User Orientation on XC 40 system

21st , Feb, 2018 IITM, Pune

RAVITEJA K

Applications Analyst, Cray

E Mail : raviteja@cray.com

Vision

- **Cray systems are designed to be High Productivity as well as High Performance Computers**
- **The Cray Programming Environment (PE) provides a simple consistent interface to users and developers.**
  - Focus on improving scalability and reducing complexity
- **The default Programming Environment provides:**
  - the highest levels of application performance
  - a rich variety of commonly used tools and libraries
  - a consistent interface to multiple compilers and libraries
  - an increased automation of routine tasks
- **Cray continues to develop and refine the PE**
  - Frequent communication and feedback to/from users
  - Strong collaborations with third-party developers

1. Compiler Optimizations

2. Cray Scientific Libraries

3. Performance Tuning and Scaling

4. Cray Tools

Compiler Optimizations

Vectorization (1):

- **Hardware Perspective:**
  - Specialized instructions, registers, or functional units to allow in-core parallelism for operations on arrays (vectors) of data.
- **Compiler Perspective:**
  - Determine how and when it is possible to express computations in terms of vector instructions
- **User Perspective:**
  - Determine how to write code in a manner that allows the compiler to deduce that vectorization is possible

Source : http://www.cac.cornell.edu

Vectorization (2):

- **Goal:**
  - Parallelize computations over vector arrays
- **Two major approaches:**
  - Pipelining :
    - Several different tasks executing simultaneously
  - SIMD (Single Instruction Multiple Data) :
    - Many instances of a single task executing simultaneously

Vectorization(3):

- **Vector Units :**
  - Performs parallel floating/integer point operations on dedicate SIMD units
- **Intel Vector units:**
  - Pentium :
  - MMX :
  - SSE :
  - AVX :
  - AVX2 :

Vectorization (4):

- **Vector Registers :**

  Floating Pointer

  Single Precision : 32 bit 

  Double Precision : 64 bit 

  - SSE : 
  - AVX : 
  - AVX2 : 

Vectorization (5):

Think vectorization in terms of loop unrolling

- Unroll N interactions of loop, where N elements of data array fit into vector register

```
for ( i=0; i<N; i++)
      A[ i ] = B[ i ] + C[ i ];
```

```
for ( i=0; i<N; i+=4)   {                    // A,B and C are DP; in case of AVX
      A[ i ] = B[ i ] + C[ i ];
      A[ i+1 ] = B[ i+1 ] + C[ i+1 ];
      A[ i+2 ] = B[ i+2 ] + C[ i+2 ];
      A[ i+3 ] = B[ i+3 ] + C[ i+3 ];
}
```

Vectorization (6):

Loop dependency?

```
for ( i=1; i<N; i++)
     A[ i ] = A[ i -1 ] + B[ i ];
```

A [0, 1, 2, 3, 4]
B [5, 6, 7, 8, 9]

Sequential operation looks like:

A[1] = A[0] + B [1] : 0 + 6 = 6

A[2] = A[1] + B [2] : 6 + 7 = 13

A[3] = A[2] + B [3] : 13 + 8 = 21

A[4] = A[3] + B [4] : 21+ 9 = 30

Updated A [0, 6, 13, 21, 30]

Vectorization (7):

Loop dependency?

A [0, 1, 2, 3, 4] and B [5, 6, 7, 8, 9]

```
for ( i=1; i<N; i++)
        A[ i ] = A[ i -1 ] + B[ i ];
```

Vector operation looks like:

A[i] = A [i – 1] + B [i]

A [1, 2, 3, 4] = A [0, 1, 2, 3] + B [6, 7, 8, 9]

= A [0, 6, 8, 10, 12]

A [0, 6, 13, 21, 30] ≠ A [0, 6, 8, 10, 12]

Vectorization (8):

- **-hvectorN (cc/CC) :** where N=0…3, specify the level of automatic vectorizing to be performed. Vectorization results in significant performance improvements with a small increase in object code size. Vectorization directives are unaffected by this option

  - 0: No automatic vectorization
  - 1: Specifies conservative vectorization. Loop nests are restructured. No vectorizations that might create false exceptions are performed. Results may differ slightly from results obtained when N=0 is specified because of vector reductions
  - 2: (Default) Specifies moderate vectorization. Characteristics include moderate compile time and size. Loop nests are restructured
  - 3: Specifies aggressive vectorization. Loop nests are restructured. Vectorizations that might create false exceptions in rare cases may be performed

Inlining:

- Inlining is the process of replacing a call with the subprogram or function itself

Eg :

```
int call1 (int x, int y)
{
  return x + y;
}

int call2 (int x, int y)
{
  return call1(x, -y);
}
```

Replaced as

```
int call2 (int x, int y)
{
  return x-y;
}
```

Using Compiler Feedback

- **Compilers can generate annotated listing of your source code indicating important optimizations. Useful for targeted use of compiler flags.**
- **CCE**
  - `ftn -rm`
  - `{cc,CC} -hlist=a`
- **Intel**
  - `ftn/cc -opt-report 3 -vec-report6`
  - If you want this into a file: add `-opt-report-file=filename`
  - See `ifort --help reports`
- **GNU**
  - `-ftree-vectorizer-verbose=9`

Compiler feedback: Loopmark

- For example, with the Cray compiler

```
%%%    L o o p m a r k   L e g e n d    %%%
Primary Loop Type      Modifiers
------- ---- ----      ---------
A  - Pattern matched   a - vector atomic memory operation
                       b – blocked
C  - Collapsed         f – fused
D  - Deleted           i – interchanged
E  - Cloned            m - streamed but not partitioned
I  - Inlined           p - conditional, partial and/or computed
M  - Multithreaded     r – unrolled
P  - Parallel/Tasked   s – shortloop
V  - Vectorized        t - array syntax temp used
                       w - unwound
```

Compiler feedback: Loopmark (cont.)

```
29.  b-------<  do i3=2,n3-1
30.  b b-----<     do i2=2,n2-1
31.  b b Vr--<        do i1=1,n1
32.  b b Vr            u1(i1) = u(i1,i2-1,i3) + u(i1,i2+1,i3)
33.  b b Vr    *         + u(i1,i2,i3-1) + u(i1,i2,i3+1)
34.  b b Vr            u2(i1) = u(i1,i2-1,i3-1) + u(i1,i2+1,i3-1)
35.  b b Vr    *         + u(i1,i2-1,i3+1) + u(i1,i2+1,i3+1)
36.  b b Vr-->        enddo
37.  b b Vr--<        do i1=2,n1-1
38.  b b Vr            r(i1,i2,i3) = v(i1,i2,i3)
39.  b b Vr    *         - a(0) * u(i1,i2,i3)
40.  b b Vr    *         - a(2) * ( u2(i1) + u1(i1-1) + u1(i1+1) )
41.  b b Vr    *         - a(3) * ( u2(i1-1) + u2(i1+1) )
42.  b b Vr-->        enddo
43.  b b----->     enddo
44.  b------->  enddo
```

Compiler Feedback: Loopmark (cont.)

```
 ftn-6289 ftn: VECTOR File = resid.f, Line = 29
  A loop starting at line 29 was not vectorized because a recurrence was found
on "U1" between lines 32 and 38.
ftn-6049 ftn: SCALAR File = resid.f, Line = 29
  A loop starting at line 29 was blocked with block size 4.
ftn-6289 ftn: VECTOR File = resid.f, Line = 30
  A loop starting at line 30 was not vectorized because a recurrence was found
on "U1" between lines 32 and 38.
ftn-6049 ftn: SCALAR File = resid.f, Line = 30
  A loop starting at line 30 was blocked with block size 4.
ftn-6005 ftn: SCALAR File = resid.f, Line = 31
  A loop starting at line 31 was unrolled 4 times.
ftn-6204 ftn: VECTOR File = resid.f, Line = 31
  A loop starting at line 31 was vectorized.
ftn-6005 ftn: SCALAR File = resid.f, Line = 37
  A loop starting at line 37 was unrolled 4 times.
ftn-6204 ftn: VECTOR File = resid.f, Line = 37
  A loop starting at line 37 was vectorized.
```

What did that loopmark note mean? Use “explain” for more information

% explain ftn-6289

VECTOR: A loop starting at line %s was not vectorized because a recurrence was found on "var" between lines num and num.

