Hybrid Multi-core Programming for Exascale Computing

John Levesque
Cray CTO Office
Director of Cray’s Supercomputing Center of Excellence

XK6 Workshop April 2012
April 9-11
Key Challenges to Get to an Exascale

**Power**
- Traditional voltage scaling is over
- Power now a major design constraint
- Cost of ownership
- Driving significant changes in architecture

**Concurrency**
- A billion operations per clock
- Billions of refs in flight at all times
- Will require huge problems
- Need to exploit all available parallelism

**Programming Difficulty**
- Concurrency and new micro-architectures will significantly complicate software
- Need to hide this complexity from the users

**Resiliency**
- Many more components
- Components getting less reliable
- Checkpoint bandwidth not scaling

Cray Inc. SNL Workshop Apr 9-11
Improving Processor Efficiency

- Multi-core was a good first response to power issues
  - Performance through parallelism
  - Modest clock rate
  - Exploit on-chip locality

- However, conventional processor architectures are optimized for single thread performance rather than energy efficiency
  - Fast clock rate with latency(performance)-optimized memory structures
  - Wide superscalar instruction issue with dynamic conflict detection
  - Heavy use of speculative execution and replay traps
  - Large structures supporting various types of predictions
  - Relatively little energy spent on actual ALU operations

- Could be much more energy efficient with multiple simple processors, exploiting vector/SIMD parallelism and a slower clock rate

- But serial thread performance is really important (Amdahl’s Law):
  - If you get great parallel speedup, but hurt serial performance, then you end up with a niche processor (less generally applicable, harder to program)
Conclusion: Heterogeneous Computing

- To achieve scale and sustained performance per \{\$/watt\}, must adopt:
  - ...a heterogeneous node architecture
    - fast cores for serial code
    - many power-efficient cores for parallel code
  - ...a deep, explicitly managed memory hierarchy
    - to better exploit locality, improve predictability, and reduce overhead
  - ...a microarchitecture to exploit parallelism at all levels of a code
    - distributed memory, shared memory, vector/SIMD, multithreaded
    - (related to the “concurrency” challenge—leave no parallelism untapped)

- Sounds a lot like GPU accelerators...
- NVIDIA Fermi™ has made GPUs feasible for HPC
  - Robust error protection and strong DP FP, plus programming enhancements
- Expect GPUs to make continued and significant inroads into HPC
  - Compelling technical reasons
  - High volume market
  - It looks like they can credibly support both masters (graphics and compute)

- Two issues w/ GPU acceleration: STRUCTURAL and PROGRAMMING
This is a short-lived situation
- NVIDIA Denver and AMD Fusion

Try to keep kernel data structures resident in GPU memory
- Avoids copying b/w CPU and GPU; work on GPU-network communication

May limit breadth of applicability over next 2-3 years
Structural Issues with Accelerated Computing Even with fused products

- GPU-CPU
  - Lower GF
- GPU
  - Higher GF

Memory Capacity
- > 32GB SDRAM
- < 16 GB GDDR

Main() vs. Flops
- Slower GB/s
- Faster GB/s

Memory Bandwidth
Programming Issues with Accelerated Computing

- Primary issues with programming for GPUs:
  - Learn new language/programming model
  - Maintain two code bases/lack of portability
  - Tuning for complex processor architecture (and split CPU/GPU structure)

- Need a single programming model that is **portable across machine types**, and also **forward scalable** in time
  - Portable expression of heterogeneity and multi-level parallelism
  - Programming model and optimization should not be significantly difference for “accelerated” nodes and multi-core x86 processors
  - *Allow users to maintain a single code base*

- Need to shield user from the complexity of dealing with heterogeneity
  - High level language with good complier and runtime support
  - Optimized libraries for heterogeneous multicore processors

- Directive-based approach makes sense (OpenACC)

- Getting the division of labor right:
  - User should focus on identifying parallelism (we can help with good tools)
  - Compiler and runtime can deal with mapping it onto the hardware
Intel ‘s Knight's Ferry

From Michael Wolfe's HPC Article
Nvidia Fermi

From Michael Wolfe’s HPC Article
<table>
<thead>
<tr>
<th><strong>Intel MIC</strong></th>
<th><strong>NVIDIA Fermi</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>MIMD Parallelism</td>
<td>32</td>
</tr>
<tr>
<td>SIMD Parallelism</td>
<td>16</td>
</tr>
<tr>
<td>Instruction-Level Parallelism</td>
<td>2</td>
</tr>
<tr>
<td>Thread Granularity</td>
<td>coarse</td>
</tr>
<tr>
<td>Multithreading</td>
<td>4</td>
</tr>
<tr>
<td>Clock</td>
<td>1.2GHz</td>
</tr>
<tr>
<td>L1 cache/processor</td>
<td>32KB</td>
</tr>
<tr>
<td>L2 cache/processor</td>
<td>256KB</td>
</tr>
<tr>
<td>programming model</td>
<td>posix threads/ Directives</td>
</tr>
<tr>
<td>virtual memory</td>
<td>yes</td>
</tr>
<tr>
<td>memory shared with host</td>
<td>no</td>
</tr>
<tr>
<td>hardware parallelism support</td>
<td>no</td>
</tr>
<tr>
<td>mature tools</td>
<td>yes</td>
</tr>
</tbody>
</table>

From Michael Wolfe’s HPC Article
Hybrid Multi-core Architecture

• Massively Parallel System with high powered nodes that exhibit
  • Multiple levels of parallelism
    • Shared Memory parallelism on the node
    • SIMD vector units on each core or thread
  • Potentially disparate processing units
    • Host with conventional X86 architecture
    • Accelerator with highly parallel – SIMD units
  • Potentially disparate memories
    • Host with conventional DDR memory
    • Accelerator with high bandwidth memory
Hybrid Multi-core Architecture

- All MPI may not be best approach
  - Memory per core will decrease
  - Injection bandwidth/core will decrease
  - Memory bandwidth/core will decrease
- Hybrid MPI + threading on node may be able to
  - Save Memory
  - Reduce amount of off node communication required
  - Reduce amount of memory bandwidth required
Cray XK6 -- Hybrid Innovation

Blending the best-of-the-best into a true hybrid supercomputer

- AMD Series 6200 16-core Interlagos processors
- NVIDIA® Tesla™ 20-series many-core processors
- Cray Gemini High-Performance Interconnect
- CLE and CPE, Cray’s scalable software environment

All this in order to create a production, scalable, adaptive supercomputer — putting our customers on the road to productive exascale computing
Accelerated Performance through Integration

AMD Series 6200 CPU

NVIDIA Tesla GPU with 665GF DPFP

6GB GDDR5; 138 GB/s

Cray Gemini High Speed Interconnect

1600 MHz DDR3; 16, 32 or 64 GB

Cray Inc. SNL Workshop Apr 9-11
And then there is Intel’s MIC processor

- Current MICs have 32 Intel processors moving to 50 processors, both of these systems have vector length of 512 bits (8 – 64 bit words of 16-32 bit words)
- While Intel is saying that codes can be compiled directly for the MIC (Including MPI), one has to be concerned about
  - The scalar performance of one of those cores
  - The amount of memory on the MIC
- If there is too much scalar code and/or too much memory required, then the MIC will necessarily be treated like the other accelerators
  - Two disparate memories
  - Two disparate computational engines
Greater than 3TF per blade
NVIDIA SXM is Feature Rich for HPC

High density form factor at less than 225 W

665 GF double precision floating point with ECC protection

6 GB of GDDR5 memory available at 138 GB/s

Field upgradeable to Kepler in 2012 for over 1 TF of peak double precision performance
Easily Upgrade to Future Accelerators
<table>
<thead>
<tr>
<th>Description</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
<td>Jaguar</td>
</tr>
<tr>
<td>Architecture</td>
<td>XT5</td>
</tr>
<tr>
<td>Processor</td>
<td>6-Core AMD</td>
</tr>
<tr>
<td>Cabinets</td>
<td>200</td>
</tr>
<tr>
<td>Nodes</td>
<td>18,688</td>
</tr>
<tr>
<td>Cores/node</td>
<td>12</td>
</tr>
<tr>
<td>Total Cores</td>
<td>224,256</td>
</tr>
<tr>
<td>Memory/Node</td>
<td>16 GB</td>
</tr>
<tr>
<td>Memory/Core</td>
<td>1.3 GB</td>
</tr>
<tr>
<td>Interconnect</td>
<td>SeaStar2+</td>
</tr>
<tr>
<td>GPUs</td>
<td>0</td>
</tr>
</tbody>
</table>
### 2011 Configuration

<table>
<thead>
<tr>
<th>Specification</th>
<th>Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
<td>Jaguar</td>
</tr>
<tr>
<td>Architecture</td>
<td>XK6</td>
</tr>
<tr>
<td>Processor</td>
<td>16-Core AMD</td>
</tr>
<tr>
<td>Cabinets</td>
<td>200</td>
</tr>
<tr>
<td>Nodes</td>
<td>18,688</td>
</tr>
<tr>
<td>Cores/node</td>
<td>16</td>
</tr>
<tr>
<td>Total Cores</td>
<td>299,008</td>
</tr>
<tr>
<td>Memory/Node</td>
<td>32 GB</td>
</tr>
<tr>
<td>Memory/Core</td>
<td>2 GB</td>
</tr>
<tr>
<td>Interconnect</td>
<td>Gemini</td>
</tr>
<tr>
<td>GPUs</td>
<td>960</td>
</tr>
</tbody>
</table>
## Final Configuration

<table>
<thead>
<tr>
<th>Name</th>
<th>Titan</th>
</tr>
</thead>
<tbody>
<tr>
<td>Architecture</td>
<td>XK6</td>
</tr>
<tr>
<td>Processor</td>
<td>16-Core AMD</td>
</tr>
<tr>
<td>Cabinets</td>
<td>200</td>
</tr>
<tr>
<td>Nodes</td>
<td>18,688</td>
</tr>
<tr>
<td>Cores/node</td>
<td>16</td>
</tr>
<tr>
<td>Total Cores</td>
<td>299,008</td>
</tr>
<tr>
<td>Memory/Node</td>
<td>32 GB</td>
</tr>
<tr>
<td>Memory/Core</td>
<td>2 GB</td>
</tr>
<tr>
<td>Interconnect</td>
<td>Gemini</td>
</tr>
<tr>
<td>GPUs</td>
<td>TBD(.GE. 20PF)</td>
</tr>
</tbody>
</table>
Interlagos
In order to optimize the utilization of the shared and dedicated resources on the chip for different types of applications, modern x86 processors offer flexible options for running applications. As a result, the definition of a core has become ambiguous.

Definition of a Core for Blue Waters:
- Equivalent to an AMD “Interlagos” Compute Unit, which is an AMD Interlagos “Bulldozer module” consisting of: one instruction fetch/decode unit, one floating point scheduler with two FMAC execution units, two integer schedulers with multiple pipelines and L1 Dcache, and a L2 cache. This is sometimes also called a “Core Module.” A “core” = “compute unit” = “core module.”
Interlagos is composed of a number of “Bulldozer modules” or “Compute Unit”

- A compute unit has shared and dedicated components
  - There are two independent integer units; shared L2 cache, instruction fetch, Icache; and a shared, 256-bit Floating Point resource
  - A single Integer unit can make use of the entire Floating Point resource with 256-bit AVX instructions
    - Vector Length
      - 32 bit operands, VL = 8
      - 64 bit operands, VL = 4
Each processor die is composed of 4 compute units
- The 4 compute units share a memory controller and 8MB L3 data cache
  ▪ Each processor die is configured with two DDR3 memory channels and multiple HT3 links
Interlagos Die Floorplan
Interlagos Processor

- Two die are packaged on a multi-chip module to form an Interlagos processor
  - Processor socket is called G34 and is compatible with Magny Cours
  - Package contains
    - 8 compute units
    - 16 MB L3 Cache
    - 4 DDR3 1333 or 1600 memory channels
Interlagos Caches and Memory

• **L1 Cache**
  - 16 KB, 4-way predicted, parity protected
  - Write-through and inclusive with respect to L2
  - 4 cycle load to use latency

• **L2 Cache**
  - 2MB, Shared within core-module
  - 18-20 cycle load to use latency

• **L3 Cache**
  - 8 MB, non-inclusive victim cache (mostly exclusive)
    - Entries used by multiple core-modules will remain in cache
    - 1 to 2 MB used by probe filter (snoop bus)
    - 4 sub-caches, one close to each compute module
    - Minimum Load to latency of 55-60 cycles

• Minimum latency to memory is 90-100 cycles
Two MPI Tasks on a Compute Unit ("Dual-Stream Mode")

- An MPI task is pinned to each integer unit
  - Each integer unit has exclusive access to an integer scheduler, integer pipelines and L1 Dcache
  - The 256-bit FP unit, instruction fetch, and the L2 Cache are shared between the two integer units
    - 256-bit AVX instructions are dynamically executed as two 128-bit instructions if the 2nd FP unit is busy

- When to use
  - Code is highly scalable to a large number of MPI ranks
  - Code can run with a 2GB per task memory footprint
  - Code is not well vectorized
One MPI Task on a Compute Unit ("Single Stream Mode")

- Only one integer unit is used per compute unit
  - This unit has exclusive access to the 256-bit FP unit and is capable of 8 FP results per clock cycle
  - The unit has twice the memory capacity and memory bandwidth in this mode
  - The L2 cache is effectively twice as large
  - The peak of the chip is not reduced

- When to use
  - Code is highly vectorized and makes use of AVX instructions
  - Code benefits from higher per task memory size and bandwidth
One MPI Task per compute unit with Two OpenMP Threads ("Dual-Stream Mode")

- An MPI task is pinned to a compute unit
- OpenMP is used to run a thread on each integer unit
  - Each OpenMP thread has exclusive access to an integer scheduler, integer pipelines and L1 Dcache
  - The 256-bit FP unit and the L2 Cache is shared between the two threads
  - 256-bit AVX instructions are dynamically executed as two 128-bit instructions if the 2nd FP unit is busy
- When to use
  - Code needs a large amount of memory per MPI rank
  - Code has OpenMP parallelism at each MPI rank
AVX (Advanced Vector Extensions)

• Max Vector length doubled to 256 bit
• Much cleaner instruction set
  – Result register is unique from the source registers
  – Old SSE instruction set always destroyed a source register
• Floating point multiple-accumulate
  – A(1:4) = B(1:4) * C(1:4) + D(1:4) ! Now one instruction
• Next gen of both AMD and Intel will have AVX
• Vectors are becoming more important, not less
Running in Dual-Stream mode

- Dual-Stream mode is the current default mode on the Cray XE6 systems. General use does not require any options. CPU affinity is set automatically by ALPS.

