Options implied by table option:
- d sa%@0.95,sa,imb_sa,imb_sa% -b gr,fu,pe=HIDE

Options for related tables:
- O samp_profile+src

The Total value for Samp is the sum for the Group values.
The Group value for Samp is the sum for the Function values.
The Function value for Samp is the avg for the PE values.
(To specify different aggregations, see: pat_help report options s1)

This table shows only lines with Samp% > 0.95.
(To set thresholds to zero, specify: -T)

Percentages at each level are of the Total for the program.
(For percentages relative to next level up, specify:
- s percent=r[elative])
### Table 1: Profile by Function

<table>
<thead>
<tr>
<th>Samp%</th>
<th>Samp</th>
<th>Imb.</th>
<th>Imb.</th>
<th>Group</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Samp</td>
<td>Samp%</td>
<td>Function</td>
<td></td>
</tr>
<tr>
<td>-------</td>
<td>------</td>
<td>-------</td>
<td>----------</td>
<td></td>
</tr>
<tr>
<td>100.0%</td>
<td>28024.1</td>
<td>--</td>
<td>--</td>
<td>Total</td>
</tr>
</tbody>
</table>
|-------------------------------------------------- ------------------------------------
| 52.3% | 14655.1 | -- | -- | USER |
|--|--|------------------------------------|
| 6.9% | 1943.9 | 209.1 | 9.7% | module_advec_em_advect_scalar_pd_ |
| 4.1% | 1162.1 | 592.9 | 33.9% | f_pack_int_ikj_ |
| 3.2% | 888.8 | 936.2 | 51.5% | module_mp_morr_two_moment_morr_two_moment_micro_ |
| 3.1% | 860.8 | 212.2 | 19.8% | module_advec_em_advect_scalar_ |
|-------------------------------------------------- ------------------------------------
| 35.5% | 9961.6 | -- | -- | MPI |
|-------------------------------------------------- ------------------------------------
| 24.1% | 6761.4 | 3424.6 | 33.7% | MPI_Wait |
| 9.5% | 2670.1 | 130.9 | 4.7% | MPI_Bcast |
|-------------------------------------------------- ------------------------------------
| 12.2% | 3407.4 | -- | -- | ETC |
The apa file

> cat wrf.exe+pat+20975-939s.apa
# You can edit this file, if desired, and use it
# to reinstrument the program for tracing like this:
#
#    pat_build -O wrf.exe+pat+20975-939s.apa
...
#    HWPC group to collect by default.
-Drtenv=PAT_RT_HWPC=23  # FP, D1, D2, and TLB metrics.
...
#    Libraries to trace.
-g mpi,stdio,sysio
...
#    User-defined functions to trace, sorted by % of samples.
# 6.94% 82973  bytes
   -T module_advect_em_advect_scalar_pd_

# 4.15% 552  bytes
#    -T f_pack_int_ikj_

# 3.05% 536  bytes
#    -T f_unpack_int_ikj_
Tracing profile

- After the sync profiling we can now do the tracing using the *.apa file:
  
  ```bash
  pat_build -O <my.apa.file>
  ```

- This new executable is run and `pat_report` used on the new profiling data created.

```
CrayPat/X: Version 5.3.0 Revision 8530 (xf 8240) 12/09/11 10:11:21
Number of PEs (MPI ranks): 320
Numbers of PEs per Node: 16 PEs on each of 20 Nodes
Numbers of Threads per PE: 1
Number of Cores per Socket: 16
System type and speed: x86_64 2300 MHz
Current path to data file:
```
Table 1 from the tracing analysis general overview, our starting point for optimization