Scalar code was generated for the loop because it contains a linear recurrence. The following loop would cause this message to be issued:

```
DO I = 2,100
   B(I) = A(I-1)
   A(I) = B(I)
ENDDO
```

“explain” utility works for any Cray PE messages, e.g., ftn-*, cc-*, ld-*

Some Cray, Intel, and GNU compiler flags

Feature	Cray	Intel	GNU
Listing	-ra (“report all”) or -rmo (“loop Mark” and “Opts used”)	-list -vec-report3 -opt-report -opt-report-file=name	-fdump-tree-all
Diagnostic	(produced by -ra)	-help diagnostic	-Wall (and other opts)
Free format	-f free	-free	-ffree-form
Preprocessing	-eZ	-P –fpp (Fortran)	-cpp
Suggested Optimization	-O2 (default)	-O3 –xAVX	-O2 -mavx -ftree-vectorize -ffast-math -funroll-loops
Aggressive Optimization	-O3,fp3	-ffast-math -funroll-loops -ftree-vectorize –xAVX	-Ofast -mavx -funroll-loops
Variables size	-s real64 -s integer64	-real-size 64 -integer-size 64	-fdefault-real-8 -fdefault-integer-8
Byte swap	-h byteswapio	-convert big_endian	-fconvert=swap
Enab. OpenMP	(default)	-openmp	-fopenmp

Recommended compiler optimization levels

- **Cray compiler**
  - The default optimization level (i.e. no flags) is equivalent to **`-O3`** of most other compilers. CCE optimizes rather aggressively by default, but this is also most thoroughly tested configuration
  - Try with **`-O3 -hfp3`** (also tested this thoroughly)
    - `-hfp3` gives you a lot more floating point optimization, esp. 32-bit
    - In case of precision errors, try a lower `–hfp<number>` (`-hfp1` first, only `-hfp0` if absolutely necessary)
- **GNU compiler**
  - Almost all HPC applications compile correctly with using `-O3`, so do that instead of the cautious default.
  - `-ffast-math` may give some extra performance
- **Intel compiler**
  - The default optimization level (equal to `-O2`) is safe.
  - Try with **`-O3`**. If that works still, you may try with **`-Ofast -fp-model fast=2`**
- **Use `-craype-verbose` flag to `{cc,CC,ftn}` to show options**

Inlining & inter-procedural optimization

- **Cray compiler**
  - Inlining within a file is enabled by default.
  - Command line options **`-OipaN`** (ftn) and **`-hipaN`** (cc/CC) where N=0..4, provides a set of choices for inlining behavior
    - 0 disables inlining, 3 is the default, 4 is even more elaborate
  - The **`-Oipafrom=`** (ftn) or **`-hipafrom=`** (cc/CC) option instructs the compiler to look for inlining candidates from other source files, or a directory of source files.
  - The **`-hwp`** combined with **`-h pl=…`** enables whole program automatic inlining.
- **GNU compiler**
  - Quite elaborate inlining enabled by **`-O3`**
- **Intel compiler**
  - Inlining within a file is enabled by default
  - Multi-file inlining enabled by the flag **`-ipo`**

Loop transformations

- **Cray compiler**
  - Most useful techniques in their aggressive state already by default
  - One may try to improve loop restructuration for better vectorization with **`-h vector3`**
- **GNU compiler**
  - Loop blocking (aka tiling) with **`-floop-block`**
  - Loop unrolling **`-funroll-loops`** or **`-funroll-all-loops`**
- **Intel compiler**
  - Loop unrolling with **`-funroll-loops`** or **`-unroll-aggressive`**

Directives for the Cray Compiler

- If you see from the compiler feedback that a loop has not been blocked, unrolled, or vectorized but you are convinced that it should be, you can use compiler directives instead of rising the optimization level –O…
- Cray compiler supports a full and growing set of directives and pragmas, e.g.
  - `!dir$ concurrent`
  - `!dir$ ivdep`
  - `!dir$ interchange`
  - `!dir$ unroll`
  - `!dir$ loop_info [max_trips] [cache_na]`
  - `!dir$ blockable`
- More information given in
  - man directives
  - man loop_info

```
!dir$ blockable(j,k)
!dir$ blockingsize(16)
    do k = 6, nz-5
       do j = 6, ny-5
          do i = 6, nx-5
          ! stencil
          end do
       end do
    end do
```

Why are CCE's results sometimes different to other compilers?

- **Cray expect applications to be conformant to language requirements**
  - This include not over-indexing arrays, no overlap between Fortran subroutine arguments, and so on
  - Applications that violate these rules may lead to incorrect results or segmentation faults
  - Note that languages do not require left-to-right evaluation of arithmetic operations, unless fully parenthesized
    - This can often lead to numeric differences between different compilers
    - Some applications expect left-to-right evaluation
    - Use **-hadd_paren** to add automatically parenthesis to select associative operations (+,–,*). Default is **-hnoadd_paren**
- **We are also fairly aggressive at floating point optimizations that violate IEEE requirements**
  - Use **-hfp[0-4]** flag to control that

About reproducibility

- **CCE compilers guarantee that repeated runs with same number of ranks and threads will give identical results. This is not the case for all other compilers. However:**
- **Results can vary with the number of ranks or threads**
  - Use **-hflex_mp=option** to control the aggressiveness of optimizations which may affect floating point and complex repeatability when application requirements require identical results when varying the number of ranks or threads.
  - **option** in order from least aggressive to most is:
    - intolerant: has the highest probability of repeatable results, but also has the highest performance penalty
    - strict: uses some safe optimizations, with high probability of repeatable results.
    - conservative: uses more aggressive optimization and yields higher performance than intolerant, but results may not be sufficiently repeatable for some applications
    - default: uses more aggressive optimization and yields higher performance than conservative, but results may not be sufficiently repeatable for some applications
    - tolerant: uses most aggressive optimization and yields highest performance, but results may not be sufficiently repeatable for some applications

Recommended for bit reproducibility

- **Start from this set**

  -hflex_mp=conservative –hfp1 –hadd_paren

- **Please note:**
  - We only strive to maintain bit reproducibility for applications that are designed correctly to be bit reproducible. The compiler cannot make a non bitrep code reproducible.

Summary

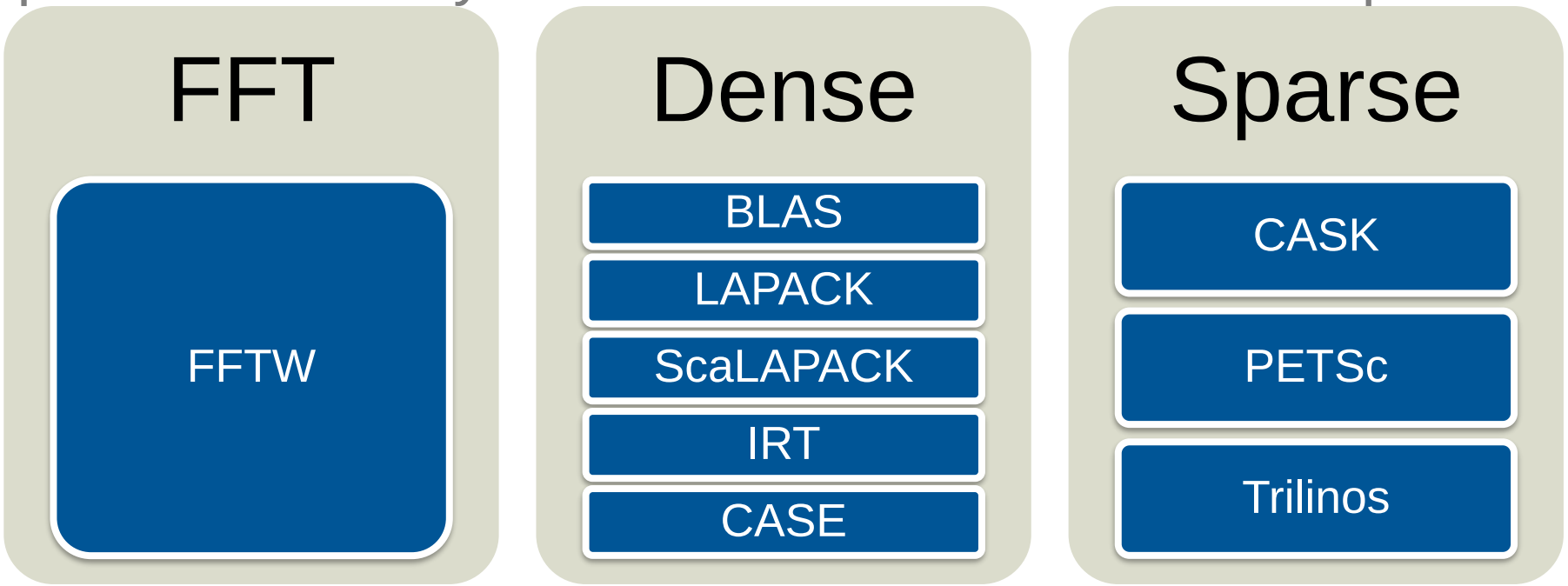
- **Four compiler environments available on the XC:**
  - Cray (PrgEnv-cray is the default)
  - Intel (PrgEnv-intel)
  - GNU (PrgEnv-gnu)
  - PGI (PrgEnv-pgi)
  - All of them accessed through the wrappers ftn, cc and CC – just do module swap to change a compiler or a version.
- **There is no universally fastest compiler**
  - Performance strongly depends on the application (even input)
  - We try however to excel with the Cray Compiler Environment
  - If you see a case where some other compiler yields better performance, let us know!
- **Compiler flags do matter**
  - be ready to spend some effort for finding the best ones for your application.
  - More information is given at the end of this presentation.