- Use the `aprun -d` option to set the number of OpenMP threads per process. If OpenMP is not used, no `-d` option is required. The `aprun -N` option is used to specify the number of MPI processes to assign per compute node. This is generally needed if OpenMP threads are used and more than one node is used.
Running in Single-Stream mode

• Single-Stream mode is simple to specify on the Cray XE6 systems if no OpenMP threads are used. The aprun -d option is set to a value of 2, and CPU affinity is set automatically by ALPS. (Make sure $OMP_NUM_THREADS is not set, or is set to a value of 1.)

• When OpenMP threads are used, careful setting of the aprun -cc cpu_list option is required to get the desired CPU affinity. A cpu_list is a map of CPUs to threads. Each process is assigned a list of CPUs, with one CPU per thread. See the aprun(1) man page for more details. The aprun -N option is used to specify the number of MPI processes to assign per compute node. This is generally needed if more than one node is used in Single-Stream mode. Also, the environment variable $OMP_NUM_THREADS needs to be set to the correct number of threads per process.
**Examples**

- No OpenMP or 1 OpenMP thread per process, 16 processes per compute node
  
  -d 2

- 2 OpenMP threads per MPI process, 8 processes per compute node
  
  -N 8 -cc 0,2:4,6:8,10:12,14:16,18:20,22:24,26:28,30

- 4 OpenMP threads per MPI process, 4 processes per compute node
  
  -N 4 -cc 0,2,4,6:8,10,12,14:16,18,20,22:24,26,28,30

- 8 OpenMP threads per MPI process, 2 processes per compute node
  
  -N 2 -cc 0,2,4,6,8,10,12,14:16,18,20,22,24,26,28,30

- 16 OpenMP threads per MPI process, 1 process per compute node
  
  -N 1 -cc 0,2,4,6,8,10,12,14,16,18,20,22,24,26,28,30
NUMA Considerations

• An XE6 compute node with 2 Interlagos processors has 4 NUMA memory domains, each with 4 Bulldozer Modules. Access to memory located in a remote NUMA domain is slower than access to local memory. Bandwidth is lower, and latency is higher.

• OpenMP performance is usually better when all threads in a process execute in the same NUMA domain. For the Dual-Stream case, 8 CPUs share a NUMA domain, while in Single-Stream mode 4 CPUs share a NUMA domain. Using a larger number of OpenMP threads per MPI process than these values may result in lower performance due to cross-domain memory access.
<table>
<thead>
<tr>
<th>Run Type</th>
<th>Dual-Stream</th>
<th>Single-Stream</th>
</tr>
</thead>
<tbody>
<tr>
<td>No OpenMP</td>
<td>No option needed</td>
<td>-d 2 (note: <code>$OMP_NUM_THREADS</code> not set)</td>
</tr>
<tr>
<td>2 OpenMP threads</td>
<td>-N 16 -d 2</td>
<td>-N 8 -cc 0,2,4,6,8,10,12,14,16,18,20,22,24,26,28,30</td>
</tr>
<tr>
<td>4 OpenMP threads</td>
<td>-N 8 -d 4</td>
<td>-N 4 -cc 0,2,4,6,8,10,12,14,16,18,20,22,24,26,28,30</td>
</tr>
<tr>
<td>8 OpenMP threads</td>
<td>-N 4 -d 8</td>
<td>-N 2 -cc 0,2,4,6,8,10,12,14,16,18,20,22,24,26,28,30</td>
</tr>
<tr>
<td>16 OpenMP threads</td>
<td>-N 2 -d 16</td>
<td>-N 1 -cc 0,2,4,6,8,10,12,14,16,18,20,22,24,26,28,30</td>
</tr>
<tr>
<td>32 OpenMP threads</td>
<td>-N 1 -d 32</td>
<td>Not Applicable</td>
</tr>
</tbody>
</table>
XE6 Node Details: 24-core Magny Cours

- 2 Multi-Chip Modules, 4 Opteron Dies
- 8 Channels of DDR3 Bandwidth to 8 DIMMs
- 24 (or 16) Computational Cores
  - 64 KB L1 and 512 KB L2 caches for each core
  - 6 MB of shared L3 cache on each die
- Dies are fully connected with HT3
- Snoop Filter Feature Allows 4 Die SMP to scale well
XE6 Node Details:
24-core Magny Cours

- 2 Multi-Chip Modules, 4 Opteron Dies
- 8 Channels of DDR3 Bandwidth to 8 DIMMs
- 24 (or 16) Computational Cores
  - 64 KB L1 and 512 KB L2 caches for each core
  - 6 MB of shared L3 cache on each die
- Dies are fully connected with HT3
- Snoop Filter Feature Allows 4 Die SMP to scale well

Run using 1 MPI task on the node

Use OpenMP across all 24 cores

To Interconnect
HT1 / HT3
XE6 Node Details: 24-core Magny Cours

- 2 Multi-Chip Modules, 4 Opteron Dies
- 8 Channels of DDR3 Bandwidth to 8 DIMMs
- 24 (or 16) Computational Cores
  - 64 KB L1 and 512 KB L2 caches for each core
  - 6 MB of shared L3 cache on each die
- Dies are fully connected with HT3
- Snoop Filter Feature Allows 4 Die SMP to scale well

Run using 2 MPI tasks on the node One on Each Die

Use OpenMP across all 12 cores in the Die
- 2 Multi-Chip Modules, 4 Opteron Dies
- 8 Channels of DDR3 Bandwidth to 8 DIMMs
- 24 (or 16) Computational Cores
  - 64 KB L1 and 512 KB L2 caches for each core
  - 6 MB of shared L3 cache on each die
- Dies are fully connected with HT3
- Snoop Filter Feature Allows 4 Die SMP to scale well

Run using 4 MPI tasks on the node
One on Each Socket

Use OpenMP across all 6 cores in the Socket
Proposed Programming Paradigm for Hybrid Multi-core

- MPI or PGAS between nodes and/or sockets
- OpenMP, Pthreads or some other shared memory parallelism across a portion of the cores on the node
- Vectorization to utilize the SSE# or SIMD units on the cores
Cray Software Objectives for Accelerators

1. Provide baseline accelerator environment
   • Don’t preclude use of tools developers/programmers are used to

2. Integrated Programming Environment
   • Extension of PE Cray has provided on XT/XE systems
   • Provide “bundled” 3rd party commonly used or expected software (compilers, libraries, tools)

3. Cray integrated programming environment include:
   • Greatly enhance the productivity of the programming writing new applications or porting existing applications to accelerators
   • Improve performance of existing applications by exploiting greater levels of parallelism
   • Maintain source compatibility between multi-core and accelerator versions of the code
Programming for Future

Multi-Petaflop and Exaflop Computers

aka

Finding more parallelism in existing applications
Back to the Futures – Combining different levels of parallelism

- **Fact**
  - For the next decade all HPC system will basically have the same architecture
    - Message passing between nodes
    - Multi-threading within the node – MPI will not do
    - Vectorization at the lower level

- **Fact**
  - Current petascale applications are not structured to take advantage of these architectures
    - Current – 80-90% of applications use a single level of parallelism, message passing between the cores of the MPP system
    - Looking forward, application developers are faced with a significant task in preparing their applications for the future
Cray’s Programming Environment

- Tools for identifying additional parallel structures within the application
  - Investigation of looping structures within a complex application
  - Scoping tools for investigating the parallelizability of high level looping structures

- Tools for maintaining performance portable applications
  - Supply compiler that accepts directives from OpenMP sub-committee formulating extensions to target companion accelerators
    - Application developer able to develop a single code that can run efficiently on multi-core nodes with or without accelerator
Hybridization* of an All MPI Application

* Creation of an application that exhibits three levels of parallelism, MPI between nodes, OpenMP** on the node and vectorized looping structures

** Why OpenMP? To provide performance portability. OpenMP is the only threading construct that a compiler can analyze sufficiently to generate efficient threading on multi-core nodes and to generate efficient code for companion accelerators.
Do not read “Automatic” into this presentation, the Hybridization of an application is difficult and efficient code only comes with a thorough interaction with the cacciler to generate the most efficient code and

- High level OpenMP structures
- Low level vectorization of major computational areas

Performance is also dependent upon the location of the data. Best case is that the major computational arrays reside on the accelerator. Otherwise computational intensity of the accelerated kernel must be significant

Cray’s Hybrid Programming Environment supplies tools for addressing these issues
Three levels of Parallelism required

- Developers will continue to use MPI between nodes or sockets
- Developers must address using a shared memory programming paradigm on the node
- Developers must vectorize low level looping structures
- While there is a potential acceptance of new languages for addressing all levels directly. Most developers cannot afford this approach until they are assured that the new language will be accepted and the generated code is within a reasonable performance range
Possible Programming Models for the Node

- Cuda
- OpenCL
- Existing Fortran, C and C++ with extensions
  - Pthreads, Thread Building Blocks
  - Comment line directives
    - OpenMP extensions for Accelerators

All of these programming models require the application developer to replace MPI within the node – to develop Hybrid versions of the application
Task 1 – Identification of potential accelerator kernels

- Identify high level computational structures that account for a significant amount of time (95-99%)
  - To do this, one must obtain global runtime statistics of the application
    - High level call tree with subroutines and DO loops showing inclusive/exclusive time, min, max, average iteration counts.

- Tools that will be needed
  - Advanced instrumentation to measure
    - DO loop statistics, iteration counts, inclusive time
    - Routine level sampling and profiling
## Table 1: Profile by Function Group and Function

<table>
<thead>
<tr>
<th>Time%</th>
<th>Time</th>
<th>Imb.</th>
<th>Imb.</th>
<th>Calls</th>
<th>Group</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Time</td>
<td>Time%</td>
<td></td>
<td>Function</td>
</tr>
<tr>
<td>100.0%</td>
<td>50.553984</td>
<td>--</td>
<td>--</td>
<td>6922023.0</td>
<td>Total</td>
</tr>
<tr>
<td>52.1%</td>
<td>26.353695</td>
<td>--</td>
<td>--</td>
<td>6915004.0</td>
<td>USER</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Time%</th>
<th>Time</th>
<th>Imb.</th>
<th>Imb.</th>
<th>Calls</th>
<th>Group</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Time</td>
<td>Time%</td>
<td></td>
<td>Function</td>
</tr>
<tr>
<td>16.9%</td>
<td>8.540852</td>
<td>0.366647</td>
<td>4.1%</td>
<td>2592000.0</td>
<td>parabola_</td>
</tr>
<tr>
<td>8.0%</td>
<td>4.034867</td>
<td>0.222303</td>
<td>5.2%</td>
<td>288000.0</td>
<td>remap_</td>
</tr>
<tr>
<td>7.1%</td>
<td>3.612980</td>
<td>0.862830</td>
<td>19.3%</td>
<td>288000.0</td>
<td>riemann_</td>
</tr>
<tr>
<td>3.7%</td>
<td>1.859449</td>
<td>0.094075</td>
<td>4.8%</td>
<td>288000.0</td>
<td>ppmlr_</td>
</tr>
<tr>
<td>3.3%</td>
<td>1.666590</td>
<td>0.06495</td>
<td>3.7%</td>
<td>288000.0</td>
<td>evolve_</td>
</tr>
<tr>
<td>2.6%</td>
<td>1.315145</td>
<td>0.119832</td>
<td>8.4%</td>
<td>576000.0</td>
<td>paraset_</td>
</tr>
<tr>
<td>1.8%</td>
<td>0.923711</td>
<td>0.048359</td>
<td>5.0%</td>
<td>864000.0</td>
<td>volume_</td>
</tr>
<tr>
<td>1.8%</td>
<td>0.890751</td>
<td>0.06495</td>
<td>6.8%</td>
<td>288000.0</td>
<td>states_</td>
</tr>
<tr>
<td>1.4%</td>
<td>0.719636</td>
<td>0.079651</td>
<td>10.0%</td>
<td>288000.0</td>
<td>flatten_</td>
</tr>
<tr>
<td>1.0%</td>
<td>0.513454</td>
<td>0.019075</td>
<td>3.6%</td>
<td>864000.0</td>
<td>forces_</td>
</tr>
<tr>
<td>1.0%</td>
<td>0.504152</td>
<td>0.027139</td>
<td>5.1%</td>
<td>1000.0</td>
<td>sweepz_</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>sweepy_</td>
</tr>
<tr>
<td>37.9%</td>
<td>19.149499</td>
<td>--</td>
<td>--</td>
<td>3512.0</td>
<td>MPI</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Time%</th>
<th>Time</th>
<th>Imb.</th>
<th>Imb.</th>
<th>Calls</th>
<th>Group</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Time</td>
<td>Time%</td>
<td></td>
<td>Function</td>
</tr>
<tr>
<td>28.7%</td>
<td>14.487564</td>
<td>0.572138</td>
<td>3.8%</td>
<td>3000.0</td>
<td>mpi_alltoall</td>
</tr>
<tr>
<td>8.7%</td>
<td>4.391205</td>
<td>2.885755</td>
<td>39.7%</td>
<td>2.0</td>
<td>mpi_comm_split</td>
</tr>
<tr>
<td>10.0%</td>
<td>5.050780</td>
<td>--</td>
<td>--</td>
<td>3502.0</td>
<td>MPI_SYNC</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Time%</th>
<th>Time</th>
<th>Imb.</th>
<th>Imb.</th>
<th>Calls</th>
<th>Group</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Time</td>
<td>Time%</td>
<td></td>
<td>Function</td>
</tr>
<tr>
<td>6.9%</td>
<td>3.483206</td>
<td>1.813952</td>
<td>52.1%</td>
<td>3000.0</td>
<td>mpi_alltoall_ (sync)</td>
</tr>
<tr>
<td>3.1%</td>
<td>1.567285</td>
<td>0.606728</td>
<td>38.7%</td>
<td>501.0</td>
<td>mpi_allreduce_ (sync)</td>
</tr>
</tbody>
</table>
## Normal Profile – Using “setenv PAT_RT_HWPC 1”

