

Future Architectural Directions

- Nodes are becoming much more powerful
 - More processors/node
 - More threads/processor
 - Vector lengths are getting longer
 - Memory hierarchy is becoming more complex
 - Scalar performance is not increasing

Threading on the Node and Vectorization is becoming more important

Today's Multi-Petascale Systems – Node Architecture

	Cores on the node	Total threading	Vector Length	Programming Model
Blue Waters	(16) 32	32	8 (4)	OpenMP/MPI/Vector
Blue Gene Q	16	32	8	OpenMP/MPI/Vector
Magna-Cours	(12) 24	(12) 24	4	OpenMP/MPI/Vector
Titan (ORNL)	16 (16)	16 (768*)	(8) (4) (32)	Threads/Cuda/Vector
Intel MIC	>50	>200	16	OpenMP/MPI/Vector
Power 7 (??)	16	32	8	OpenMP/MPI/Vector

* Nvidia allows oversubscription to SIMT units

Vectorization is becoming more important

- ALL accelerated nodes require vectorization at a good size to achieve reasonable performance
 - Nvidia Kepler 32 length
 - Intel MIC >8
- All compilers other than Cray's CCE were designed for marginal vector performance, they do not understand current tradeoffs
 - Be sure to get listing indicating if loop vectorizes
- User refactoring of loop is paramount in gaining good performance on future systems

Memory Hierarchy is becoming more complex

- As processors get faster, memory bandwidth cannot keep up
 - More complex caches
 - Non Uniform Memory Architecture (NUMA) for shared memory on node
 - Operand alignment is becoming more important
- Going forward this will become even more complex – two memories within same address space
 - Fast expensive memory
 - Slow less expensive memory
 - More about this later

Scalar performance is not getting better

- Consider Intel's chips
 - Xeon line with more cores per node using traditional X86 instruction set
 - MIC line with many more cores of slower processors
- Hosted system – Xeon with MIC
 - Native mode – run complete app on the MIC
 - Scalar performance will be an issue
 - Non-vector code will be an issue
 - Off Load mode – use Xeon as host, major computation on MIC
 - Memory transfer to and from Host will be an issue

Scalar Performance is not getting better

- Consider Nvidia approach
- Looking at ARM chip as co-processor
 - Once again scalar is far below state of the art Xeon
- So why not build an Exascale system out of Xeons or Power 7
 - **TOO MUCH POWER**

Code Design Question?

- Should code designers be concerned with memory management like that required to utilize a hosted accelerator like XK7

YES

- This is not throw away work?

WHY??

- All systems will soon have a secondary memory that is as large as we require; however, it will not have high bandwidth to the principal compute engine.
- There will be a smaller faster memory that will supply the principal compute engine.
- While system software may manage the two memories for the user, the user will have to manage these desperate memories to achieve maximum performance

So What is Heterogeneous Computing

- I believe it will have more to do with different memories
 - If doing scalar processing, application can afford to access slower larger memory
 - Scalar processing may be significantly slower than state-of-the-art Xeon
 - If doing high speed compute, application must have major computational arrays in fast memory
 - Parallel vector processing need high memory bandwidth and larger caches/registers

- **What to avoid**

- Excessive memory movement
 - Memory organization is the most important analysis for moving an application to these systems
- Avoid wide gaps between operands
 - Indirect addressing is okay, if it is localized
- Avoid scalar code
 - Think about Cyber 205, Connection Machine

- **What to do – Good Threading (OpenMP)**
 - Must do high level threading
 - Thread must access close shared memory rather than distant shared memory
 - Load Balancing
- **What to do – Good Vectorization**
 - Vectorization advantage allows for introducing overhead to vectorize
 - Vectorization of Ifs
 - Conditional vector merge (too many paths??)
 - Gather/scatter (Too much data motion??)
 - Identification of strings

- **Given the success of OpenMP extensions for accelerators, OpenACC and Intel's OffLoad Directives OpenMP offers an approach to develop a performance portable application that targets ALL future architecture**

Programming for Future Multi-Petaflop and Exaflop Computers

aka

Finding more parallelism in existing applications

Porting an existing application to new Systems

The Cray logo is located in the top right corner of the slide. It consists of the word "CRAY" in a blue, sans-serif font. To the right of the text is a decorative graphic of a grid of small circles, with some circles highlighted in orange and yellow, suggesting a network or data flow.