Cray Scientific Libraries

Overview

Cray Scientific Libraries

- **Large variety of standard libraries available via modules**
  - Optimized for Cray Hardware and also for Haswell processor.



IRT – Iterative Refinement Toolkit
CASK – Cray Adaptive Sparse Kernels
CASE – Cray Adaptive Simplified Eigensolver

What makes Cray libraries special

1. **Node performance**
   - Highly tuned routines at the low-level (ex. BLAS)
2. **Network performance**
   - Optimized for network performance
   - Overlap between communication and computation
   - Use the best available low-level mechanism
   - Use adaptive parallel algorithms
3. **Highly adaptive software**
   - Use auto-tuning and adaptation to give the user the known best (or very good) codes at runtime
4. **Productivity features**
   - Simple interfaces into complex software

Library Usage Overview

- **LibSci**
  - Includes BLAS, CBLAS, BLACS, LAPACK, ScaLAPACK
  - Module is loaded by default (`man libsci`)
  - Threading used within LibSci (OMP_NUM_THREADS). If you call within a parallel region, single thread used. More info later on.
- **FFTW**
  - `module load fftw` and `man fftw`
- **PETSc**
  - `module load cray-petsc{-complex}` and `man intro_petsc`
- **Trilinos**
  - `module load cray-trilinos` and `man intro_trilinos`
- **Third Party Scientific Libraries**
  - `module load cray-tpsl` (use online documentation)
- **Iterative Refinement Toolkit (IRT) through LibSci.**
  - man intro_irt
- **Cray Adaptive Sparse Kernels (CASK) are used in cray-petsc and cray-trilinos (transparent to the developer).**

Third party Scientific Libraries (cray-tpsl)

- TPSL (Third Party Scientific Libraries) contains a collection of outside mathematical libraries that can be used with PETSc and Trilinos.
- This module will increase the flexibility of PETSc and Trilinos by providing users with multiple options for solving problems in dense and sparse linear algebra.
- The cray-tpsl module is automatically loaded when PETSc or Trilinos is loaded. The libraries included are MUMPs, SuperLU, SuperLU_dist, ParMetis, Hypre, Sundials, and Scotch.

Check you got the right library!

- Add options to the linker to make sure you have the correct library loaded.
- `-Wl` adds a command to the linker from the driver
- You can ask for the linker to tell you where an object was resolved from using the –y option.
  - E.g. `-Wl,-ydgemm_` (notice the '_' at the end of the name)

```
.//main.o: reference to dgemm_
/opt/xt-libsci/11.0.05.2/cray/73/mc12/lib/libsci_cray_mp.a(dgemm.o):
definition  of dgemm_
```

Note: do not explicitly link "-lsci". This will not be found from libsci 11+ and means a single core library for 10.x.

Threading for BLAS and LAPACK

- **LibSci is compatible with OpenMP**
  - Control the number of threads to be used in your program using `OMP_NUM_THREADS`, e.g. `export OMP_NUM_THREADS=…`
  - Then run with `aprun … -d $OMP_NUM_THREADS …`
- **What behavior you get from the library depends on your code**
  1. No threading in code
     - The BLAS call will use `OMP_NUM_THREADS` threads
  2. Threaded code, outside parallel regions
     - The BLAS call will use `OMP_NUM_THREADS` threads
  3. Threaded code, inside parallel regions
     - The BLAS call will use a single thread
  - Threaded LAPACK works exactly the same as threaded BLAS
  - Anywhere LAPACK uses BLAS, those BLAS can be threaded.
  - Some LAPACK routines are threaded at the higher level

Intel MKL

- **The Intel Math Kernel Libraries (MKL) is an alternative to LibSci**
  - Features tuned performance for Intel CPUs as well
- **Linking quite complicated, but the Intel MKL Link Line Advisor can tell you what to add to your link line**
  - http://software.intel.com/sites/products/mkl/
- **Using MKL together with the Intel compilers (PrgEnv-intel) is usually straightforward. Simply add `-mkl` to your compile and linker options. You will get the following**

```
Warning:
        libraries from PE_LIBSCI will be ignored
        because they conflict with -mkl.
```

Performance Tuning and Scaling

Frequent Scenarios

- **Frequent Scenarios**
- **General Remarks on Optimization**
- **Possible Bottlenecks and Overview of Remedies.**
- **Very short Review**
  - Hardware
  - Job submission

Frequent Scenarios

- **Code has been recently ported to the Cray XC system**
  - Previously running on a smaller cluster at the institute. Already using MPI and/or OpenMP.
  - Trying to run on larger amount of nodes through either strong or weak scaling.
- **Code was subjected to a significant change in numeric and science.**
  - After notable amount of debugging the results are as expected.
- **Code is running in production since a while.**
  - Researchers are happy with the results and can live with time to solution.
- **In all these scenarios the performance plays a minor role.**
  - Correctness of results and the ability to run large cases thanks to large memory and storage has priority.

General Remarks on Optimization

- **But performance does matter**
  - Solve the same problem in less time on same or more resources.
  - Solve larger problem or more problems in approx. the same time on more resources.
  - It makes sense to plan some time for optimization besides development.
- **Performance is usually associated to FLOPs/sec.**
  - But what if your code does many scattered memory references like in Graph Analytics and not many FLOPs ?
  - Chose the metric which suits your application (like time to solution or updates/sec instead of FLOPs/sec) and keep that metric throughout the optimization process.
- **Modeling helps to understand the upper bound for performance.**
  - Models are simplified mathematical descriptions of a kernel using specific values like amount of bytes used and FLOPs performed as well as system values like memory bandwidth and CPU performance to estimate the overall performance. (not covered in this tutorial.)

General Remarks on Optimization

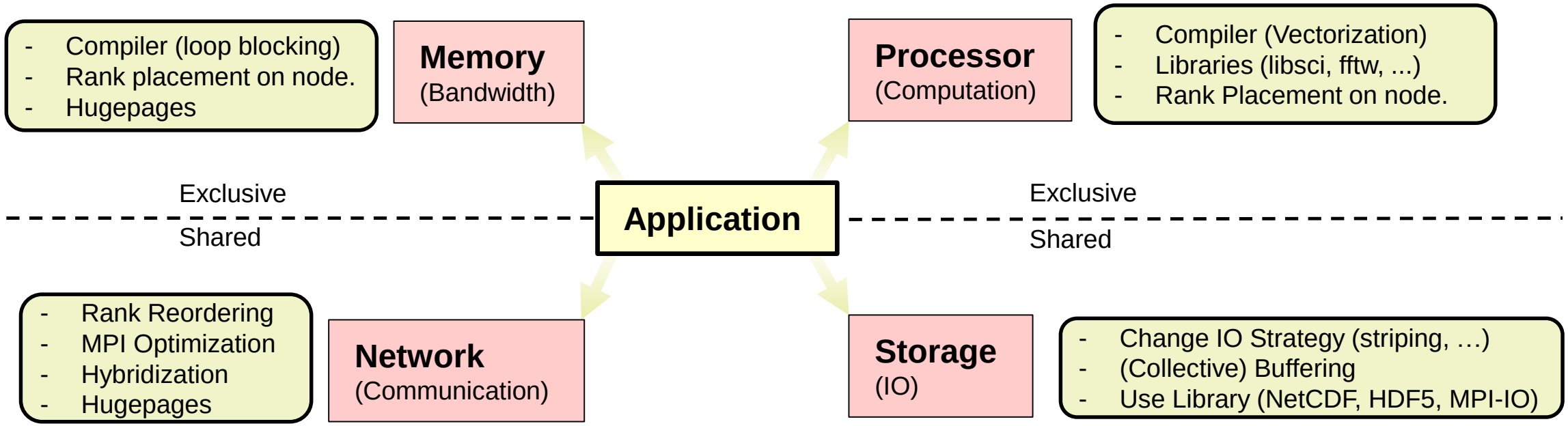
- **Start with low hanging fruits, i.e. avoid code modifications first of all**
  - Compiler optimizations
  - Manual Rank Reordering
  - Use optimized libraries
  - Huge Pages
  - Try Hyperthreads
  - MPI variables and DMAP
  - Experiment with different application placements on a node.

- **Use performance tools**
  - Guided rank reordering.
  - Automatic parallelization (OpenMP)
  - Identify code regions which might benefit from a rewrite.