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Time%</td>
<td>12.4%</td>
</tr>
<tr>
<td>Time</td>
<td>9.438486 secs</td>
</tr>
<tr>
<td>Imb. Time</td>
<td>0.851876 secs</td>
</tr>
<tr>
<td>Imb. Time%</td>
<td>8.3%</td>
</tr>
<tr>
<td>Calls</td>
<td>0.265M/sec 2592000.0 calls</td>
</tr>
<tr>
<td>PAPI_L1_DCM</td>
<td>42.908M/sec 419719824 misses</td>
</tr>
<tr>
<td>PAPI_TLB_DM</td>
<td>0.048M/sec 474094 misses</td>
</tr>
<tr>
<td>PAPI_L1_DCA</td>
<td>1067.727M/sec 10444336795 refs</td>
</tr>
<tr>
<td>PAPI_FP_OPS</td>
<td>1808.848M/sec 17693862446 ops</td>
</tr>
<tr>
<td>Average Time per Call</td>
<td>0.000004 secs</td>
</tr>
<tr>
<td>CrayPat Overhead : Time</td>
<td>75.3%</td>
</tr>
<tr>
<td>User time (approx)</td>
<td>9.782 secs 21520125183 cycles 100.0% Time</td>
</tr>
<tr>
<td>HW FP Ops / User time</td>
<td>1808.848M/sec 17693862446 ops 10.3%peak(DP)</td>
</tr>
<tr>
<td>HW FP Ops / WCT</td>
<td>1808.848M/sec</td>
</tr>
<tr>
<td>Computational intensity</td>
<td>0.82 ops/cycle 1.69 ops/ref</td>
</tr>
<tr>
<td>MFLOPS (aggregate)</td>
<td>7409042.08M/sec</td>
</tr>
<tr>
<td>TLB utilization</td>
<td>22030.09 refs/miss 43.028 avg uses</td>
</tr>
<tr>
<td>D1 cache hit,miss ratios</td>
<td>96.0% hits 4.0% misses</td>
</tr>
<tr>
<td>D1 cache utilization (misses)</td>
<td>24.88 refs/miss 3.111 avg hits</td>
</tr>
</tbody>
</table>
**Re-compiling with --hprofile_generate**

<table>
<thead>
<tr>
<th>Loop Incl Time</th>
<th>Loop Hit</th>
<th>Loop Trips Min</th>
<th>Loop Trips Max</th>
<th>Function</th>
<th>PE=HIDE</th>
</tr>
</thead>
<tbody>
<tr>
<td>51.107386</td>
<td>500</td>
<td>0</td>
<td>16</td>
<td></td>
<td>sweepx2_.LOOP.1.li.34</td>
</tr>
<tr>
<td>51.10682</td>
<td>8000</td>
<td>0</td>
<td>16</td>
<td></td>
<td>sweepx2_.LOOP.2.li.35</td>
</tr>
<tr>
<td>50.373481</td>
<td>500</td>
<td>0</td>
<td>16</td>
<td></td>
<td>sweepx1_.LOOP.1.li.34</td>
</tr>
<tr>
<td>50.372915</td>
<td>8000</td>
<td>0</td>
<td>16</td>
<td></td>
<td>sweepx1_.LOOP.2.li.35</td>
</tr>
<tr>
<td>12.480442</td>
<td>1000</td>
<td>0</td>
<td>16</td>
<td></td>
<td>sweepy_.LOOP.1.li.38</td>
</tr>
<tr>
<td>12.478967</td>
<td>16000</td>
<td>0</td>
<td>1</td>
<td></td>
<td>sweepy_.LOOP.2.li.39</td>
</tr>
<tr>
<td>11.949236</td>
<td>500</td>
<td>0</td>
<td>16</td>
<td></td>
<td>sweepz_.LOOP.05.li.54</td>
</tr>
<tr>
<td>11.948618</td>
<td>8000</td>
<td>0</td>
<td>1</td>
<td></td>
<td>sweepz_.LOOP.06.li.55</td>
</tr>
<tr>
<td>5.479066</td>
<td>288000</td>
<td>0</td>
<td>1031</td>
<td></td>
<td>riemann_.LOOP.2.li.63</td>
</tr>
<tr>
<td>3.082245</td>
<td>51168000</td>
<td>0</td>
<td>12</td>
<td></td>
<td>riemann_.LOOP.3.li.64</td>
</tr>
<tr>
<td>1.796424</td>
<td>2592000</td>
<td>0</td>
<td>1032</td>
<td></td>
<td>parabola_.LOOP.6.li.67</td>
</tr>
<tr>
<td>1.503023</td>
<td>2592000</td>
<td>0</td>
<td>1034</td>
<td></td>
<td>parabola_.LOOP.2.li.30</td>
</tr>
<tr>
<td>1.377911</td>
<td>2592000</td>
<td>0</td>
<td>1032</td>
<td></td>
<td>parabola_.LOOP.7.li.75</td>
</tr>
<tr>
<td>1.094964</td>
<td>2592000</td>
<td>0</td>
<td>1033</td>
<td></td>
<td>parabola_.LOOP.4.li.44</td>
</tr>
<tr>
<td>0.815105</td>
<td>288000</td>
<td>0</td>
<td>1025</td>
<td></td>
<td>remap_.LOOP.7.li.83</td>
</tr>
<tr>
<td>0.792899</td>
<td>2592000</td>
<td>0</td>
<td>1032</td>
<td></td>
<td>parabola_.LOOP.5.li.53</td>
</tr>
<tr>
<td>0.76888</td>
<td>2592000</td>
<td>0</td>
<td>1032</td>
<td></td>
<td>parabola_.LOOP.8.li.84</td>
</tr>
<tr>
<td>0.590497</td>
<td>128000</td>
<td>0</td>
<td>64</td>
<td></td>
<td>sweepx2_.LOOP.3.li.38</td>
</tr>
<tr>
<td>0.505536</td>
<td>288000</td>
<td>0</td>
<td>1031</td>
<td></td>
<td>riemann_.LOOP.1.li.44</td>
</tr>
<tr>
<td>0.478305</td>
<td>2592000</td>
<td>0</td>
<td>1034</td>
<td></td>
<td>parabola_.LOOP.3.li.36</td>
</tr>
<tr>
<td>0.465781</td>
<td>2592000</td>
<td>0</td>
<td>1035</td>
<td></td>
<td>parabola_.LOOP.1.li.24</td>
</tr>
<tr>
<td>0.463514</td>
<td>576000</td>
<td>0</td>
<td>1036</td>
<td></td>
<td>paraset_.LOOP.1.li.117</td>
</tr>
<tr>
<td>0.352512</td>
<td>288000</td>
<td>0</td>
<td>1032</td>
<td></td>
<td>states_.LOOP.2.li.64</td>
</tr>
<tr>
<td>0.338868</td>
<td>288000</td>
<td>0</td>
<td>1030</td>
<td></td>
<td>evolve_.LOOP.4.li.70</td>
</tr>
<tr>
<td>0.335398</td>
<td>288000</td>
<td>0</td>
<td>1026</td>
<td></td>
<td>remap_.LOOP.8.li.111</td>
</tr>
</tbody>
</table>
Re-compiling with `--hprofile_generate "pat_report --O callers"`

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>100.0%</td>
<td>117.646170</td>
<td>13549032.0</td>
<td>Total</td>
</tr>
<tr>
<td>-------------------------------------------------------</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>75.4%</td>
<td>88.723495</td>
<td>13542013.0</td>
<td>USER</td>
</tr>
<tr>
<td>-------------------------------------------------------</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10.7%</td>
<td>12.589734</td>
<td>2592000.0</td>
<td>parabola_</td>
</tr>
<tr>
<td>-------------------------------------------------------</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3.2%</td>
<td>3.708452</td>
<td>768000.0</td>
<td>sweepx2_.LOOP.2.li.35</td>
</tr>
<tr>
<td>3.1%</td>
<td>3.663423</td>
<td>768000.0</td>
<td>sweepx1_.LOOP.2.li.35</td>
</tr>
<tr>
<td>-------------------------------------------------------</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3.6%</td>
<td>4.229443</td>
<td>864000.0</td>
<td>ppmlr_</td>
</tr>
<tr>
<td>-------------------------------------------------------</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1.6%</td>
<td>1.880874</td>
<td>384000.0</td>
<td>sweepx2_.LOOP.2.li.35</td>
</tr>
<tr>
<td>1.6%</td>
<td>1.852820</td>
<td>384000.0</td>
<td>sweepx1_.LOOP.2.li.35</td>
</tr>
</tbody>
</table>

Cray Inc. SNL Workshop Apr 9-11
Task 2 Parallel Analysis, Scoping and Vectorization

- Investigate parallelizability of high level looping structures
  - Often times one level of loop is not enough, must have several parallel loops
  - User must understand what high level DO loops are in fact independent.
- Without tools, variable scoping of high level loops is very difficult
  - Loops must be more than independent, their variable usage must adhere to private data local to a thread or global shared across all the threads
- Investigate vectorizability of lower level Do loops
  - Cray compiler has been vectorizing complex codes for over 30 years
Current scoping tool, -homp_analyze, is meant to interface to a code restructuring GUI called “reveal”. This week we need to use cryptic output and massage it with editor/script.

In order to utilize scoping tool for loops that contain procedures the program library need to be employed

- -hwp –hpl=vhone.aid
  - This will do an initial pass of the code, checking for error and then at the load it will build the program library and perform the analysis

Compiler will be very conservative

- <object_message kind="warn">LastPrivate of array may be very expensive.</object_message>
Current output of –homp_analyze

<construct kind="loop" begin_line="54" end_line="119">
    <construct_message kind="warn">Call or I/O at line 100 of sweepz.f90</construct_message>
    <construct_message kind="warn">Call or I/O at line 84 of sweepz.f90</construct_message>
    <object state="known">
        <symbol name="dotflo"/>
        <scope source="compiler"> <shared/> </scope>
    </object>
    <object state="known">
        <symbol name="dt"/>
        <scope source="compiler"> <shared/> </scope>
    </object>
    <object state="known">
        <symbol name="dvol"/>
        <scope source="compiler"> <private first="true" last="true"/> </scope>
        <object_message kind="warn">LastPrivate of array may be very expensive.</object_message>
    </object>
    <object state="known">
        <symbol name="dx"/>
        <scope source="compiler"> <private first="true" last="true"/> </scope>
        <object_message kind="warn">LastPrivate of array may be very expensive.</object_message>
    </object>
    <object state="known">
        <symbol name="dx0"/>
        <scope source="compiler"> <private first="true" last="true"/> </scope>
        <object_message kind="warn">LastPrivate of array may be very expensive.</object_message>
    </object>
    <object state="known">
        <symbol name="e"/>
        <scope source="compiler"> <private first="true" last="true"/> </scope>
        <object_message kind="warn">LastPrivate of array may be very expensive.</object_message>
    </object>
</construct>
Main window of reveal
Scoping window

![OpenMP Construct window]

- **Name**: zyc, a, ai, amid, ar, b, bi, c, cdtdx, ci, ci, cift, crgh
- **Type**: Scalar
- **Scope**: Unknown, Private

**Options**
- **First/Last Private**: Enable First Private, Enable Last Private
- **Reduction**: None
- **Search**: 

**Buttons**
- **Dump Data**, **Close**
Private Variables in module, need to use Threadprivate

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

Reduction variable down callchain, need to use
!$OMP CRITICAL;!$OMP END CRITICAL

hdt   = 0.5*dt
do n = nmin-4, nmax+4
   Cdtdx (n) = sqrt(gam*p(n)/r(n))/(dx(n)*radius)
enddo
!$omp critical
do n = nmin-4, nmax+4
   svel  = max(svel,Cdtdx(n))
enddo
!$omp end critical
do n = nmin-4, nmax+4
   Cdtdx (n) = Cdtdx(n)*hdt
   fCdtdx(n) = 1. - fourthd*Cdtdx(n)
enddo
Differences in OpenMP version

module zone

! (formerly zone.h) global (3D) data arrays

INTEGER, PARAMETER :: imax = 64, jmax =1024, kmax = 1024  ! Memory dimens
INTEGER, PARAMETER :: pey = 64, pdez = 64   ! number of MPI tasks
!!!!# for 2D: ^^ IF kmax=1, MUST HAVE pez=1 !!!!!##
INTEGER, PARAMETER :: nvar = 6  ! number of primitive fluid var

INTEGER :: isy, isz, js, ks, Ya2abuff_size, Za2abuff_size
INTEGER :: npe, npey, npez, mype, mypey, mypez  ! # of pes and local pe num
INTEGER :: MPI_COMM_ROW, MPI_COMM_COL
INTEGER :: VHI_DATATYPE

INTEGER :: ngeomx, ngeomy, ngeonz  ! XYZ Geometry flag
INTEGER :: nleftx, nlefty, nleftz  ! XYZ Lower Boundary Condition
INTEGER :: nrightx, nrighty, nrightz  ! XYZ Upper Boundary Condition

REAL, DIMENSION(imax,jmax,pey,kmax,pdez) :: zro, zpr, zu, zu, zuf
REAL, DIMENSION(imax) :: zxa, zdx, zxc
REAL, DIMENSION(jmax) :: zya, zdy, zyc
REAL, DIMENSION(kmax) :: zza, zdz, zzc