<table>
<thead>
<tr>
<th>Time%</th>
<th>Time</th>
<th>Imb. Time</th>
<th>Imb. Time</th>
<th>Calls</th>
<th>Group</th>
</tr>
</thead>
<tbody>
<tr>
<td>100.0%</td>
<td>328.724848</td>
<td>--</td>
<td>--</td>
<td>9360355.6</td>
<td>Total</td>
</tr>
<tr>
<td>60.4%</td>
<td>198.524711</td>
<td>--</td>
<td>--</td>
<td>4045677.8</td>
<td>USER</td>
</tr>
<tr>
<td>8.2%</td>
<td>26.94613</td>
<td>9.001085</td>
<td>25.1%</td>
<td>1.0</td>
<td>wrf_</td>
</tr>
<tr>
<td>6.4%</td>
<td>21.044201</td>
<td>52.429400</td>
<td>71.6%</td>
<td>557568.0</td>
<td>morr_two_moment_micro$module...</td>
</tr>
<tr>
<td>5.0%</td>
<td>16.469715</td>
<td>0.547850</td>
<td>3.2%</td>
<td>4320.0</td>
<td>solve_em_LOOP@li.883</td>
</tr>
<tr>
<td>4.6%</td>
<td>15.176664</td>
<td>3.921010</td>
<td>20.6%</td>
<td>14400.0</td>
<td>advect_scalar_pd$module_advect_em_</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>... + 15 more routines with &gt;1% Time%</td>
</tr>
<tr>
<td>27.5%</td>
<td>90.238221</td>
<td>--</td>
<td>--</td>
<td>3456959.0</td>
<td>MPI</td>
</tr>
<tr>
<td>24.9%</td>
<td>82.000491</td>
<td>37.697216</td>
<td>31.6%</td>
<td>532803.5</td>
<td>MPI_Wait</td>
</tr>
<tr>
<td>1.4%</td>
<td>4.631645</td>
<td>0.332811</td>
<td>6.7%</td>
<td>53600.0</td>
<td>MPI_SCATTERV</td>
</tr>
<tr>
<td>10.2%</td>
<td>33.453203</td>
<td>--</td>
<td>--</td>
<td>66162.0</td>
<td>MPI_SYNC</td>
</tr>
<tr>
<td>8.0%</td>
<td>26.371888</td>
<td>25.857180</td>
<td>98.0%</td>
<td>6545.0</td>
<td>MPI_Bcast(sync)</td>
</tr>
<tr>
<td>1.1%</td>
<td>3.692381</td>
<td>3.687237</td>
<td>99.9%</td>
<td>3.0</td>
<td>mpi_bcast_(sync)</td>
</tr>
</tbody>
</table>
5th-a step: Application is load imbalanced

- **What is causing the load imbalance**
  - Computation
    - Is decomposition appropriate?
    - Would RANK_REORDER help?
  - Communication
    - Is decomposition appropriate?
    - Would RANK_REORDER help?
    - Are receives pre-posted
    - Any All-to-1 communication

- **OpenMP may help**
  - Able to spread workload with less overhead
    - Large amount of work to go from all-MPI to Hybrid
    - Must accept challenge to OpenMP-ize large amount of code

- **Go back to instrumentation step**
  - Re-gather statistics

Need Craypat reports

Is SYNC time due to computation?
Motivation for Load Imbalance Analysis

- **Increasing system software and architecture complexity**
  - Current trend in high end computing is to have systems with tens of thousands of processors
    - This is being accentuated with multi-core processors

- **Applications have to be very well balanced in order to perform at scale on these MPP systems**
  - Efficient application scaling includes a balanced use of requested computing resources

- **Desire to minimize computing resource “waste”**
  - Identify slower paths through code
  - Identify inefficient “stalls” within an application
CrayPat: Performance Measurement and Analysis

- **Load imbalance**
  - Identifies computational code regions and synchronization calls that could benefit most from load balance optimization (some processes have less work than others, some are waiting longer on barriers, etc)
  - Estimates savings if corresponding section of code were balanced
  - MPI sync time (determines late arrivers to global communication)
  - MPI rank placement suggestions (maximize on-node communication)
  - Imbalance metrics (user functions, MPI functions, OpenMP threads)
5th-c : Communication is Major Bottleneck

- What is causing the Bottleneck?
  - Collectives
    - MPI_ALLTOALL
    - MPI_ALLREDUCE
    - MPI_REDUCE
    - MPI_VGATHER/MPI_VSCATTER
  - Point to Point
    - Are receives pre-posted
      - Don’t use MPI_SENDRECV
    - What are the message sizes
      - Small – Combine
      - Large – divide and overlap

- OpenMP may help
  - Able to spread workload with less overhead
    - Large amount of work to go from all-MPI to Hybrid
      - Must accept challenge to OpenMP-ize large amount of code

- Go back to instrumentation step
  - Re-gather statistics
Rank Placement

- The default ordering can be changed using the following environment variable:
  - MPICH_RANK_REORDER_METHOD