General Remarks on Optimization

- **Use examples with different sizes for your experiments.**
  - small, medium, large, where node count differs by an order of magnitude. Can be strong or weak scaling.
    - Use small case to experiment with application placements and compiler flags for instance. Load balancing issues might already appear.
    - Keep an eye on communication time and load balancing for the large cases especially for strong scaling.
  - A possibility to create this set of examples could be to use the same problem with different resolutions.
    - Change work per node (strong scaling) or fix work per node (weak scaling). Resolution could be the number of grid points in continuum mechanics or the number of particles in molecular dynamics for instance.
- **A good understanding of the workflow of your application (Communication, Computation, IO, …) helps to better interpret the profiles.**

Bottlenecks and Remedies



- Good: One bottleneck which can be easily improved without creating a new one.
- Bad: Several bottlenecks interacting with each other and changing over time.
- Need a profiler to identify bottleneck(s) and a model to estimate optimization potential.

IOBUF

- **IOBUF** is an **I/O buffering library** that can reduce the I/O wait time for programs that read or write large files sequentially. IOBUF **intercepts I/O system calls** such as read and open and adds a layer of buffering, thus improving program performance by enabling asynchronous prefetching and caching of file data.

- **IOBUF** can also gather runtime statistics and print a summary report of I/O activity for each file. (***verbose*** option)

How to use IOBUF

Four Steps to use IOBUF:

1. module load iobuf
2. Link application
3. export IOBUF_PARAMS='*'
4. Run application

- For a detailed output use:
  - export IOBUF_PARAMS='*:verbose'
- Other options:
  - count: changes number of buffers
  - size: changes size of buffers
- Example:
  - export IOBUF_PARAMS='*:verbose:size=10M:count=10'
- Many other options available.
  - For details see man iobuf

IOBUF

- **Application produced massive serial IO on Lustre**
- **A generic solution for serial IO is buffering.**
  - Default Linux buffering offers no control.
- **Other solutions:**
  - Moving part of the IO to /tmp , which resides in the memory or is local
    - i.e. changing the source code
    - With cce options for assign available
  - Changing the IO pattern
  - Rewriting the algorithm
- **No source code available, only object files**
  Possible solutions by :
  - Buffering of the Intel Compiler
  - IOBUF

CRAY tools

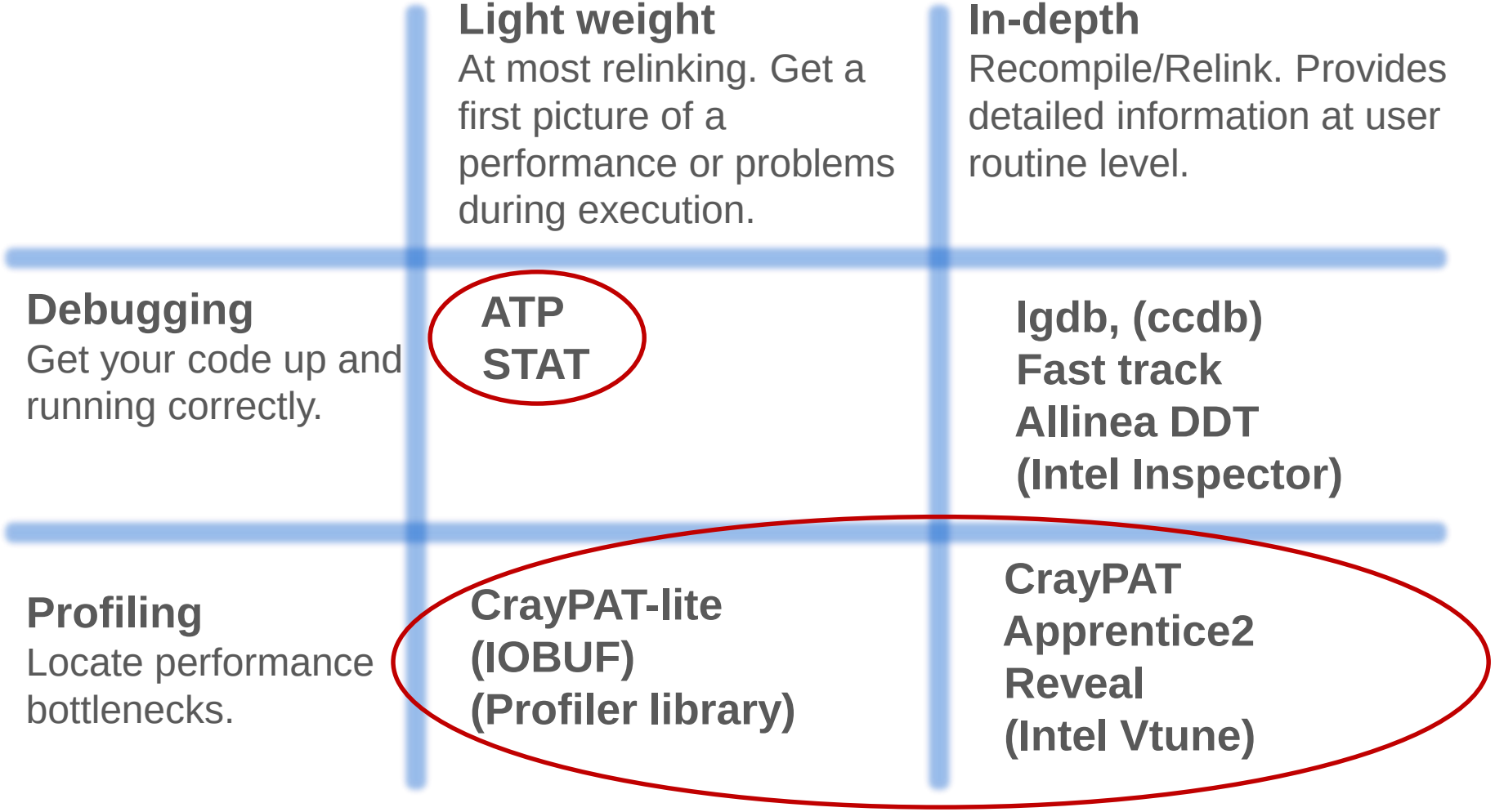
Debugging and profiling at scale

Overview

- **Debugging tools**
  - Stack Trace Analysis Tool (STAT)
  - Abnormal Termination Processing (ATP)

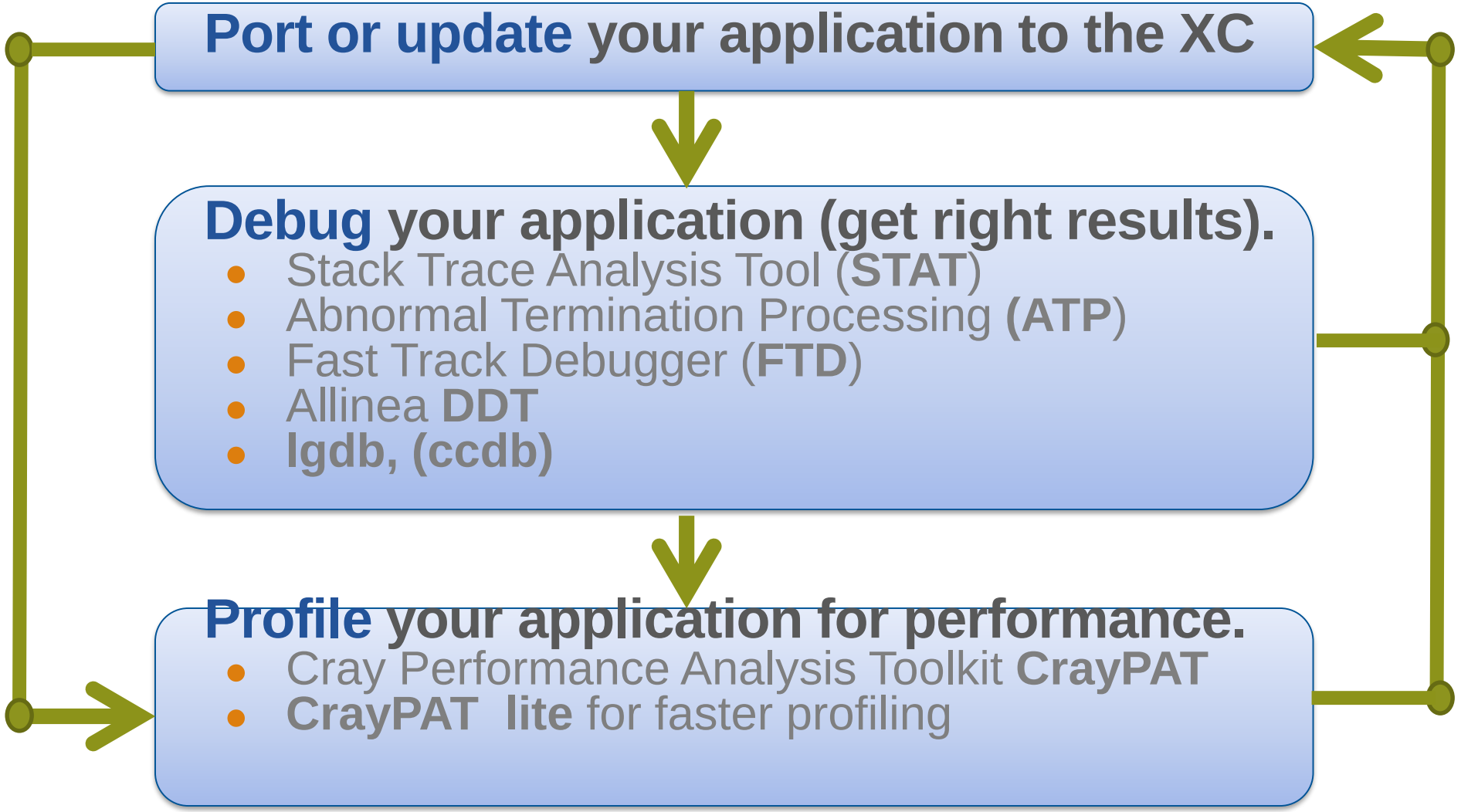
- **Profiling**
  - Perftools
  - CrayPAT-lite
  - CrayPAT
  - Apprentice^2
  - Reveal