REAL, DIMENSION(nvar,kmax,pdez,jmax,pey,imax) :: send1
REAL, DIMENSION(nvar,kmax,pdez,jmax,pey,imax) :: send2
REAL, DIMENSION(nvar,jmax,pey,kmax,pdez,imax) :: send3
REAL, DIMENSION(nvar,jmax,pey,kmax,pdez,imax) :: send4
Differences in OpenMP version

```fortran
module sweeysize
!
! Dimension of 1D sweeps. maxsweep must be as long as the longest of the
! 3D arrays PLUS the ghost zones:
! ie, must have maxsweep >= max(imax,jmax,kmax) + 12
!

integer, parameter :: maxsweep=1050
end module sweeysize

module sweeps
!
! Data structures used in 1D sweeps, dimensioned maxsweep (set in sweeysize)
!
use sweeysize

character(len=1) :: sweep ! direction of
integer :: nmin, nmax, ngeom, nleft, nright ! number of fir

real, dimension(maxsweep) :: r, p, e, q, u, v, w ! fluid variabl
real, dimension(maxsweep) :: xa, xa0, dx, dx0, dvol ! coordinate va
real, dimension(maxsweep) :: f, flat ! flattening pa
real, dimension(maxsweep,5) :: para ! parabolic int
real :: radius, theta, stheta
end module sweeps
```

```fortran
module sweeysize
!
! Dimension of 1D sweeps. maxsweep must be as long as the longest of the
! 3D arrays PLUS the ghost zones:
! ie, must have maxsweep >= max(imax,jmax,kmax) + 12
!

integer, parameter :: maxsweep=1050
end module sweeysize

module sweeps
!
! Data structures used in 1D sweeps, dimensioned maxsweep (set in sweeysize)
!
use sweeysize

character(len=1) :: sweep ! direction of
integer :: nmin, nmax, ngeom, nleft, nright ! number of fir

real, dimension(maxsweep) :: r, p, e, q, u, v, w ! fluid variabl
real, dimension(maxsweep) :: xa, xa0, dx, dx0, dvol ! coordinate va
real, dimension(maxsweep) :: f, flat ! flattening pa
real, dimension(maxsweep,5) :: para ! parabolic int
real :: radius, theta, stheta
end module sweeps
```
Differences in OpenMP version

```fortran
#endif

do k = 1, ks
  do j = 1, js
    ! Put state variables into 1D arrays, padding with 6 ghost zones
    do i = 1,imax
      n = i + 6
      r (n) = zro(i,j,k)
      p (n) = zpr(i,j,k)
      u (n) = zux(i,j,k)
      v (n) = zuv(i,j,k)
      w (n) = zuw(i,j,k)
      f (n) = zfl(i,j,k)

      xa0(n) = zxa(i)
      dx0(n) = zdx(i)
      xa (n) = zxa(i)
      dx (n) = zdx(i)
      p (n) = max(smallp,p(n))
      e (n) = p(n)/(r(n)*gamma)+0.5*(u(n)**2+v(n)**2+w(n)**2)
  enddo

  ! Do 1D hydro update using PPMLR
  call ppmlr
  ifdef DEBUGX
    print *, 'In sweep1', svel
  endif
  !$omp parallel do private(i,j,k,n)
  do k = 1, ks
    do j = 1, js
      ! Put state variables into 1D arrays, padding with 6 ghost zones
      do i = 1,imax
        n = i + 6
        r (n) = zro(i,j,k)
        p (n) = zpr(i,j,k)
        u (n) = zux(i,j,k)
        v (n) = zuv(i,j,k)
        w (n) = zuw(i,j,k)
        f (n) = zfl(i,j,k)

        xa0(n) = zxa(i)
        dx0(n) = zdx(i)
        xa (n) = zxa(i)
        dx (n) = zdx(i)
        p (n) = max(smallp,p(n))
        e (n) = p(n)/(r(n)*gamma)+0.5*(u(n)**2+v(n)**2+w(n)**2)
      enddo

      ! Do 1D hydro update using PPMLR
      call ppmlr
      ifdef DEBUGX
        print *, 'In sweep1', svel
      endif
```

Cray Inc. SNL Workshop Apr 9-11
Differences in runtime

All MPI on 4096 cores 43.01 seconds
Hybrid 256 nodesx16 threads 45.05 seconds
Task 3 Moving from OpenMP to OpenACC

- Things that are different between OpenMP and OpenACC
  - Cannot have CRITICAL REGION down callchain
  - Cannot have THREADPRIVATE
  - Vectorization is much more important
  - Cache/Memory Optimization much more important
  - No EQUIVALENCE
- Currently both OpenMP and OpenACC must be included in the source

```c
#ifdef GPU
!$acc parallel loop private( k,j,i,n,r, p, e, q, u, v, w, svel0,&
!$acc& xa, xa0, dx, dx0, dvol, f, flat, para, radius, theta, stheta)&
!$acc& reduction(max:svel)
#endif

#else
!$omp parallel do private( k,j,i,n,r, p, e, q, u, v, w, svel0,&
!$omp& xa, xa0, dx, dx0, dvol, f, flat, para, radius, theta, stheta)&
!$omp& reduction(max:svel)
#endif
```
Changes to remove THREADPRIVATE

!$omp parallel do private( k, j, i, n, r, p, e, q, u, v, w, svel0,&
!$omp& xa, xa0, dx, dx0, dvol, f, flat, para, radius, theta, stheta)&
!$omp& reduction(max:svel)
#endif

do k = 1, ks
  do j = 1, js
    theta=0.0
    stheta=0.0
    radius=0.0

    ! Put state variables into 1D arrays, padding with 6 ghost zones
    do i = 1,imax
      n = i + 6
      r (n) = zro(i,j,k)
      p (n) = zpr(i,j,k)
      u (n) = zux(i,j,k)
      v (n) = zuy(i,j,k)
      w (n) = zuz(i,j,k)
      f (n) = zfl(i,j,k)

      xa0(n) = xza(i)
      dx0(n) = zdx(i)
      xa (n) = zxa(i)
      dx (n) = zdx(i)
      p (n) = max(smallp,p(n))
      e (n) = p(n)/(r(n)*gamma)+0.5*(u(n)**2+v(n)**2+w(n)**2)
    enddo
  enddo
enddo
Changes to remove THREADPRIVATE and CRITICAL REGION

```fortran
xa (n) = zxa(i)
dx (n) = zdx(i)
p (n) = max(smallp,p(n))
e (n) = p(n)/(r(n)*gamm)+0.5*(u(n)**2+v(n)**2+w(n)**2)
enddo

! Do 1D hydro update using PPMLR

   call ppmlr (svel0, sweep, nmin, nmax, ngeom, nleft, nright, r, p, e, u, u0, u, 
               xa, xa0, dx, dx0, dvol, f, flat, para, radius, theta, stheta)

   do n = nmin-4, nmax+4
       svel = max(svel, svel0(n))
   enddo
#endif DEBUGX

   print *, 'In sweep1', svel
#endif

! Put updated values into send buffer for sweepy, dropping ghost zones
   do i = 1,imax
       n = i + 6
       sendl(1, k, j, i) = r(n)
       sendl(2, k, j, i) = p(n)
       sendl(3, k, j, i) = u(n)
       sendl(4, k, j, i) = v(n)
       sendl(5, k, j, i) = w(n)
       sendl(6, k, j, i) = f(n)
   enddo
```

```fortran
xa (n) = zxa(i)
dx (n) = zdx(i)
p (n) = max(smallp,p(n))
e (n) = p(n)/(r(n)*gamm)+0.5*(u(n)**2+v(n)**2+w(n)**2)
enddo

! Do 1D hydro update using PPMLR

   call ppmlr

#endif DEBUGX

   print *, 'In sweep1', svel
#endif

! Put updated values into send buffer for sweepy, dropping ghost zones
   do i = 1,imax
       n = i + 6
       sendl(1, k, j, i) = r(n)
       sendl(2, k, j, i) = p(n)
       sendl(3, k, j, i) = u(n)
       sendl(4, k, j, i) = v(n)
       sendl(5, k, j, i) = w(n)
       sendl(6, k, j, i) = f(n)
   enddo
```
Differences in runtime

All MPI on 4096 cores 43.01 seconds
Hybrid 256 nodesx16 threads 45.05 seconds
Rest Hybrid 256x16 threads 48.03 seconds
Differences in runtime

- All MPI on 4096 cores: 43.01 seconds
- Hybrid 256 nodesx16 threads: 45.05 seconds
- Rest Hybrid 256x16 threads: 47.58 seconds
**Differences in runtime**

- **All MPI on 4096 cores**: 43.01 seconds
- **Hybrid 256 nodesx16 threads**: 45.05 seconds
- **Rest Hybrid 256x16 threads**: 47.58 seconds
Developing efficient OpenMP regions is not an easy task; however, the performance will definitely be worth the effort.

The next step will be to add OpenACC directives to allow for compilation of the same OpenMP regions to accelerator by the compiler.

- With OpenACC data transfers between multi-core socket and the accelerator as well as utilization of registers and shared memory can be optimized.
- With OpenACC user can control the utilization of the accelerator memory and functional units.
Task 3 Correctness Debugging

- Run transformed application on the accelerator and investigate the correctness and performance
  - Run as OpenMP application on multi-core socket
    - Use multi-core socket Debugger - DDT
  - Run as Hybrid multi-core application across multi-core socket and accelerator
- Tools That will be needed
  - Information that was supplied by the directives/user’s interaction with the compiler
The only requirement for using the !$acc$ parallel loop is that the user specify the private variables and the compiler will do the rest.

- If subroutine calls are contained in the loop, -hwp must be used.

```
#ifndef GPU
!$acc parallel loop private( k,j,i,n,r, p, e, q, u, v, w, svel0,&
!$acc& xa, xa0, dx, dx0, dvol, f, flat, para,radius, theta, stheta)&
!$acc& reduction(max:svel)
#else
!$omp parallel do private( k,j,i,n,r, p, e, q, u, v, w, svel0,&
!$omp& xa, xa0, dx, dx0, dvol, f, flat, para,radius, theta, stheta)&
!$omp& reduction(max:svel)
#endif
```

- The Compiler will then show:
  - All data motion required to run the loop on the accelerator.
  - Show how it handled the looping structures in the parallel region.
#ifndef GPU
  !$acc parallel loop private( k,j,i,n,r, p, e, q, u, v, w, svel0,&
  xa, xa0, dx, dx0, dvol, f, flat, para, radius, theta, stheta)&
  !$acc&
  reduction(max:svel)
#endif

else
  !$omp parallel do private( k,j,i,n,r, p, e, q, u, v, w, svel0,&
  xa, xa0, dx, dx0, dvol, f, flat, para, radius, theta, stheta)&
  !$omp&
  reduction(max:svel)
#endif


do k = 1, ks
  theta=0.0
  stheta=0.0
  radius=0.0
  do i = 1,imax
    n = i + 6
    r  (n) = zro(i,j,k)
    p  (n) = zpr(i,j,k)
    u  (n) = zux(i,j,k)
    v  (n) = zuy(i,j,k)
    w  (n) = zuz(i,j,k)
    f  (n) = zfl(i,j,k)
    xa0(n) = zxa(i)
    dx0(n) = zdx(i)
    xa (n) = zxa(i)
    dx (n) = zdx(i)
    p  (n) = max(smallp,p(n))
    e  (n) = p(n)/(r(n)*gamm)+0.5*(u(n)**2+v(n)**2+w(n)**2)
  enddo

! Do 1D hydro update using PPMLR

call ppmlr (svel0, sweep, nmin, nmax, ngeom, nleft, nright,r, p, e, q, u, v, w,&
xa, xa0, dx, dx0, dvol, f, flat, para, radius, theta, stheta)
A region starting at line 46 and ending at line 104 was placed on the accelerator.

If not already present: allocate memory and copy whole array "zro" to accelerator, free at line 104 (acc_copyin).

If not already present: allocate memory and copy whole array "zpr" to accelerator, free at line 104 (acc_copyin).

If not already present: allocate memory and copy whole array "zux" to accelerator, free at line 104 (acc_copyin).

If not already present: allocate memory and copy whole array "zuy" to accelerator, free at line 104 (acc_copyin).

If not already present: allocate memory and copy whole array "zuz" to accelerator, free at line 104 (acc_copyin).

If not already present: allocate memory and copy whole array "zfl" to accelerator, free at line 104 (acc_copyin).

If not already present: allocate memory and copy whole array "send1" to accelerator, copy back at line 104 (acc_copy).
Task 5 Fine tuning of accelerated program

- Understand current performance bottlenecks
  - Is data transfer between multi-core socket and accelerator a bottleneck?
  - Is shared memory and registers on the accelerator being used effectively?
  - Is the accelerator code utilizing the MIMD parallel units?
    - Is the shared memory parallelization load balanced?
  - Is the low level accelerator code vectorized?
    - Are the memory accesses effectively utilizing the memory bandwidth?
Table 1: Time and Bytes Transferred for Accelerator Regions