- **Converting to Hybrid OpenMP/MPI**
 - Identifying high level OpenMP loops
 - Using the Cray Scoping tool
 - Using the program library (-hwp)
 - NUMA effects on the XK6 node
 - Comparing Hybrid OpenMP/MPI to all MPI
 - Using the progress engine for overlapping MPI and computation
- **Looking at methods of acceleration**
 - Using Cuda with OpenACC and Cuda Fortran, Visual profiler, command line profiler, libsci being used with OpenACC
- **A systematic approach for converting a Hybrid OpenMP/MPI application to OpenACC**
 - Using OpenACC
 - First, let the compiler do most of the work
 - Using Craypat to identify the most time consuming portions of the accelerated code
 - Optimizing the OpenACC code
 - Most optimizations will improve OpenMP code
 - Employing Cuda and/or Cuda Fortran in an OpenACC application

- 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 application 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

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.

CAUTION!!

- 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 compiler 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

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.
- Identify major computational arrays
- **Tools that will be needed**
 - **Advanced instrumentation to measure**
 - DO loop statistics, iteration counts, inclusive time
 - Routine level sampling and profiling

Normal Profile – default Craypat report

Table 1: Profile by Function Group and Function

Time%	Time	Imb. Time	Imb. Time%	Calls	Group Function PE=HIDE
100.0%	50.553984	--	--	6922023.0	Total

52.1%	26.353695	--	--	6915004.0	USER

16.9%	8.540852	0.366647	4.1%	2592000.0	parabola_
8.0%	4.034867	0.222303	5.2%	288000.0	remap_
7.1%	3.612980	0.862830	19.3%	288000.0	riemann_
3.7%	1.859449	0.094075	4.8%	288000.0	ppmlr_
3.3%	1.666590	0.064095	3.7%	288000.0	evolve_
2.6%	1.315145	0.119832	8.4%	576000.0	paraset_
1.8%	0.923711	0.048359	5.0%	864000.0	volume_
1.8%	0.890751	0.064695	6.8%	288000.0	states_
1.4%	0.719636	0.079651	10.0%	288000.0	flatten_
1.0%	0.513454	0.019075	3.6%	864000.0	forces_
1.0%	0.508696	0.023855	4.5%	500.0	sweepz_
1.0%	0.504152	0.027139	5.1%	1000.0	sweepy_
=====					
37.9%	19.149499	--	--	3512.0	MPI

28.7%	14.487564	0.572138	3.8%	3000.0	mpi_alltoall
8.7%	4.391205	2.885755	39.7%	2.0	mpi_comm_split
=====					
10.0%	5.050780	--	--	3502.0	MPI_SYNC

6.9%	3.483206	1.813952	52.1%	3000.0	mpi_alltoall_(sync)
3.1%	1.567285	0.606728	38.7%	501.0	mpi_allreduce_(sync)
=====					

Normal Profile – Using “setenv PAT_RT_HWPC 1”

```

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

```

Re-compiling with `-hprofile_generate "pat_report-O callers"`

```

100.0% | 117.646170 | 13549032.0 |Total
|-----
| 75.4% | 88.723495 | 13542013.0 |USER
|-----
|| 10.7% | 12.589734 | 2592000.0 |parabola_
|-----
||| 7.1% | 8.360290 | 1728000.0 |remap_.LOOPS
4|| | | | | remap_
5|| | | | | ppmlr_
|-----
||||| 3.2% | 3.708452 | 768000.0 |sweepx2_.LOOP.2.li.35
7||||| | | | | sweepx2_.LOOP.1.li.34
8||||| | | | | sweepx2_.LOOPS
9||||| | | | | sweepx2_
10||||| | | | | vhone_
6||||| 3.1% | 3.663423 | 768000.0 |sweepx1_.LOOP.2.li.35
7||||| | | | | sweepx1_.LOOP.1.li.34
8||||| | | | | sweepx1_.LOOPS
9||||| | | | | sweepx1_
10||||| | | | | vhone_
|-----
||||| 3.6% | 4.229443 | 864000.0 |ppmlr_
|-----
4||| 1.6% | 1.880874 | 384000.0 |sweepx2_.LOOP.2.li.35
5||| | | | | sweepx2_.LOOP.1.li.34
6||| | | | | sweepx2_.LOOPS
7||| | | | | sweepx2_
8||| | | | | vhone_
4||| 1.6% | 1.852820 | 384000.0 |sweepx1_.LOOP.2.li.35
5||| | | | | sweepx1_.LOOP.1.li.34
6||| | | | | sweepx1_.LOOPS
7||| | | | | sweepx1_
8||| | | | | vhone_
|-----

```