Overview



- **More information about Cray Tools on pubs.cray.com**

The porting optimization Cycle

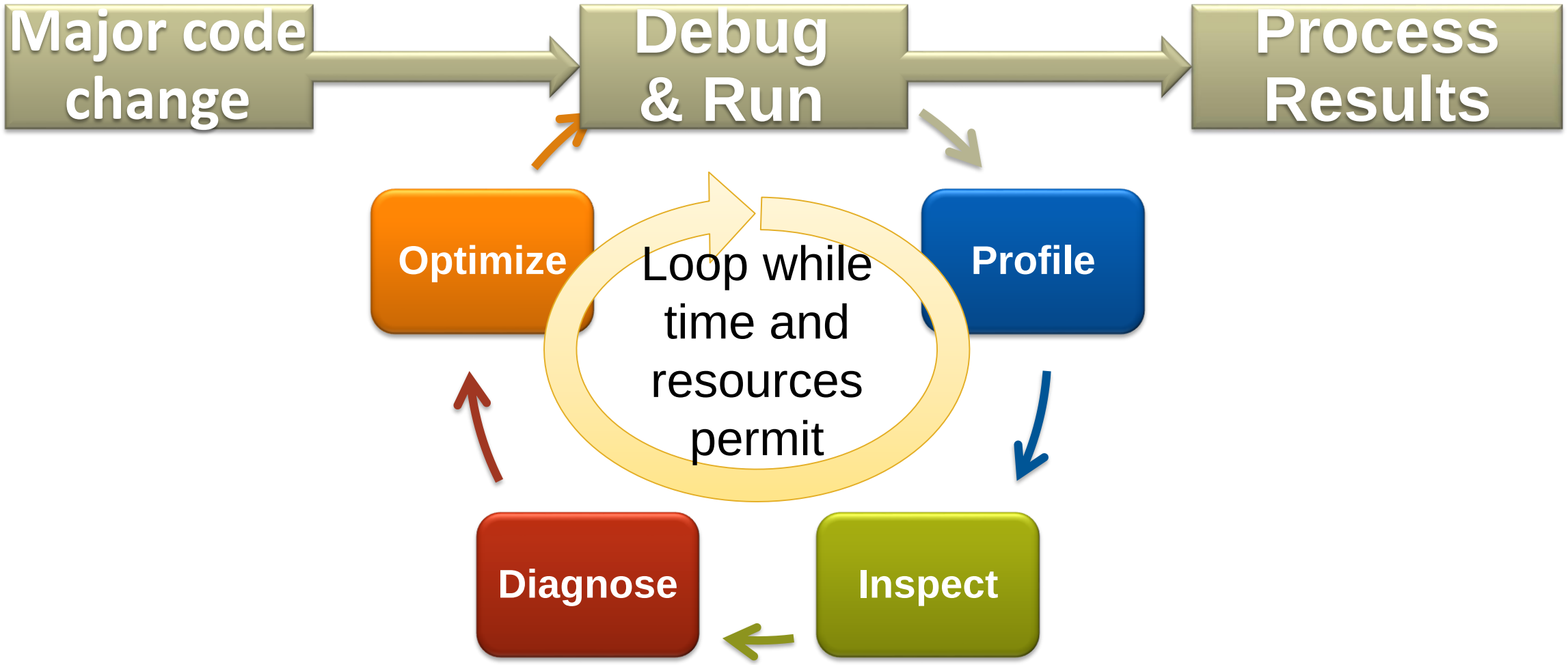


Debugging in production and scale

- **Even with the most rigorous testing, bugs may occur during development or production runs.**
  - It can be very difficult to recreate a crash without additional information
  - Even worse, for production codes need to be efficient so usually have debugging disabled
- **The failing application may have been using tens of or hundreds of thousands of processes**
  - If a crash occurs one, many, or all of the processes might issue a signal.
  - We don't want the core files from every crashed process, they're slow and too big!
  - We don't want a backtrace from every process, they're difficult to comprehend and analyze.

Performance Analysis with CrayPat

The Optimization Cycle



New Program Instrumentation Modules

- loaded low-impact module perftools-base
- Instrumentation modules available after perftools-base is loaded:
  - perftools-lite (sampling experiments)
  - perftools-lite-events (tracing experimants)
  - perftools-lite-loops (collect data for auto-parallelization / loop estimates in Reveal)
  - perftools-lite-gpu (gpu kernel and data movemnets)
  - perftools (fully adjustable CrayPAT, using pat_build and pat_report)

What Do the Instrumentation Modules Do?

perftools-lite

- Default CrayPat-lite profiling
- Load before building and running program to get a basic performance profile sent to stdout
- Equivalent to loading perftools-lite module in earlier releases

perftools-lite-events

- CrayPat-lite event profile
- Load before building and running program to get more in-depth performance data sent to stdout
- Equivalent to loading perftools-lite module and setting CRAYPAT_LITE environment variable to event_profile in earlier release

Perftools Instrumentation Modules

perftools-lite-loops

- CrayPat-lite loop work estimates
- Must be used with Cray compiler
- Load before building and running program to get loop work estimates sent to stdout and to .ap2 file for use with Reveal
- Automates loop work experiment by modifying the compile and link steps to include CCE's –h profile_generate option and instrumenting the program for tracing (pat_build -w).
- Remember that –h profile_generate reduces compiler optimization levels. After experiment is complete, unload perftools-lite-loops to prevent further program instrumentation.

Perftools Instrumentation Modules

perftools-lite-gpu

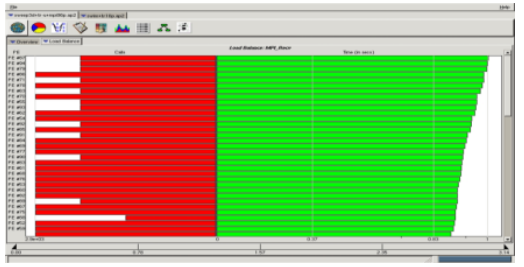
- CrayPat-lite GPU kernel and data movement information
- Load before building and running program to get GPU-specific performance data sent to stdout
- Equivalent to loading the perftools-lite module and setting CRAYPAT_LITE environment variable to gpu in earlier releases

perftools

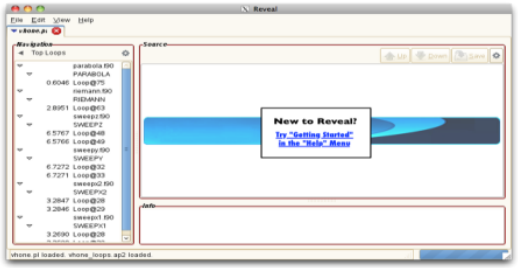
- Full access to CrayPAT functionality
- Use pat_build to instrument, pat_report to process data and collect reports
- (more details follow)

Components of Perftools

- **CrayPAT-lite-XXX** – automatic instrumentation and profiling
- **CrayPAT** – instrumentation and performance analysis tool, including pat_build and pat_report (details follow)



- **Cray Apprentice2** - A graphical analysis tool that can be used to visualize and explore the performance data captured during program execution.
- **Reveal** - A graphical source code analysis tool that can be used to correlate performance analysis data with annotated source code listings, to identify key opportunities for optimization.



Components of Perftools (cont.)