<table>
<thead>
<tr>
<th>Acc Time%</th>
<th>Acc Time</th>
<th>Host Time</th>
<th>Acc Copy In (MBytes)</th>
<th>Acc Copy Out (MBytes)</th>
<th>Calls</th>
<th>Function</th>
<th>PE=HIDE</th>
<th>Thread=HIDE</th>
</tr>
</thead>
<tbody>
<tr>
<td>100.0%</td>
<td>58.363</td>
<td>67.688</td>
<td>24006.022</td>
<td>16514.196</td>
<td>14007</td>
<td>Total</td>
<td></td>
<td></td>
</tr>
<tr>
<td>30.3%</td>
<td>17.697</td>
<td>0.022</td>
<td>--</td>
<td>--</td>
<td>1000</td>
<td>sweepy_.ACC_KERNEL@li.47</td>
<td></td>
<td></td>
</tr>
<tr>
<td>22.0%</td>
<td>12.827</td>
<td>0.010</td>
<td>--</td>
<td>--</td>
<td>500</td>
<td>sweepx2_.ACC_KERNEL@li.46</td>
<td></td>
<td></td>
</tr>
<tr>
<td>21.2%</td>
<td>12.374</td>
<td>0.013</td>
<td>--</td>
<td>--</td>
<td>500</td>
<td>sweepz_.ACC_KERNEL@li.67</td>
<td></td>
<td></td>
</tr>
<tr>
<td>14.0%</td>
<td>8.170</td>
<td>0.013</td>
<td>--</td>
<td>--</td>
<td>500</td>
<td>sweepx1_.ACC_KERNEL@li.46</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3.9%</td>
<td>2.281</td>
<td>1.161</td>
<td>12000.004</td>
<td>--</td>
<td>1000</td>
<td>sweepy_.ACC_COPY@li.47</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2.0%</td>
<td>1.162</td>
<td>0.601</td>
<td>6000.002</td>
<td>--</td>
<td>500</td>
<td>sweepz_.ACC_COPY@li.67</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1.6%</td>
<td>0.953</td>
<td>0.014</td>
<td>--</td>
<td>6000.004</td>
<td>1000</td>
<td>sweepy_.ACC_COPY@li.129</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1.0%</td>
<td>0.593</td>
<td>0.546</td>
<td>3000.002</td>
<td>--</td>
<td>500</td>
<td>sweepx1_.ACC_COPY@li.46</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1.0%</td>
<td>0.591</td>
<td>0.533</td>
<td>3000.002</td>
<td>--</td>
<td>500</td>
<td>sweepx2_.ACC_COPY@li.46</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0.8%</td>
<td>0.494</td>
<td>0.015</td>
<td>--</td>
<td>3000.002</td>
<td>500</td>
<td>sweepx2_.ACC_COPY@li.107</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0.8%</td>
<td>0.485</td>
<td>0.007</td>
<td>--</td>
<td>3000.002</td>
<td>500</td>
<td>sweepx1_.ACC_COPY@li.104</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0.8%</td>
<td>0.477</td>
<td>0.007</td>
<td>--</td>
<td>3000.002</td>
<td>500</td>
<td>sweepz_.ACC_COPY@li.150</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0.4%</td>
<td>0.250</td>
<td>0.016</td>
<td>--</td>
<td>1503.174</td>
<td>500</td>
<td>vhone_.ACC_COPY@li.251</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0.0%</td>
<td>0.005</td>
<td>0.005</td>
<td>6.012</td>
<td>--</td>
<td>1</td>
<td>vhone_.ACC_COPY@li.205</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0.0%</td>
<td>0.001</td>
<td>0.000</td>
<td>--</td>
<td>6.012</td>
<td>1</td>
<td>vhone_.ACC_COPY@li.283</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0.0%</td>
<td>0.001</td>
<td>0.000</td>
<td>--</td>
<td>5.000</td>
<td>1</td>
<td>vhone_.ACC_COPY@li.266</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Differences in runtime

- All MPI on 4096 cores: 43.01 seconds
- Hybrid 256 nodesx16 threads: 45.05 seconds
- Rest Hybrid 256x16 threads: 47.58 seconds
- OpenACC 256xgpu: 105.92 seconds
Task 4 Fine tuning of accelerated program

- Tools that will be needed:
  - Compiler feedback on parallelization and vectorization of input application
  - Hardware counter information from the accelerator to identify bottlenecks in the execution of the application.
    - Information on memory utilization
    - Information on performance of SIMT units

Several other vendors are supplying similar performance gathering tools
• Craypat profiling
  • Tracing: "pat_build -u <executable>" (can do APA sampling first)
  • "pat_report -O accelerator <.xf file>"; -T also useful
    • Other pat_report tables (as of perftools/5.2.1.7534)
      • acc_fu flat table of accelerator events
      • acc_time call tree sorted by accelerator time
      • acc_time_fu flat table of accelerator events sorted by accelerator time
      • acc_show_by_ct regions and events by calltree sorted alphabetically
Run and gather runtime statistics

Table 1: Profile by Function Group and Function

<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>83.277477</td>
<td>--</td>
<td>--</td>
<td>851.0</td>
<td>Total</td>
</tr>
<tr>
<td>51.3%</td>
<td>42.762837</td>
<td>--</td>
<td>--</td>
<td>703.0</td>
<td>ACCELERATOR</td>
</tr>
<tr>
<td>18.8%</td>
<td>15.672371</td>
<td>1.146276</td>
<td>7.3%</td>
<td>20.0</td>
<td>recolor_.SYNC_COPY@li.790 ➜ not good</td>
</tr>
<tr>
<td>16.3%</td>
<td>13.585707</td>
<td>0.404190</td>
<td>3.1%</td>
<td>20.0</td>
<td>recolor_.SYNC_COPY@li.793 ➜ not good</td>
</tr>
<tr>
<td>7.5%</td>
<td>6.216010</td>
<td>0.873830</td>
<td>13.1%</td>
<td>20.0</td>
<td>lbm3d2p_d_.ASYNC_KERNEL@li.116</td>
</tr>
<tr>
<td>1.6%</td>
<td>1.337119</td>
<td>0.193826</td>
<td>13.5%</td>
<td>20.0</td>
<td>lbm3d2p_d_.ASYNC_KERNEL@li.119</td>
</tr>
<tr>
<td>1.6%</td>
<td>1.322690</td>
<td>0.059387</td>
<td>4.6%</td>
<td>1.0</td>
<td>lbm3d2p_d_.ASYNC_COPY@li.100</td>
</tr>
<tr>
<td>1.0%</td>
<td>0.857149</td>
<td>0.245369</td>
<td>23.7%</td>
<td>20.0</td>
<td>collisionb_.ASYNC_KERNEL@li.586</td>
</tr>
<tr>
<td>1.0%</td>
<td>0.822911</td>
<td>0.172468</td>
<td>18.5%</td>
<td>20.0</td>
<td>lbm3d2p_d_.ASYNC_KERNEL@li.114</td>
</tr>
<tr>
<td>0.9%</td>
<td>0.786618</td>
<td>0.386807</td>
<td>35.2%</td>
<td>20.0</td>
<td>injection_.ASYNC_KERNEL@li.1119</td>
</tr>
<tr>
<td>0.9%</td>
<td>0.727451</td>
<td>0.221332</td>
<td>24.9%</td>
<td>20.0</td>
<td>lbm3d2p_d_.ASYNC_KERNEL@li.118</td>
</tr>
</tbody>
</table>
Keep data on the accelerator with acc_data region

```fortran
!$acc data copyin(cix,ci1,ci2,ci3,ci4,ci5,ci6,ci7,ci8,ci9,ci10,ci11,&
!$acc& ci12,ci13,ci14,r,b,uxyz,coll,rho,grad,index_max,index,&
!$acc& ciy,ciz,wet,np,streaming_sbuf1, &
!$acc& streaming_sbuf1,streaming_sbuf2,streaming_sbuf4,streaming_sbuf5,&
!$acc& streaming_sbuf7s,streaming_sbuf8s,streaming_sbuf9n,streaming_sbuf10s,&
!$acc& streaming_sbuf11n,streaming_sbuf12n,streaming_sbuf13s,streaming_sbuf14n,&
!$acc& streaming_sbuf7e,streaming_sbuf8w,streaming_sbuf9e,streaming_sbuf10e,&
!$acc& streaming_sbuf11w,streaming_sbuf12w,streaming_sbuf13w,streaming_sbuf14w, &
!$acc& streaming_rbuf1,streaming_rbuf2,streaming_rbuf4,streaming_rbuf5,&
!$acc& streaming_rbuf7n,streaming_rbuf8n,streaming_rbuf9s,streaming_rbuf10n,&
!$acc& streaming_rbuf11s,streaming_rbuf12s,streaming_rbuf13n,streaming_rbuf14s,&
!$acc& streaming_rbuf7w,streaming_rbuf8e,streaming_rbuf9w,streaming_rbuf10w,&
!$acc& streaming_rbuf11e,streaming_rbuf12w,streaming_rbuf13e,streaming_rbuf14e, &
!$acc& send_e,send_w,send_n,send_s,recv_e,recv_w,recv_n,recv_s)
  do ii=1,ntimes
    o o o
    call set_boundary_macro_press2
    call set_boundary_micro_press
    call collisiona
    call collisionb
    call recolor
```
Now when we do communication we have to update the host

```fortran
!$acc parallel_loop private(k,j,i)
   do j=0,local_ly-1
      do i=0,local_lx-1
         if (cell(i,j,0)==1) then
            grad (i,j,-1) = (1.0d0-wet)*db*press
         else
            grad (i,j,-1) = db*press
         end if
         grad (i,j,lz) = grad(i,j,lz-1)
      end do
   end do
!$acc end parallel_loop
!$acc update host(grad)
   call mpi_barrier(mpi_comm_world,ierr)
   call grad_exchange
!$acc update device(grad)
```

But we would rather not send the entire grad array back – how about
Packing the buffers on the accelerator

```fortran
!$acc data present(grad, recv_w, recv_e, send_e, send_w, recv_n,&
                   recv_s, send_n, send_s)
!$acc parallel_loop
   do k=-1,1z
     do j=-1,local_ly
       send_e(j,k) = grad(local_lx-1,j        ,k)
       send_w(j,k) = grad(0         ,j        ,k)
     end do
   end do
!$acc end parallel_loop
!$acc update host(send_e,send_w)
   call mpi_irecv(recv_w, bufsize(2),mpi_double_precision,w_id, &
                   tag(25),mpi_comm_world,irequest_in(25),ierr)
   call mpi_isend(send_w, bufsize(2),mpi_double_precision,w_id, &
                  tag(26),& mpi_comm_world,irequest_out(26),ierr)
   call mpi_waitall(2,irequest_in(25),istatus_req,ierr)
   call mpi_waitall(2,irequest_out(25),istatus_req,ierr)
!$acc update device(recv_e,recv_w)
!$acc parallel
!$acc loop
   do k=-1,1z
     do j=-1,local_ly
       grad(local_lx,j        ,k) = recv_e(j,k)
       grad(-1        ,j        ,k) = recv_w(j,k)
     end do
   end do
```
## Final Profile - bulk of time in kernel execution

<p>| | | | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>37.9%</td>
<td>236.592782</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>11403.0</td>
</tr>
<tr>
<td>ACCELERATOR</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>15.7%</td>
<td>98.021619</td>
<td>43.078137</td>
<td>31.0%</td>
<td>200.0</td>
<td>lbm3d2p_d_.ASYNC_KERNEL@li.129</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3.7%</td>
<td>23.359080</td>
<td>2.072147</td>
<td>8.3%</td>
<td>200.0</td>
<td>lbm3d2p_d_.ASYNC_KERNEL@li.127</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3.6%</td>
<td>22.326085</td>
<td>1.469419</td>
<td>6.3%</td>
<td>200.0</td>
<td>lbm3d2p_d_.ASYNC_KERNEL@li.132</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3.0%</td>
<td>19.035232</td>
<td>1.464608</td>
<td>7.3%</td>
<td>200.0</td>
<td>collisionb_.ASYNC KERNEL@li.599</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2.6%</td>
<td>16.216648</td>
<td>3.505232</td>
<td>18.1%</td>
<td>200.0</td>
<td>lbm3d2p_d_.ASYNC KERNEL@li.131</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2.5%</td>
<td>15.401916</td>
<td>8.093716</td>
<td>35.0%</td>
<td>200.0</td>
<td>injection_.ASYNC KERNEL@li.1116</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1.9%</td>
<td>11.734026</td>
<td>4.488785</td>
<td>28.1%</td>
<td>200.0</td>
<td>recolor_.ASYNC KERNEL@li.786</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0.9%</td>
<td>5.530201</td>
<td>2.132243</td>
<td>28.3%</td>
<td>200.0</td>
<td>collisionb_.SYNC COPY@li.593</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0.8%</td>
<td>4.714995</td>
<td>0.518495</td>
<td>10.1%</td>
<td>200.0</td>
<td>collisionb_.SYNC COPY@li.596</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0.6%</td>
<td>3.738615</td>
<td>2.986891</td>
<td>45.1%</td>
<td>200.0</td>
<td>collisionb_.ASYNC KERNEL@li.568</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0.4%</td>
<td>2.656962</td>
<td>0.454093</td>
<td>14.8%</td>
<td>1.0</td>
<td>lbm3d2p_d_.ASYNC COPY@li.100</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0.4%</td>
<td>2.489231</td>
<td>2.409892</td>
<td>50.0%</td>
<td>200.0</td>
<td>streaming_exchange_.ASYNC COPY@li.810</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0.4%</td>
<td>2.487132</td>
<td>2.311190</td>
<td>48.9%</td>
<td>200.0</td>
<td>streaming_exchange_.ASYNC COPY@li.625</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0.2%</td>
<td>1.322791</td>
<td>0.510645</td>
<td>28.3%</td>
<td>200.0</td>
<td>streaming_exchange_.SYNC COPY@li.622</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0.2%</td>
<td>1.273771</td>
<td>0.288743</td>
<td>18.8%</td>
<td>200.0</td>
<td>streaming_exchange_.SYNC COPY@li.574</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0.2%</td>
<td>1.212260</td>
<td>0.290853</td>
<td>20.0%</td>
<td>200.0</td>
<td>streaming_exchange_.SYNC COPY@li.759</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0.2%</td>
<td>1.208250</td>
<td>0.422182</td>
<td>26.3%</td>
<td>200.0</td>
<td>streaming_exchange_.SYNC COPY@li.806</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0.1%</td>
<td>0.696120</td>
<td>0.442372</td>
<td>39.5%</td>
<td>200.0</td>
<td>streaming_exchange_.ASYNC KERNEL@li.625</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0.1%</td>
<td>0.624982</td>
<td>0.379697</td>
<td>38.4%</td>
<td>200.0</td>
<td>streaming_exchange_.ASYNC KERNEL@li.525</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Useful tools

- Compiler feedback:
  - `-ra` to generate `*.lst` loopmark files (equivalent for C)
  - `-rd` to generate `*.cg` and `*.opt` files
    - `*.cg` useful to understand synchronisation points (CAF and ACC)
  - "ptxas -v *.ptx" gives information on register and shared memory usage (no way yet for user to adjust this)

- Runtime feedback (no recompilation needed)
  - "export CRAY_ACC_DEBUG=[1,2,3]" commentary to STDERR
  - NVIDIA compute profiler works with CUDA and directives
    - "export COMPUTE_PROFILE=1"
    - gives information on timings and occupancy in separate file
      - "more /opt/nvidia/cuda/<version>/doc/Compute_Profiler.txt" for documentation
      - Vince Graziano has a great script for summarising the output
Cray GPU Programming Environment

- Objective: Enhance productivity related to porting applications to hybrid multi-core systems
- Four core components
  - Cray Statistics Gathering Facility on host and GPU
  - Cray Optimization Explorer – Scoping Tools (COE)
  - Cray Compilation Environment (CCE)
  - Cray GPU Libraries
Titan: Early Science Applications

**WL-LSMS**
Role of material disorder, statistics, and fluctuations in nanoscale materials and systems.

**S3D**
How are going to efficiently burn next generation diesel/bio fuels?

**PFLOTRAN**
Stability and viability of large scale CO$_2$ sequestration; predictive containment groundwater transport

**LAMMPS**
Biofuels: An atomistic model of cellulose (blue) surrounded by lignin molecules comprising a total of 3.3 million atoms. Water not shown.