- These are the different values that you can set it to:
  - 0: Round-robin placement – Sequential ranks are placed on the next node in the list. Placement starts over with the first node upon reaching the end of the list.
  - 1: (DEFAULT) SMP-style placement – Sequential ranks fill up each node before moving to the next.
  - 2: Folded rank placement – Similar to round-robin placement except that each pass over the node list is in the opposite direction of the previous pass.
  - 3: Custom ordering. The ordering is specified in a file named MPICH_RANK_ORDER.
Rank Placement

- **When is this useful?**
  - Point-to-point communication consumes a significant fraction of program time and a load imbalance detected
  - Also shown to help for collectives (alltoall) on subcommunicators
  - Spread out IO across nodes
What type of I/O?
- One writer – large files
  - Stripe across most OSTs
- All writers – small files
  - Stripe across one OST
- MPI-I/O?
  - Try using subset of writers

Go back to instrumentation step
- Re-gather statistics

OST = Object Storage Target
How can IO be done: Spokesperson

- One process performs I/O.
  - Data Aggregation or Duplication
  - Limited by single I/O process.
- Easy to program
- Pattern does not scale.
  - Time increases linearly with amount of data.
  - Time increases with number of processes.
- Care has to be taken when doing the "all to one"-kind of communication at scale
- Can be used for a dedicated IO Server (not easy to program)
How can IO be done: Multiple Writers – Multiple Files

- All processes perform I/O to individual files.
  - Limited by file system.
- Easy to program
  - Requires job to always run on the same number of cores
- Pattern does not scale at large process counts.
  - Number of files creates bottleneck with metadata operations.
  - Number of simultaneous disk accesses creates contention for file system resources.
How can IO be done: Collective IO to single or multiple files

- Aggregation to a processor in a group which processes the data.
  - Serializes I/O in group.
- I/O process may access independent files.
  - Limits the number of files accessed.
- Group of processes perform parallel I/O to a shared file.
  - Increases the number of shares to increase file system usage.
  - Decreases number of processes which access a shared file to decrease file system contention.
- Cray MPI-IO does this by default
MPI-IO Interaction with Lustre

- Included in the Cray MPT library.
- Environmental variable used to help MPI-IO optimize I/O performance.
  - MPICH_MPIIO_CB_ALIGN Environmental Variable. (Default 2)
    - sets collective buffering behavior
  - MPICH_MPIIO_HINTS Environmental Variable
  - Can set striping_factor and striping_unit for files created with MPI-IO.
  - If writes and/or reads utilize collective calls, collective buffering can be utilized (romio_cb_read/write) to approximately stripe align I/O within Lustre.
  - “man mpi” for all env. variables

- HDF5 and NETCDF are both implemented on top of MPI-IO and thus also uses the MPI-IO environment variables.
Visualize CCE’s Loopmark with Performance Profile

Performance feedback

Loopmark and optimization annotations

Compiler feedback
Visualize CCE’s Loopmark with Performance Profile (2)

The compiler unrolled the loop. Unrolling creates a number of copies of the loop body. When unrolling an outer loop, the compiler attempts to fuse replicated inner loops – a transformation known as unroll-and-jam. The compiler will always employ the unroll-and-jam mode when unrolling an outer loop. literal outer loop unrolling may occur when unrolling to satisfy a user directive (pragma).

This message indicates that unroll-and-jam was performed with respect to the identified loop. A different message is issued when literal outer loop unrolling is performed, as this transformation is far less likely to be beneficial.

For sake of illustration, the following contrasts unroll-and-jam with literal outer loop unrolling.

The literal outer unroll code performs the same sequence of memory operations as the original nest, while the unroll-and-jam transformation interleaves operations from outer loop iterations. The compiler employs literal outerloop unrolling only when the data dependencies in the loop, or a control flow impediment, prevent fusion of the replicated inner loops. Literal outer loop unrolling is generally not desirable. It is provided to ensure expected behavior and for those rare instances where the user has determined that it is beneficial.
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
Questions

Cray XE6 Optimization workshop
5\textsuperscript{th} -> 8\textsuperscript{th} November 2012
https://fs.hlrs.de/projects/par/events/2012/parallel_prog_2012/XE6-2.html