- **grid_order** - Generates MPI rank order information that can be used with the MPICH_RANK_REORDER environment variable to override the default MPI rank placement scheme and specify a custom rank placement. (For more information, see the **intro_mpi(3)** man page.)
- **pat_help** - Help system, which contains extensive usage information and examples. This help system can be accessed by entering pat_help at the command line.
- **Documentation** - The individual components of CrayPat are documented in the following man pages (info on hardware counters will follow):
  - **intro_craypat(1), pat_build(1), pat_report(1), pat_help(1), grid_order(1), app2(1), reveal(1)**

CrayPAT - lite

Examples of sampling, tracing and loop profiling

Generate a Sampling Profile

```
$> module load perftools-base
$> module load perftools-lite
```

- Provide basic tools and environment settings
- Set environment for sampling experiments

```
$> make clean; make
```

- Builds already instrumented binary e.g. app.exe

```
$> aprun –n 24 app.exe >& job.out
$> less job.out
```

- Running the instrumented binary creates a *.rpt and a *.ap2 file
- The report is additionally printed to stdout

Sampling Report

```
$> make
...
INFO: A maximum of 51 functions from group 'io' will be traced.
INFO: A maximum of 208 functions from group 'mpi' will be traced.
INFO: A maximum of 20 functions from group 'realtime' will be traced.
INFO: A maximum of 56 functions from group 'syscall' will be traced.
INFO: creating the CrayPat-instrumented executable
'/a/certain/dir/cp2k.pdbg' (sample_profile) ...OK

> cat job.out
...
#####################################################################
#                                                                   #
#            CrayPat-lite Performance Statistics                    #
#                                                                   #
#####################################################################

CrayPat/X:  Version 6.3.0 Revision 14378 (xf 14041)  09/15/15 10:48:06
Experiment:                  lite  lite/sample_profile
Number of PEs (MPI ranks):     48
Numbers of PEs per Node:       24  PEs on each of  2  Nodes
Numbers of Threads per PE:      1
Number of Cores per Socket:    12
Execution start time:  Wed Oct 14 14:07:17 2015
System name and speed:  mom11 2501 MHz

Avg Process Time:          5.14 secs
High Memory:               2,070 MBytes  43.13 MBytes per PE
MFLOPS:           Not supported (see observation below)
I/O Read Rate:          4.803892 MBytes/sec
I/O Write Rate:        88.963763 MBytes/sec
Avg CPU Energy:            1,499 joules 749.50 joules per node
Avg CPU Power:           291.59 watts  145.80 watts per node
```

General job information

Portions of samples

Significant portion of communication

```
...
Table 1:  Profile by Function Group and Function (top 8 functions shown)

  Samp% |  Samp |  Imb. |  Imb. |Group
        |       |  Samp | Samp% | Function
        |       |       |       |  PE=HIDE

 100.0% | 263.4 |    -- |    -- |Total
|-----------------------------------------------------------------------
|  78.0% | 205.3 |    -- |    -- |MPI
||----------------------------------------------------------------------
||  62.4% | 164.4 | 115.6 | 42.2% |mpi_bcast
||  10.4% |  27.4 | 186.6 | 89.1% |MPI_ALLREDUCE
||   4.7% |  12.4 |  86.6 | 89.3% |MPI_IPROBE
||======================================================================
|  13.1% |  34.5 |    -- |    -- |USER
||----------------------------------------------------------------------
||   3.3% |   8.6 |  61.4 | 89.5% |__message_passing_MOD_mp_probe
||   2.8% |   7.5 |   8.5 | 54.4% |__fist_nonbond_force_MOD_force_nonbond
||   2.0% |   5.2 |   5.8 | 53.6% |__ewalds_MOD_ewald_evaluate
||   1.1% |   2.9 |   3.1 | 52.5% |__splines_methods_MOD_potential_s
||======================================================================
|   8.2% |  21.5 |    -- |    -- |ETC
||----------------------------------------------------------------------
||   2.5% |   6.6 |   9.4 | 59.7% |__memmove_ssse3
||   1.7% |   4.4 |   4.6 | 52.7% |__memset_sse2
|======================================================================...
...
```

Sampling report

```
...
==================  Observations and suggestions  ==================
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 Samp

  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_Samp was generated
  along with this report and contains usage instructions and the
  Custom rank order from the following table.

    Rank    Node Reduction    Maximum  Average
    Order  Metric    in Max      Value  Value
             Imb.     Value

   Current  11.17%            9.310e+02  8.270e+02
   Custom   2.59%    8.808%  8.490e+02  8.270e+02
=======================  End Observations  ========================
...
```

Rank reorder suggestions

```
...
Table 2:  File Input Stats by Filename
     Read |     Read |  Read Rate |   Reads |   Bytes/ |File Name[max15]
     Time |   MBytes | MBytes/sec |         |    Call | PE=HIDE
 0.113291 | 0.544238 |   4.803892 | 2,964.0 |   192.54 |Total
|-------------------------------------------------------------------------
| 0.057170 | 0.214447 |   3.751054 | 1,586.0 |   141.78 |topology_fist_WAT.psf
| 0.026845 | 0.138477 |   5.158328 |   844.0 |   172.04 |H2O_ice.inp
| 0.014117 | 0.000700 |   0.049586 |     3.0 |   244.67 |TMC_NPT.inp
| 0.007784 | 0.098442 |  12.646622 |   176.0 |   586.50 |/proc/meminfo
| 0.006957 | 0.078669 |  11.307646 |    25.0 | 3,299.60 |./ice_Ih_96.xyz
|=========================================================================

Table 3:  File Output Stats by Filename
    Write |     Write | Write Rate |  Writes |
     Time |    MBytes | MBytes/sec |         |     Call | PE=HIDE
 0.162883 | 14.490714 |  88.963763 | 5,203.0 | 2,920.36 |Total
|-------------------------------------------------------------------------
| 0.096137 | 13.861026 | 144.179480 | 3,805.0 | 3,819.80 |tmc_traj_T270.xyz
| 0.021800 |  0.064217 |   2.945740 |    18.0 | 3,740.89 |tmc_E_worker_1.out
| 0.016016 |  0.064296 |   4.014441 |    18.0 | 3,745.50 |tmc_E_worker_6.out
| 0.013735 |  0.155310 |  11.307340 |   761.0 |   214.00 |tmc_traj_T270.cell
| 0.004775 |  0.063504 |  13.300140 |    18.0 | 3,699.39 |tmc_E_worker_7.out
| 0.003025 |  0.026007 |   8.596676 |   505.0 |    54.00 |stdout
| 0.001983 |  0.064375 |  32.470347 |    19.0 | 3,552.74 |tmc_E_worker_3.out
| 0.001915 |  0.064375 |  33.624425 |    19.0 | 3,552.74 |tmc_E_worker_2.out
| 0.001905 |  0.063979 |  33.588895 |    18.0 | 3,727.06 |tmc_E_worker_4.out
| 0.001582 |  0.063504 |  40.142573 |    18.0 | 3,699.39 |tmc_E_worker_5.out
| 0.000011 |  0.000122 |  11.053907 |     4.0 |    32.00 |_UnknownFile_
|=========================================================================
```

Input/Output analysis

Generate a Tracing Profile

```
$> module load perftools-base
$> module load perftools-lite-events
```

- Provide basic tools and environment settings
- Set environment for tracing experiments

```
$> make clean; make
```