**CAM / HOMME**
Answer questions about specific climate change adaptation and mitigation scenarios; realistically represent features like precipitation patterns/statistics and tropical storms

**Denovo**
Unprecedented high-fidelity radiation transport calculations that can be used in a variety of nuclear energy and technology applications.
Structured Cartesian mesh flow solver

Solves compressible reacting Navier-Stokes, energy and species conservation equations.
  - 8th order explicit finite difference method
  - 4th order Runge-Kutta integrator with error estimator

Detailed gas-phase thermodynamic, chemistry and molecular transport property evaluations

Lagrangian particle tracking

MPI-1 based spatial decomposition and parallelism

Fortran code. Does not need linear algebra, FFT or solver libraries.

Developed and maintained at CRF, Sandia (Livermore) with BES and ASCR sponsorship. PI – Jacqueline H. Chen (jhchen@sandia.gov)
A benchmark problem was defined to closely resemble the target simulation:

- 52 species n-heptane chemistry and $48^3$ grid points per node

- $48^3 \times 18,500$ nodes = 2 billion grid points
- Target problem would take two months on today’s Jaguar

- Code was benchmarked and profiled on dual-hex core XT5
- Several kernels identified and extracted into stand-alone driver programs
Acceleration Strategy

Team:

Ramanan Sankaran ORNL
Ray Grout NREL
John Levesque Cray

Goals:

- Convert S3D to a hybrid multi-core application suited for a multi-core node with or without an accelerator.
- Be able to perform the computation entirely on the accelerator.
  - Arrays and data able to reside entirely on the accelerator.
  - Data sent from accelerator to host CPU for halo communication, I/O and monitoring only.

Strategy:

- To program using both hand-written and generated code.
  - Hand-written and tuned CUDA*.
  - Automated Fortran and CUDA generation for chemistry kernels
  - Automated code generation through compiler directives
- S3D is now a part of Cray’s compiler development test cases
## Original S3D

<table>
<thead>
<tr>
<th>Time Step</th>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Runge K</td>
<td>Solve_Drive</td>
</tr>
<tr>
<td></td>
<td>Runge K</td>
<td>Integrate</td>
</tr>
<tr>
<td></td>
<td>Runge K</td>
<td>RHS</td>
</tr>
<tr>
<td></td>
<td>Runge K</td>
<td>calc_temp</td>
</tr>
<tr>
<td></td>
<td>Runge K</td>
<td>Compute Grads</td>
</tr>
<tr>
<td></td>
<td>Runge K</td>
<td>Diffusive Flux</td>
</tr>
<tr>
<td></td>
<td>Runge K</td>
<td>Derivatives</td>
</tr>
<tr>
<td></td>
<td>Runge K</td>
<td>reaction rates</td>
</tr>
</tbody>
</table>
### Table 1: Profile by Function Group and Function

<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></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>PE=HIDE</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Thread=HIDE</td>
</tr>
<tr>
<td>100.0%</td>
<td>284.732812</td>
<td>--</td>
<td>--</td>
<td>156348682.1</td>
<td>Total</td>
</tr>
<tr>
<td>92.1%</td>
<td>262.380782</td>
<td>--</td>
<td>--</td>
<td>155578796.1</td>
<td>USER</td>
</tr>
</tbody>
</table>

| 12.4%   | 35.256420 | 0.237873 | 0.7%      | 391200.0 | ratt_i_.LOOPS |
| 9.6%    | 27.354247 | 0.186752 | 0.7%      | 391200.0 | ratx_i_.LOOPS |
| 7.7%    | 21.911069 | 1.037701 | 4.5%      | 1562500.0 | mcedif_.LOOPS |
| 5.4%    | 15.247551 | 2.389440 | 13.6%     | 35937500.0 | mceval4_ |
| 5.2%    | 14.908749 | 0.237873 | 0.7%      | 391200.0 | rhsf_.LOOPS |
| 4.7%    | 13.495681 | 0.237873 | 0.7%      | 391200.0 | mceval4_.LOOPS |
| 4.6%    | 12.985353 | 0.620839 | 4.6%      | 701.0 | calc_temp$thermchem_m_.LOOPS |
| 4.3%    | 12.274200 | 0.167054 | 1.3%      | 1562500.0 | mcevis_new$transport_m_.LOOPS |
| 4.0%    | 11.363281 | 0.606625 | 5.1%      | 600.0 | computespeciesdiff$transport_m_.LOOPS |
| 2.9%    | 8.257434  | 0.743004 | 8.3%      | 21921875.0 | mixcp$thermchem_m_ |
| 2.9%    | 8.150646  | 0.205423 | 2.5%      | 100.0 | integrate_.LOOPS |
| 2.4%    | 6.942384  | 0.078555 | 1.1%      | 391200.0 | qssa_i_.LOOPS |
| 2.3%    | 6.430820  | 0.481475 | 7.0%      | 21921875.0 | mixcp$thermchem_m_.LOOPS |
| 2.0%    | 5.585500  | 0.343099 | 5.8%      | 600.0 | computeheat$transport_m_.LOOPS |
| 1.8%    | 5.252285  | 0.062576 | 1.2%      | 391200.0 | rdwdot_i_.LOOPS |
| 1.7%    | 4.801062  | 0.723213 | 13.1%     | 31800.0 | derivative_x_calc_.LOOPS |
| 1.6%    | 4.461274  | 1.310813 | 22.7%     | 31800.0 | derivative_y_calc_.LOOPS |
| 1.5%    | 4.327627  | 1.290121 | 23.0%     | 31800.0 | derivative_z_calc_.LOOPS |
| 1.4%    | 3.963951  | 0.138844 | 3.4%      | 701.0 | get_mass_frac$variables_m_.LOOPS |
## Restructured S3D for multi-core systems

<table>
<thead>
<tr>
<th>Time Step</th>
<th>Runge K</th>
<th>S3D</th>
</tr>
</thead>
<tbody>
<tr>
<td>Time Step</td>
<td>Solve_Drive</td>
<td></td>
</tr>
<tr>
<td>Time Step</td>
<td>Integrate</td>
<td></td>
</tr>
<tr>
<td>Time Step</td>
<td>RHS</td>
<td></td>
</tr>
<tr>
<td>Time Step</td>
<td>grid loop -omp</td>
<td>get mass fraction</td>
</tr>
<tr>
<td>Time Step</td>
<td>grid loop-omp</td>
<td>get_velocity</td>
</tr>
<tr>
<td>Time Step</td>
<td>grid loop-omp</td>
<td>calc_inv_avg</td>
</tr>
<tr>
<td>Time Step</td>
<td>grid loop-omp</td>
<td>calc_temp</td>
</tr>
<tr>
<td>Time Step</td>
<td>grid loop-omp</td>
<td>Compute Grads</td>
</tr>
<tr>
<td>Time Step</td>
<td>grid loop-omp</td>
<td>Diffusive Flux</td>
</tr>
<tr>
<td>Time Step</td>
<td>grid loop-omp</td>
<td>Derivatives</td>
</tr>
<tr>
<td>Time Step</td>
<td>grid loop-omp</td>
<td>reaction rates</td>
</tr>
</tbody>
</table>
## Table 1: Profile by Function Group and Function

<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>85.3%</td>
<td>539.077983</td>
<td>--</td>
<td>--</td>
<td>144908.0</td>
<td>USER</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>21.7%</td>
<td>136.950871</td>
<td>0.583731</td>
<td>0.5%</td>
<td>600.0</td>
<td>rhsf_</td>
</tr>
<tr>
<td>14.7%</td>
<td>93.237279</td>
<td>0.132829</td>
<td>0.2%</td>
<td>600.0</td>
<td>rhsf_.LOOP@li.1084</td>
</tr>
<tr>
<td>8.7%</td>
<td>55.047054</td>
<td>0.309278</td>
<td>0.6%</td>
<td>600.0</td>
<td>rhsf_.LOOP@li.1098</td>
</tr>
<tr>
<td>6.3%</td>
<td>40.129463</td>
<td>0.265153</td>
<td>0.8%</td>
<td>600.0</td>
<td>rhsf_.LOOP@li.1211</td>
</tr>
<tr>
<td>5.8%</td>
<td>36.647080</td>
<td>0.237180</td>
<td>0.7%</td>
<td>600.0</td>
<td>rhsf_.LOOP@li.194</td>
</tr>
<tr>
<td>5.6%</td>
<td>35.264114</td>
<td>0.091537</td>
<td>0.3%</td>
<td>600.0</td>
<td>rhsf_.LOOP@li.320</td>
</tr>
<tr>
<td>3.7%</td>
<td>23.624271</td>
<td>0.054666</td>
<td>0.3%</td>
<td>600.0</td>
<td>rhsf_.LOOP@li.540</td>
</tr>
<tr>
<td>2.7%</td>
<td>17.211435</td>
<td>0.095793</td>
<td>0.6%</td>
<td>600.0</td>
<td>rhsf_.LOOP@li.1784</td>
</tr>
<tr>
<td>2.4%</td>
<td>15.471160</td>
<td>0.358690</td>
<td>2.6%</td>
<td>14400.0</td>
<td>derivative_y_calc_buff_r_.LOOP@li.1794</td>
</tr>
<tr>
<td>2.4%</td>
<td>15.113374</td>
<td>1.020242</td>
<td>7.2%</td>
<td>14400.0</td>
<td>derivative_z_calc_buff_r_.LOOP@li.1822</td>
</tr>
<tr>
<td>2.3%</td>
<td>14.335142</td>
<td>0.144579</td>
<td>1.1%</td>
<td>14400.0</td>
<td>derivative_x_calc_buff_r_.LOOP@li.1794</td>
</tr>
<tr>
<td>1.9%</td>
<td>11.794965</td>
<td>0.073742</td>
<td>0.7%</td>
<td>600.0</td>
<td>integrate_.LOOP@li.96</td>
</tr>
<tr>
<td>1.7%</td>
<td>10.747430</td>
<td>0.063508</td>
<td>0.7%</td>
<td>600.0</td>
<td>computespeciesdifflux2$transport_m_.LOOP</td>
</tr>
<tr>
<td>1.5%</td>
<td>9.733830</td>
<td>0.096476</td>
<td>1.1%</td>
<td>600.0</td>
<td>rhsf_.LOOP@li.247</td>
</tr>
<tr>
<td>1.2%</td>
<td>7.649953</td>
<td>0.043920</td>
<td>0.7%</td>
<td>600.0</td>
<td>rhsf_.LOOP@li.274</td>
</tr>
<tr>
<td>0.8%</td>
<td>5.116578</td>
<td>0.008031</td>
<td>0.2%</td>
<td>600.0</td>
<td>rhsf_.LOOP@li.398</td>
</tr>
<tr>
<td>0.6%</td>
<td>3.966540</td>
<td>0.089513</td>
<td>2.5%</td>
<td>1.0</td>
<td>s3d_</td>
</tr>
<tr>
<td>0.3%</td>
<td>2.027255</td>
<td>0.017375</td>
<td>1.0%</td>
<td>100.0</td>
<td>integrate_.LOOP@li.73</td>
</tr>
<tr>
<td>0.2%</td>
<td>1.318550</td>
<td>0.001374</td>
<td>0.1%</td>
<td>600.0</td>
<td>rhsf_.LOOP@li.376</td>
</tr>
<tr>
<td>0.2%</td>
<td>0.986124</td>
<td>0.017854</td>
<td>2.0%</td>
<td>600.0</td>
<td>rhsf_.REGION@li.1096</td>
</tr>
<tr>
<td>0.1%</td>
<td>0.700156</td>
<td>0.027669</td>
<td>4.3%</td>
<td>1.0</td>
<td>exit</td>
</tr>
</tbody>
</table>
Advantage of raising loops

- Create good granularity OpenMP Loop
- Improves cache re-use
- Reduces Memory usage significantly
- Creates a good potential kernel for an accelerator
Resultant Hybrid S3D Performance

Weak Scaling, Lower is Better

Number of cores (12 cores/node)

Wallclock Time (Seconds)