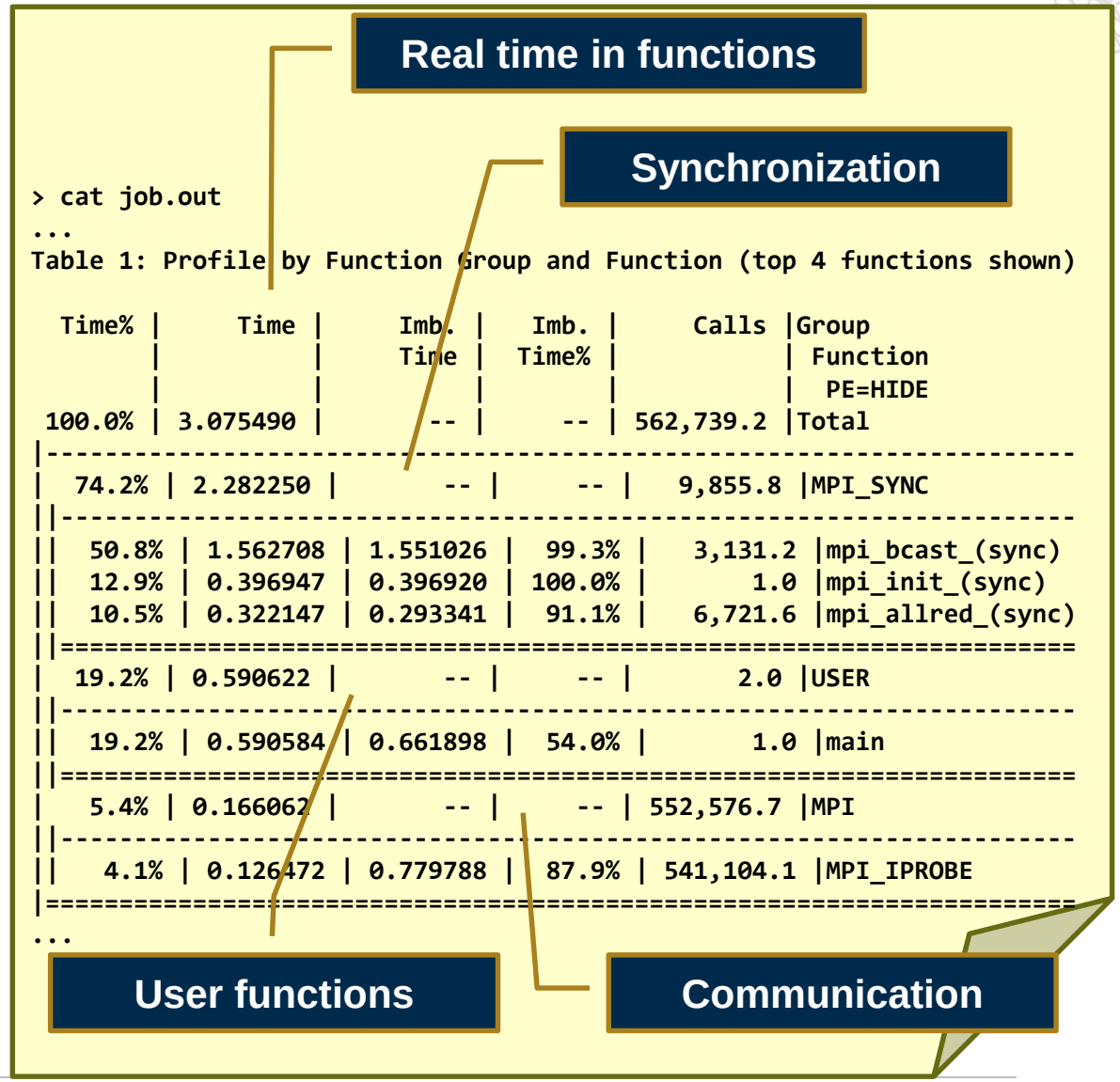
- Builds already instrumented binary e.g. app.exe

```
$> aprun –n 24 app.exe >& job.out
$> less job.out
```

- Running the instrumented binary creates a *.rpt and a *.ap2 file
- The report is additionally printed to stdout

Tracing report

- Comparable to sampling experiment, but now the function are really traced from beginning to end
- Again observations and suggestions are printed
  - E.g. rank reordering
  - And IO observations



Generate a loop Profile

```
$> module load perftools-base
$> module load perftools-lite-loops
```

- Provide basic tools and environment settings
- Set environment for tracing experiments with loop profiling

```
$> make clean; make
```

- Builds already instrumented binary e.g. app.exe

```
$> aprun –n 24 app.exe >& job.out
$> less job.out
```

- Running the instrumented binary creates a *.rpt and a *.ap2 file
- The report is additionally printed to stdout

Loop timing report

Subroutine

Line number

```
Table 1:  Inclusive and Exclusive Time in Loops (from -hprofile_generate)
  Loop | Loop Incl |      Time |    Loop |  Loop |  Loop |  Loop |Function=/.LOOP[.]
  Incl |      Time |     (Loop |     Hit | Trips | Trips | Trips | PE=HIDE
 Time% |           |     Adj.) |         |   Avg |   Min |   Max |
|-----------------------------------------------------------------------------
| 93.0% | 19.232051 |  0.000849 |       2 |  26.5 |     3 |    50 |jacobi.LOOP.1.li.236
| 77.8% | 16.092021 |  0.001350 |      53 | 255.0 |   255 |   255 |jacobi.LOOP.2.li.240
| 77.8% | 16.090671 |  0.110827 |   13515 | 255.0 |   255 |   255 |jacobi.LOOP.3.li.241
| 77.3% | 15.979844 | 15.979844 | 3446325 | 511.0 |   511 |   511 |jacobi.LOOP.4.li.242
| 14.1% |  2.906115 |  0.001238 |      53 | 255.0 |   255 |   255 |jacobi.LOOP.5.li.263
| 14.0% |  2.904878 |  0.688611 |   13515 | 255.0 |   255 |   255 |jacobi.LOOP.6.li.264
| 10.7% |  2.216267 |  2.216267 | 3446325 | 511.0 |   511 |   511 |jacobi.LOOP.7.li.265
|  4.3% |  0.881573 |  0.000010 |       1 | 259.0 |   259 |   259 |initmt.LOOP.1.li.191
|  4.3% |  0.881563 |  0.000645 |     259 | 259.0 |   259 |   259 |initmt.LOOP.2.li.192
|  4.3% |  0.880918 |  0.880918 |   67081 | 515.0 |   515 |   515 |initmt.LOOP.3.li.193
|  2.7% |  0.560055 |  0.000055 |       1 | 257.0 |   257 |   257 |initmt.LOOP.4.li.210
|  2.7% |  0.560444 |  0.006603 |     257 | 257.0 |   257 |   257 |initmt.LOOP.5.li.211
|  2.7% |  0.553842 |  0.553842 |   66049 | 513.0 |   513 |   513 |initmt.LOOP.6.li.212
```

Nested Loops

CrayPAT

fully adjustable profiling

Profiling with CrayPAT

```
$> module load perftools-base
$> module load perftools
```

- Makes the default version of CrayPAT available.

```
$> make clean; make
```

- If your application is already built with `perftools` loaded you do not have to rebuild when switching the experiment.

```
$> pat_build <pat_options> app.exe
```

- pat_options are described below
- Creates instrumented binary app.exe+pat

```
$> aprun -n 24 ./app.exe+pat
```

```
$> pat_report -o myrep.txt himeno+pat+*
```

- Running the “+pat” binary creates a data file or directory
- pat_report reads that data file and prints lots of human-readable performance data. Creates an *.ap2 file.

Some pat_build options

Option	Description
	Sampling profile
-u	tracing of functions in source file owned by the user
-w	Tracing is default experiment
-T <func>	Specifies a function which will be traced
-t <file>	All functions in the specified file will be traces
-g <group>	Instrument all functions belonging to the specified trace function group, e.g. blas, io, mpi, netcdf, syscall

- **More information:**
  - is given in `man pat_build` page
  - functions in tracegroups are given in `$CRAYPAT_ROOT/share/traces` after loading the `perftools` module
- **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.**

Using pat_report

- **pat_report perform data conversion**
  - Combines information from *.xf output (raw data files, optimized for writing to disk)
  - Instrumented binary must still exist when data is converted!
  - produce *.ap2 file (compressed performance file, optimized for visualization analysis)
    - ap2 file is the input for subsequent pat_report calls and Reveal or Apprentice[2]
  - *.xf files and instrumented binary files can be removed once ap2 file is generated.
- **Generates a text report of performance results**
  - Many options for sorting, slicing or dicing data in the tables.

```
$> pat_report -O <table option> *.ap2
$> pat_report -O help (list of available profiles)
```

  - Volume and type of information depends upon sampling vs tracing.
  - Several output formats {plot | rpt | ap2 | ap2-xml | ap2-txt | xf-xml | xf-txt | html} available through -f option.
- **filter the gathered data**

```
$> pat_report -sfilter_input='condition' …
```

  - The 'condition' could be an expression involving 'pe' such as 'pe<1024' or 'pe%2==0'.

Combining Sampling and Tracing: APA

- **Automatic Profiling Analysis:**
  - **Target:** large, long-running program (general a trace will inject considerable overhead)
  - **Goal:** limit tracing to those functions that consume the most time.
  - **Procedure:** use a preliminary sampling experiment to determine and instrument functions consuming the most time

```
$> module load perftools
```

```
$> make clean; make
$> pat_build myapp.exe
```

• The APA is the default experiment. No option needed.

```
$> aprun –n 24 ./myapp.exe+pat
$> pat_report –o myrep.txt myapp+pat+*
```

• Applying pat_report to the *.xf generates an ***.apa** file in addition to the *.ap2 file.

```
$> vi *.apa
```

• The *.apa file contains instructions for the next instrumentation step.
• Modify it according to your needs.

```
$> pat_build –O *.apa
```

• Generates an instrumented binary `*.exe+apa` for tracing

```
$> aprun –n 24 ./myapp.exe+apa
$> pat_report –o myrep.txt myapp+apa+*
```

• Applying `pat_report` to the *.xf generates a new*.ap2 file.

Loop Work Estimates

- Gives information on inclusive time spent in the loop nests and typical trip count of the loops
- Please use only one thread (OMP_NUM_THREADS=1)
- Only available with CCE.

```
$> module load perftools-base
$> module load perftools
```

```
$> ftn -c -h profile_generate himeno.f90
$> ftn -o myapp.exe myapp.o
```

- Recompile your program for gathering loop statistics
- It is recommended to turn off OpenMP and OpenACC for the loop work estimates via `-h noomp -h noacc`

```
$> pat_build -w[-u] myapp.exe
```

- Instrument the application for tracing (APA also possible)

Apprentice^2

Graphical representation of performance data

Cray Apprentice^2

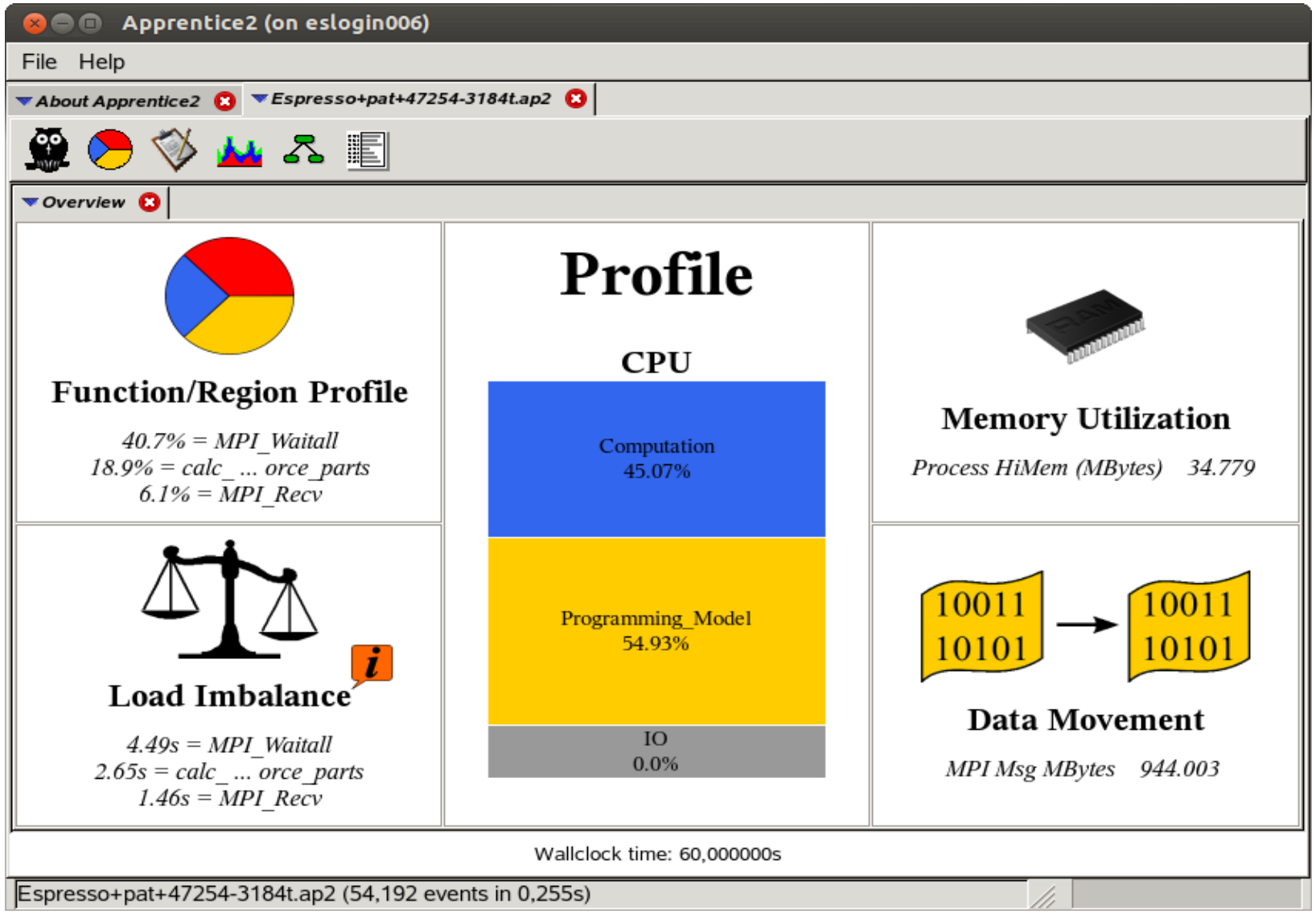
- **Cray Apprentice2 is a post-processing performance data visualization tool. Takes *.ap2 files as input.**

- **Main features are**
  - Call graph profile
  - Communication statistics
  - Time-line view for Communication and IO.
  - Activity view
  - Pair-wise communication statistics
  - Text reports

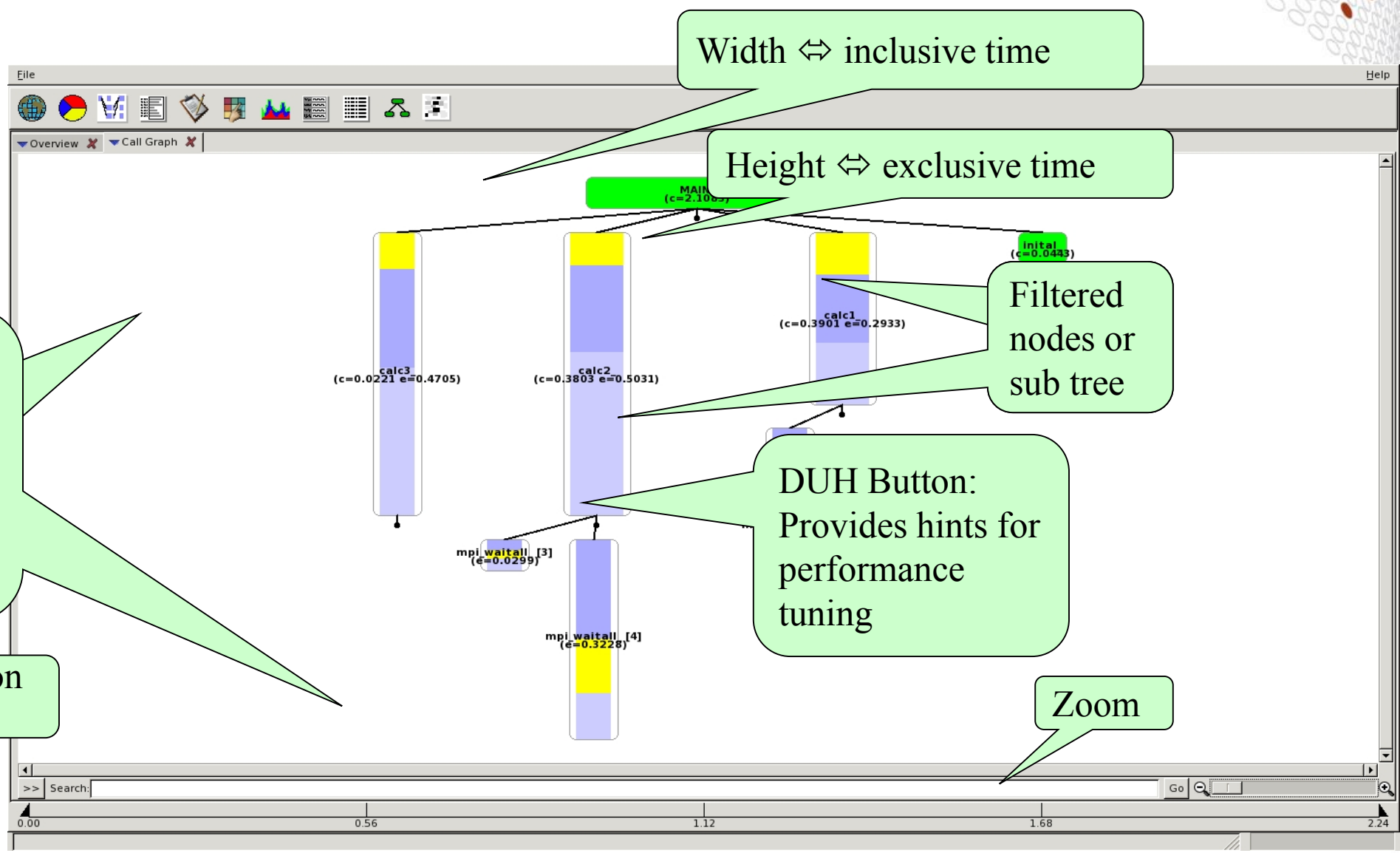
- **helps identify:**
  - Load imbalance
  - Excessive communication
  - Network contention
  - Excessive serialization
  - I/O Problems

```
$> module load perftools-base
$> app2 my_program.ap2 &
```

Cray Apprentice²



Call Tree View



Installing Apprentice2 on Laptop

From a Cray login node

$ module load perftools

Go to:

- $CRAYPAT_ROOT/share/desktop_installers/
- Download .dmg or .exe installer to laptop
- Double click on installer and follow directions to install
- Of course, can just run app2 from the login prompt instead

Reveal

Compiler Feedback and Variable scoping

Reveal

- **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

```
$> module load perftools
$> 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.

For more details : https://pubs.cray.com

Questions?

Thank you

For any issues, contact ***pratyushsupport@tropmet.res.in***