Hybrid S3D
All MPI S3D
Restructured S3D for multi-core systems

<table>
<thead>
<tr>
<th>Time Step – acc_data</th>
<th>Runge K</th>
<th>grid loop -ACC</th>
<th>get mass fraction</th>
</tr>
</thead>
<tbody>
<tr>
<td>Time Step – acc_data</td>
<td>Runge K</td>
<td>grid loop-ACC</td>
<td>get _velocity</td>
</tr>
<tr>
<td>Time Step – acc_data</td>
<td>Runge K</td>
<td>grid loop-ACC</td>
<td>calc_inv_avg</td>
</tr>
<tr>
<td>Time Step – acc_data</td>
<td>Runge K</td>
<td>grid loop-ACC</td>
<td>calc_temp</td>
</tr>
<tr>
<td>Time Step – acc_data</td>
<td>Runge K</td>
<td>grid loop-ACC</td>
<td>Compute Grads</td>
</tr>
<tr>
<td>Time Step – acc_data</td>
<td>Runge K</td>
<td>grid loop-ACC</td>
<td>Diffusive Flux</td>
</tr>
<tr>
<td>Time Step – acc_data</td>
<td>Runge K</td>
<td>grid loop-ACC</td>
<td>Derivatives</td>
</tr>
<tr>
<td>Time Step – acc_data</td>
<td>Runge K</td>
<td>grid loop-ACC</td>
<td>reaction rates</td>
</tr>
</tbody>
</table>
 !$acc data copyin(q,volum) shared(yspecies,u,avmolwt,mixMW,temp) !$acc parallel_loop private(i,ml,mu)
 do i = 1, nx*ny*nz, ms
  ml = i
  mu = min(i+ms-1, nx*ny*nz)
call get_mass_frac_r( q, volum, yspecies, ml, mu)
call get_velocity_vec_r( u, q, volum, ml, mu)
call calc_inv_avg_mol_wt_r( yspecies, avmolwt, mixMW, ml, mu)
voltmp(ml:mu,1,1)=q(ml:mu,1,1,5)*volum(ml:mu,1,1)
call calc_temp_r(temp, volttmp, u, yspecies, cpmix, avmolwt, ml, mu)
end do
 !$acc end parallel_loop
! Start communication - the _prep routines do posts and sends
! using buffer identified by itmp
itmp = 1
 !$acc acc_update host(u,temp,yspecies)
call computeVectorGradient_prep(u, itmp )
call computeScalarGradient_prep(temp, itmp )
do n=1,n_spec
  call computeScalarGradient_prep( yspecies(:,,:,:,:n), itmp )
enddo
! Compute remaining properties whilst communication is underway
 !$acc parallel_loop private(i,ml,mu)
doi = 1, nx*ny*nz, ms
  ml = i
  mu = min(i+ms-1, nx*ny*nz)
call calc_gamma_r( gamma, cpmix, avmolwt, ml, mu)
call calc_press_r( pressure, q(:,,:,:,:4), temp, avmolwt, ml, mu )
call calc_specEnth_allpts_r(temp, h_spec, ml, mu)
end do
 !$acc end parallel_loop
! Now wait for communication
  call derivative_xyz_wait( itmp )
  calc_buff_internal_wait = .false.
  itmp = 1
!$acc update device(u,temp,yspecies)
  call computeVectorGradient_calc( u, grad_u, itmp )
  call computeScalarGradient_calc( temp, grad_T, itmp )
!$acc parallel_loop private(n,itmp)
  do n=1,n_spec
    itmp = n + 4
    call computeScalarGradient5d_calc( yspecies(1,1,1,n), &
      grad_Ys(1,1,1,1,1), n_spec, n, itmp,sscale_1x,sscale_1y,sscale_1z  )
  enddo
!$acc end parallel_loop
!$acc end data
For the next year, until we can call subroutines and functions on the accelerator, the compiler must inline all subroutines and functions within a `acc_region`.

This is performed automatically by the compiler.

- Can be incrementally controlled by using compile line options
  - `-hwp` `-hpl=<path to program library>`
There are several things that inhibit the inlining of the call chain beneath the acc_region

- Call to subroutines and functions that the compiler does not see
- I/O, STOP, etc (Not anymore)
- Array shape changing through argument passing
- Dummy arguments
  - Real*8 dummy(*), dummy_2d(nx,*)
Successful Inlining

248.    !$acc parallel_loop private(i,ml,mu)
249.  1-------< do i = 1, nx*ny*nz, ms
250.  1     ml = i
251.  1     mu = min(i+ms-1, nx*ny*nz)
252.  1 I  call get_mass_frac_r( q, volum, yspecies, ml, mu)       ! get Ys from rho*Ys, volum from rho
253.  1 I  call get_velocity_vec_r( u, q, volum, ml, mu)           ! fill the velocity vector
254.  1 I  call calc_inv_avg_mol_wt_r( yspecies, avmolwt, mixMW, ml, mu)  ! set inverse of mixture MW
255.  1-------> end do
Inliner diagnostics

333. 1--------< do n=1,n_spec
334. 1       itmp = n + 4
335. 1       !call computeScalarGradient_calc( yspecies(:,;,:,n), grad_Ys(:,;,:,n,:), itmp )
336. 1       call computeScalarGradient5d_calc( yspecies(1,1,1,n), &

^ ftn-3007 ftn: ERROR RHSF, File = rhsf.f90, Line = 336, Column = 12
  Routine "write_date_and_time", referenced in "terminate_run", was not inlined because a scalar actual argument at position 1 is being mapped to an array dummy argument.

^ ftn-3007 ftn: ERROR RHSF, File = rhsf.f90, Line = 336, Column = 12
  Routine "write_date_and_time", referenced in "terminate_run", was not inlined because a scalar actual argument at position 1 is being mapped to an array dummy argument.

^ ftn-3007 ftn: ERROR RHSF, File = rhsf.f90, Line = 336, Column = 12
  Routine "write_date_and_time", referenced in "terminate_run", was not inlined because a scalar actual argument at position 2 is being mapped to an array dummy argument.

^ ftn-3007 ftn: ERROR RHSF, File = rhsf.f90, Line = 336, Column = 12
  Routine "write_date_and_time", referenced in "terminate_run", was not inlined because a scalar actual argument at position 1 is being mapped to an array dummy argument.
Inliner diagnostics (-rmp)

^  
ftn-3007 ftn: ERROR RHSF, File = rhsf.f90, Line = 336, Column = 12  
  Routine "write_date_and_time", referenced in "terminate_run", was not inlined because a scalar actual argument at position 1 is being mapped to an array dummy argument.

^  
ftn-3007 ftn: ERROR RHSF, File = rhsf.f90, Line = 336, Column = 12  
  Routine "write_date_and_time", referenced in "terminate_run", was not inlined because a scalar actual argument at position 2 is being mapped to an array dummy argument.

^  
ftn-3007 ftn: ERROR RHSF, File = rhsf.f90, Line = 336, Column = 12  
  Routine "write_date_and_time", referenced in "terminate_run", was not inlined because a scalar actual argument at position 1 is being mapped to an array dummy argument.

^  
ftn-3007 ftn: ERROR RHSF, File = rhsf.f90, Line = 336, Column = 12  
  Routine "write_date_and_time", referenced in "terminate_run", was not inlined because a scalar actual argument at position 1 is being mapped to an array dummy argument.
Inliner diagnostics ( -rmp )

^ fn-3021 fn: ERROR RHSF, File = rhsf.f90, Line = 336, Column = 12
Routine "mpi_finalize", referenced in "terminate_run", was not inlined because the compiler was unable to locate the routine to expand it inline.

^ fn-3021 fn: ERROR RHSF, File = rhsf.f90, Line = 336, Column = 12
Routine "mpi_barrier", referenced in "terminate_run", was not inlined because the compiler was unable to locate the routine to expand it inline.

^ fn-3021 fn: ERROR RHSF, File = rhsf.f90, Line = 336, Column = 12
Routine "mpi_wait", referenced in "derivative_y_calc_buff_r", was not inlined because the compiler was unable to locate the routine to expand it inline.

^ fn-3021 fn: ERROR RHSF, File = rhsf.f90, Line = 336, Column = 12
Routine "mpi_wait", referenced in "derivative_y_calc_buff_r", was not inlined because the compiler was unable to locate the routine to expand it inline.
Currently many compiler internal errors are given when forms are encountered that inhibit acceleration

- Calls within the acc_region
  - These can be identified by using the inliner

- Derived Types
  - These are being worked

- Dummy arguments

- Etc.
Early Software/Hardware Issues

- Finding lots of bugs in tools and compiler
  - Cannot fix them until they are identified
- Identified bottleneck in MPI messaging between GPUs
  - This is being addressed by Cray/Nvidia
    - Want zero transfer messages – GPU directly to other GPU
- Directives are emerging – changing
  - Usage is identifying new capabilities – pipelining
- Future GPUs will have a higher performance advantage over x86 sockets
The latest version of S3D with OpenMP and OpenACC performs:
- 1.2X faster on XK6 with GPU than a dual-CPU XE6
- 1.5X faster on XK6 with GPU than a XK6 without GPU
Comparisons between OpenMP and OpenACC

<table>
<thead>
<tr>
<th>Speedup</th>
<th>OpenMP</th>
<th>Acc</th>
<th>Acc</th>
<th>Host</th>
<th>Acc Copy</th>
<th>Acc Copy</th>
<th>Calls</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>Time</td>
<td>Time%</td>
<td>Time</td>
<td>Time</td>
<td>Time</td>
<td>In</td>
<td>Out</td>
<td></td>
<td>PE=HIDE</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Total (MBytes) (MBytes) Thread=HIDE</td>
</tr>
<tr>
<td>100.00%</td>
<td>187.783</td>
<td>234.011</td>
<td>148248</td>
<td>81843.75</td>
<td>1960504</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

| 2.094895 | 94.22    | 24.00% | 44.976 | 0.042 | --       | --       | 600 reaction_rate_vec_ACC_KERNEL@li.165 |
| 4.362416 | 65       | 7.90%  | 14.9  | 0.002 | --       | --       | 100 computecoefficients_r_ACC_KERNEL@li.141 |
| 1.781216 | 19.8     | 5.90%  | 11.16 | 0.01  | --       | --       | 600 rhsf_ACC_COPY@li.1806 |
| 2.889151 | 29.4     | 5.40%  | 10.176| 0.037 | --       | --       | 600 rhsf_ACC_KERNEL@li.379 |
| 2.527231 | 25.29    | 5.30%  | 10.007| 0.039 | --       | --       | 600 rhsf_ACC_KERNEL@li.1712 |
| 0.314012 | 3.05     | 5.20%  | 7.913 | 0.039 | --       | --       | 600 rhsf_ACC_KERNEL@li.1810 |
| 1.124026 | 7.93     | 3.80%  | 7.055 | 0.037 | --       | --       | 600 rhsf_ACC_KERNEL@li.1758 |
| 4.46391  | 25.48    | 3.00%  | 5.708 | 0.037 | --       | --       | 600 rhsf_ACC_KERNEL@li.419 |
|          |          | 3.00%  | 5.627 | 5.63  | --       | --       | 600 reaction_rate_vec_ACC_COPY@li.4958 |
|          |          | 2.90%  | 5.373 | 5.376 | --       | --       | 600 rhsf_ACC_COPY@li.2212 |
|          |          | 1.70%  | 3.131 | 0.008 | --       | 19490.63| 100 integrate_ACC_COPY@li.74 |
| 9.521674 | 25.48    | 1.40%  | 2.676 | 0.014 | --       | --       | 700 calc_primary_vars_ACC_KERNEL@li.42 |
|          |          | 1.40%  | 2.584 | 0.086 | 14175    | --       | 600 rhsf_ACC_COPY@li.366 |
|          |          | 1.20%  | 2.266 | 1.526 | --       | 6496.875 | 92400 derivative_z_pack_np_ACC_COPY@li.351 |
| 5.925433 | 13.35    | 1.20%  | 2.253 | 0.008 | --       | --       | 600 rhsf_ACC_KERNEL@li.989 |
|          |          | 1.10%  | 2.104 | 0.042 | --       | 12993.75| 1800 derivative_y_pack_np_ACC_COPY@li.429 |
|          |          | 1.10%  | 2.104 | 0.073 | --       | 12993.75| 1800 derivative_x_pack_np_ACC_COPY@li.433 |
|          |          | 1.10%  | 2.053 | 1.247 | --       | 6496.875 | 92400 derivative_z_pack_np_ACC_COPY@li.340 |
| 4.48176  | 7.74     | 0.90%  | 1.727 | 0.009 | --       | --       | 600 integrate_ACC_KERNEL@li.113 |
GPU Annotated Timeline

Integration loop – 6 times per call to compute transport coefficients

Transport Coefficients: 7%

Derivatives: Kernels only 2%, and should get near-perfect overlap w/ memcpys. But w/ current overheads and synchronous memcpys, currently totals 21%. (We will fix this.)

RHS: 24% ... but low GPU utilization in these kernels, so could be less. (We will improve this.)

Example of missed overlap opportunity, this one costing 6.5% of total time. (We will fix this.)

GPU is idle appx. 23% of total time, largely due to missing MPI overlap. (We will improve this.)
Future Developments

- Timeline shows where improvements can be obtained
  - Asynchronous updates – available this week
  - Overlapping MPI with GPU computation - needs
    - GPU direct
      - Available later in the year
    - Use `!$acc host_data use_device` directive to simply communication between device and host
      - Significantly cleans up code
    - Cuda proxy for running multiple MPI ranks on node and sharing the GPU
      - This would be used if the overlap and GPU direct succeeds on fully utilizing the GPU
!$acc host_data use_device

#ifdef GPU
!$acc data present(f)
!$acc host_data use_device(f)
#endif

if( deriv_z_list(idx)%packed ) then
  deriv_z_list(idx)%packed = .false.
if(deriv_z_list(idx)%neg_nbr>=0) then
  call MPI_ISend(f(1,1,1),(mx*my*iorder/2),
      MPI_REAL8,deriv_z_list(idx)%neg_nbr,deriv_list_size + idx, &
      gcomm,deriv_z_list(idx)%req(2),ierr)
endif
if(deriv_z_list(idx)%pos_nbr>=0) then
  ! send ghost cells to neighbor on (+z) side
  nm = mz + 1 - iorder/2
  call MPI_ISend(f(1,1,nm),(mx*my*iorder/2), &
      MPI_REAL8,deriv_z_list(idx)%pos_nbr,idx, &
      gcomm,deriv_z_list(idx)%req(4),ierr)
endif
else
  if(deriv_z_list(idx)%neg_nbr>=0) then
    call MPI_ISend(f(1,1,1),(mx*my*iorder/2), &
        MPI_REAL8,deriv_z_list(idx)%neg_nbr,deriv_list_size + idx, &
       gcomm,deriv_z_list(idx)%req(2),ierr)
  endif
  if(deriv_z_list(idx)%pos_nbr>=0) then
    ! send ghost cells to neighbor on (+z) side
    nm = mz + 1 - iorder/2
    call MPI_ISend(f(1,1,nm),(mx*my*iorder/2), &
        MPI_REAL8,deriv_z_list(idx)%pos_nbr,idx, &
        gcomm,deriv_z_list(idx)%req(4),ierr)
  endif
endif
#endif
!$acc end host_data
!$acc end data
#ifdef GPU
#endif
Projections to Kepler

• Key Kepler architectural features that will help us are:
  – 1.5x memory bandwidth, which will help key kernels
  – 4x as many addressable registers per thread, which cuts register spilling for top kernels significantly and reduces memory bandwidth bottleneck
    • This is a significant issue in getrates and compute_coefficients – the two most compute intensive kernels
  – Better overlap of concurrent independent tasks, enabling CUDA proxy feature, which could help us keep GPU busy a higher % of the time
<table>
<thead>
<tr>
<th>Memcpy Size (bytes) or Kernel Name</th>
<th>Time Taken (ms)</th>
<th>% Total</th>
<th>XX7 estimate</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Count</td>
<td>Min</td>
<td>Avg</td>
</tr>
<tr>
<td>memcpyH2D</td>
<td>2719</td>
<td>4</td>
<td>4,370,10</td>
</tr>
<tr>
<td>memcpyD2H</td>
<td>32560</td>
<td>25</td>
<td>263,57</td>
</tr>
<tr>
<td>kernel launches</td>
<td>12725</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>reaction_rate_vec_Sck_L142_1</td>
<td>60</td>
<td>64.21</td>
<td>64.86</td>
</tr>
<tr>
<td>computecoefficients_r_Sck_L141_1</td>
<td>10</td>
<td>148.7</td>
<td>149.1</td>
</tr>
<tr>
<td>rhsf_Sck_L378_50</td>
<td>60</td>
<td>16.86</td>
<td>16.93</td>
</tr>
<tr>
<td>rhsf_Sck_L1718_77</td>
<td>60</td>
<td>16.65</td>
<td>16.67</td>
</tr>
<tr>
<td>rhsf_Sck_L1813_79</td>
<td>60</td>
<td>15.81</td>
<td>15.88</td>
</tr>
<tr>
<td>rhsf_Sck_L1764_78</td>
<td>60</td>
<td>11.67</td>
<td>11.76</td>
</tr>
<tr>
<td>rhsf_Sck_L418_51</td>
<td>60</td>
<td>9.24</td>
<td>9.2</td>
</tr>
<tr>
<td>rhsf_Sck_L656_53</td>
<td>60</td>
<td>8.80</td>
<td>8.8</td>
</tr>
<tr>
<td>calc_primary_vars_Sck_L42_1</td>
<td>70</td>
<td>3.75</td>
<td>3.8</td>
</tr>
<tr>
<td>rhsf_Sck_L992_55</td>
<td>60</td>
<td>3.68</td>
<td>3.7</td>
</tr>
<tr>
<td>integrate_Sck_L113_2</td>
<td>60</td>
<td>2.82</td>
<td>2.8</td>
</tr>
<tr>
<td>rest.</td>
<td>12105</td>
<td>0.00</td>
<td>0.2</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CPU time</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>TOTAL</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

(memcpy should get near-perfect overlap by app improvement or by Kepler+proxy)
(memcpy should get near-perfect overlap by app improvement or by Kepler+proxy)
(assuming 1.5x improvement achieved if all kernels bandwidth limited). Should be a bit better than that due to reduced register spilling, but haven't fully modeled that yet.

whole-app speedup anticipated vs today by architectural improvements only:

2.0

(assuming better overlap due to Kepler + proxy)
Himeno Benchmark on Cray XK6: an OpenMP for Accelerators exercise

Roberto Ansaloni
Alistair Hart

Cray Performance Symposium, 25.July.11
Contents of talk

- A performance case study
  - The Himeno benchmark
  - Accelerating Himeno using OpenMP directives
    - assume you have met these already
    - Performance and scaling of the Himeno code
- How to accelerate a code using directives
  - A vademecum
- Suitability of codes and examples available
- Useful tools and tricks for accelerator directives
The Himeno Benchmark

- 3D Poisson equation
  - 19-point stencil
  - Highly memory intensive, memory bandwidth bound
- Fortran, C, MPI and OpenMP implementations available from [http://ompc.riken.jp/HPC_e/himenobmt_e.html](http://ompc.riken.jp/HPC_e/himenobmt_e.html)
- Several configurations available
  - Tests on XL configuration: 1024 x 512 x 512
- NVIDIA paper on GPU CUDA implementation
  - Phillips, E.H.; Fatica, M.; Implementing the Himeno benchmark with CUDA on GPU clusters IEEE International Symposium on Parallel & Distributed Processing (IPDPS), 2010 [PDF](http://example.com), or [ahart@cray.com](mailto:ahart@cray.com)
The Jacobi computational kernel

- The stencil is applied to pressure array \( p \)
- Updated pressure values are saved to temporary array \( \text{wrk2} \)
- Control value \( \text{wgosa} \) is computed
- In the benchmark this kernel is iterated a fixed number of times (nn)

```fortran
DO K=2,kmax-1
  DO J=2,jmax-1
    DO I=2,imax-1
      S0=a(I,J,K,1)*p(I+1,J, K )
        +a(I,J,K,2)*p(I, J+1,K ) &
        +a(I,J,K,3)*p(I, J, K+1) &
        +b(I,J,K,1)*(p(I+1,J+1,K )-p(I+1,J-1,K ) &
                  -p(I-1,J+1,K )+p(I-1,J-1,K )) &
        +b(I,J,K,2)*(p(I, J+1,K+1)-p(I, J-1,K+1) &
                  -p(I, J+1,K-1)+p(I, J-1,K-1)) &
        +b(I,J,K,3)*(p(I+1,J, K+1)-p(I-1,J, K+1) &
                  -p(I+1,J, K-1)+p(I-1,J, K-1)) &
        +c(I,J,K,1)*p(I-1,J, K ) &
        +c(I,J,K,2)*p(I, J-1,K ) &
        +c(I,J,K,3)*p(I, J, K-1) &
        + wrk1(I,J,K)
      SS=(S0*a(I,J,K,4)-p(I,J,K))*bnd(I,J,K)
      WGOSA=WGOSA+SS*SS
      wrk2(I,J,K)=p(I,J,K)+OMEGA *SS
    ENDDO
  ENDDO
ENDDO
ENDDO
```
Distributed implementation

- The outer loop is performed a fixed number of times.
- The Jacobi kernel is executed and new pressure array \( \text{wrk2} \) and control value \( \text{wgosa} \) are computed.
- The array is updated with the new pressure values.
- The halo region values are exchanged between neighbor PEs.
- Send and receive buffers are used.
- The maximum control value is computed with an Allreduce operation across all the PEs.

```
DO loop = 1, nn
  compute Jacobi kernel → \( \text{wrk2}, \text{wgosa} \)
  copy back \( \text{wrk2} \) into \( p \)
  pack halo from \( p \) into send buffers
  exchange halos with neighbour PEs
  unpack halo into \( p \) from recv buffers
  Allreduce to sum \( \text{wgosa} \) across PEs
ENDDO
```
Porting Himeno to the Cray XK6

- Several versions tested, with communication implemented in MPI or Fortran coarrays
- GPU version using OpenMP Accelerator directives
- Comparing Cray XK6 timings with best Cray XE6 results (hybrid MPI/OpenMP)
- Arrays reside permanently on the GPU memory
- Data transfers between host and GPU are:
  - Communication buffers for the halo exchange
  - Control value
Allocating arrays on the GPU

- Arrays are allocated on the GPU memory in the main program with the `data` directive.
- In the subroutines the `data` directive is replicated with the `present` clause, to use the data already present in the GPU memory and avoid extra allocations.
- Since present clause is used, no `acc_copy` clauses are used, and data transfers to/from host are implemented by `acc_update` directives.

```fortran
PROGRAM himenobmtxp
...
$acc data shared &
$acc (p,a,b,c,wrk1,wrk2,bnd, &
$acc sendbuffx_up,sendbuffx_dn, &
$acc sendbuffy_up,sendbuffy_dn, &
$acc sendbuffz_up,sendbuffz_dn)
...
$acc end data

SUBROUTINE jacobi(nn,gosa)
$acc data present &
$acc (p,a,b,c,wrk1,wrk2,bnd, &
$acc sendbuffx_up,sendbuffx_dn, &
$acc sendbuffy_up,sendbuffy_dn, &
$acc sendbuffz_up,sendbuffz_dn)
```
The GPU kernel for the main loop is created with the `acc_region_loop` directive.

The scoping of the main variables is specified earlier with the `acc_data` directive - no need to replicated it here.

 wgosa is computed by specifying the `reduction` clause, as in a standard OpenMP parallel loop.

`num_pes` clause is used to indicate the number of threads within a threadblock (compiler default 128).

```
DO loop=1,nn
    gosa = 0
    wgosa = 0

 !$acc parallel_loop &
 !$acc& private(s0,ss) &
 !$acc& reduction(+:wgosa) &
 !$acc& num_pes(2:256) &

    DO K=2,kmax-1
        DO J=2,jmax-1
            DO I=2,imax-1
                S0=a(I,J,K,1)*p(I+1,J, K ) &
                ...
                wgosa = wgosa + SS*SS
            ENDDO
        ENDDO
    ENDDO
ENDDO
ENDDO
```
Halo region buffers

- Halo values are extracted from the \texttt{wrk2} array and packed into the send buffers, on the GPU.
- A global \textit{parallel} is specified and buffers in the X, Y, and Z directions are packed within \textit{loop} blocks.
- The send buffers are copied to host memory with \textit{update}.
- In the same way, after the halo exchange, the recv buffers are transferred to the GPU memory and used to update the array \texttt{p}.
- N.B. Currently it’s not possible to include array sections in \texttt{acc\_update} – buffers are necessary.
Coarray implementation

- Coarrays are used to perform the halo exchange
- Non-blocking communication needs *pgas defer_sync* directive
- Programmer now responsible for data synchronization
- By deferring sync point, network comms can be overlapped with CPU or GPU activity
- Updating p from *wrk2* (on GPU) overlapped with halo exchange
- N.B. no *sync all*: CAF intrinsic *COSUM* has loose synchronisation (so do need *sync memory* first).

```plaintext
!dir$ pgas defer_sync
recvbuffz_up(:, :) [myx, myy, myz-1] = &
    sendbuffz_dn(:, :)
...
!$acc parallel_loop
DO k = 2, kmax-1
    DO j = 2, jmax-1
        DO i = 2, imax-1
            p(i, j, k) = wrk2(i, j, k)
        ENDDO
    ENDDO
ENDDO
ENDDO

!$acc end parallel_loop
sync memory

gosa = COSUM(wgosa)
!$acc update &
!$acc&
device(recvbuffz_dn, recvbuffz_up)
```
Coarray implementation

- Coarrays are used to perform the halo exchange.
- Non-blocking communication needs *pgas* `defer_sync` directive.
- Programmer now responsible for data synchronization.
- By deferring sync point, network comms can be overlapped with CPU or GPU activity.
- Updating `p` from `wrk2` (on GPU) overlapped with halo exchange.
- N.B. no *sync all*: CAF intrinsic *COSUM* has loose synchronisation (so do need *sync memory* first).

Compiler does not currently support using coarrays in an accelerator region, so this does not work!

You need to make a local copy of the coarray buffers to non-coarray buffers and then transfer them to GPU memory.

This affects the performance, by increasing the host CPU time.

Compiler directive:
```
%
\%dir$ pgas defer_sync recvbuffz_up(:, :) myx.myy.myz-1 =
\%sendbuffz_dn(:, :) 
%
sequently
\%omp omp_region_loop
DO k = 2, kmax - 1
\%omp end omp_region_loop
sync memory
\%omp sync memory
wgosa = COSUM(wgosa)
\%acc acc_update &
\%acc acc(recvbuffz_dn, recvbuffz_up)
```
Total number of lines in the original Himeno MPI-Fortran code: 629
Total number lines in the modified version with coarrays and accelerator directives: 554
  - don't need MPI_CART_CREATE and the like
Total number of accelerator directives: 27
  - plus 18 "end" directives
Benchmarking the code

• Cray XK6 configuration (vista)
  • Single AMD MC12 2.1GHz CPU cores, 12 cores per node
  • Nvidia Tesla X2090 GPU, 1 per node
  • Running with 1 PE (GPU) per node
  • Himeno case XL needs at least 8 Cray XK6 nodes

• Cray XE6 configuration (kaibab)
  • Dual AMD MC12 2.1 GHz nodes, 24 cores per node
  • Running on fully packed nodes: all cores used
  • Depending on the number of nodes, 1-6 OpenMP threads per PE are used

• All comparisons are for strong scaling
  • fixed total problem size
  • Nvidia CUDA example is weak scaling
Himeno Performance

- Node-for-node, Cray XK6 (GPU) outperforms Cray XE6 (CPU)
- CAF/ACC is the faster than MPI/ACC on high number of nodes
- ACC code has slightly worse scalability than MPI/acc
  - more on this later
Cray XK6 ratio to Cray XE6

- Cray XK6 is always faster
  - Ratio drops on 16 nodes
  - On 16 nodes the CPU code gets a superlinear boost due to cache effect
- On 128 nodes GPU code is about 20% faster than CPU code
CAF/ACC code breakdown (craypat!)

- Host/GPU transfers always take more time than the halo exchange (network)
  - this code would benefit from an efficient direct GPU-GPU communication
- On 128 nodes less than 50% of the time is spent in the GPU compute kernel
- Extra copy of coarray buffers increases the CPU time (potentially avoidable)
  - This is why CAF code is slower at low node count

![Himeno benchmark](image)

- XL case (1024x512x512)
- GPU compute
- GPU -> Host
- Halo exchange
- Host -> GPU
- CPU
Conclusions from the Himeno case study

- It has been very simple to implement the GPU code with OpenMP accelerator directives
- Work has evolved with updates in the (pre-release) compiler
  - Always got the right answers
  - Occasionally needed workarounds before features implemented
  - Compiler team extremely responsive
- Future releases will provide more control of the GPU and allow for better performance
- Codes where data can permanently reside in GPU memory will benefit from an efficient direct GPU-GPU communication
  - N.B. GPUs not on same PCIe bus
  - Many hardware questions need addressing to do this
Future work for Himeno

- Increased overlap of communication and computation
  - `async` clause for accelerator kernels, data transfers will help this
  - is there enough work in himeno to really hide the comms?
    - we tried precomputing halo regions of temporary array `wrk2` for earlier halo exchange
    - allows better overlap with GPU computation (interior of `wrk2`, copy of `wrk2` into `p`)
    - so far this has not improved code performance
  - measuring overlap is not easy
- Better tuning of GPU kernels
- A distributed CUDA implementation should be implemented to verify the efficiency of the OpenMP for Accelerator directives
Thank you. Questions?