BLUE WATERS

SUSTAINED PETASCALE COMPUTING

Running Applications on Blue Waters

Jing Li, Omar Padron, Gengbin Zheng

GREAT LAKES CONSORTIUM FOR PETASCALE COMPUTATION

CRAY

Modules

- The user environment is controlled using the modules environment management system.
- The module utility helps you quickly identify software that is available on the system and makes it easier to modify your environment.
- List all available modules and versions:
  - module avail
- List all modules currently loaded
  - module list

Modules (cont.)

- Modules may be loaded, unloaded, or swapped either on a command line or in $HOME/.bashrc (.cshrc for csh) shell startup file. E.g.
  - module load PrgEnv-gnu
  - module unload PrgEnv-gnu
  - module swap PrgEnv-gnu PrgEnv-cray
  - Module load ddt
  - Module load fftw

Blue Waters Programming Environments

Three programming environments available, managed by the module utility:

- Cray Programming Environment, the default
- PGI programming environment
- Gnu programming environment

Blue Waters Programming Environments

- Programming Environments managed through the module utility.
- Modules help ensure that your environment is always configured properly. Paths, libraries, etc, will be properly set by the chosen programing environment using module.
- Compiler wrappers ftn, cc, CC, etc, enable the use of desired compilers, and their corresponding include files, library paths etc.

Blue Waters Programming Environments

- ``module list” shows all currently loaded software modules, including the programing environment, which is defaulted to PrgEnv-cray
- ``module avail” displays all the available software modules
- ``module swap” or ``module unload/load” both can change the programming environments. For example:

  module swap PrgEnv-cray PrgEnv-pig

  swiches from Cray to Pgi.

Programming Models

- MPI
- OpenMP
- Hybrid Programming: MPI + OpenMP
- Partitioned Global Address Space (PGAS) paradigm
  - CAF
  - UPC
- Charm++

MPI

- Compiling and linking is performed using wrapper scripts ftn, cc, and CC for source code written in Fortran, C, and C++, respectively.
  - Wrappers invoke the appropriate compiler based on the current Programming Environment
  - Wrappers automatically link in a wide variety of libraries as necessary, including MPI (for instance, -lmpi is not required and will cause the link step to fail).

OpenMP

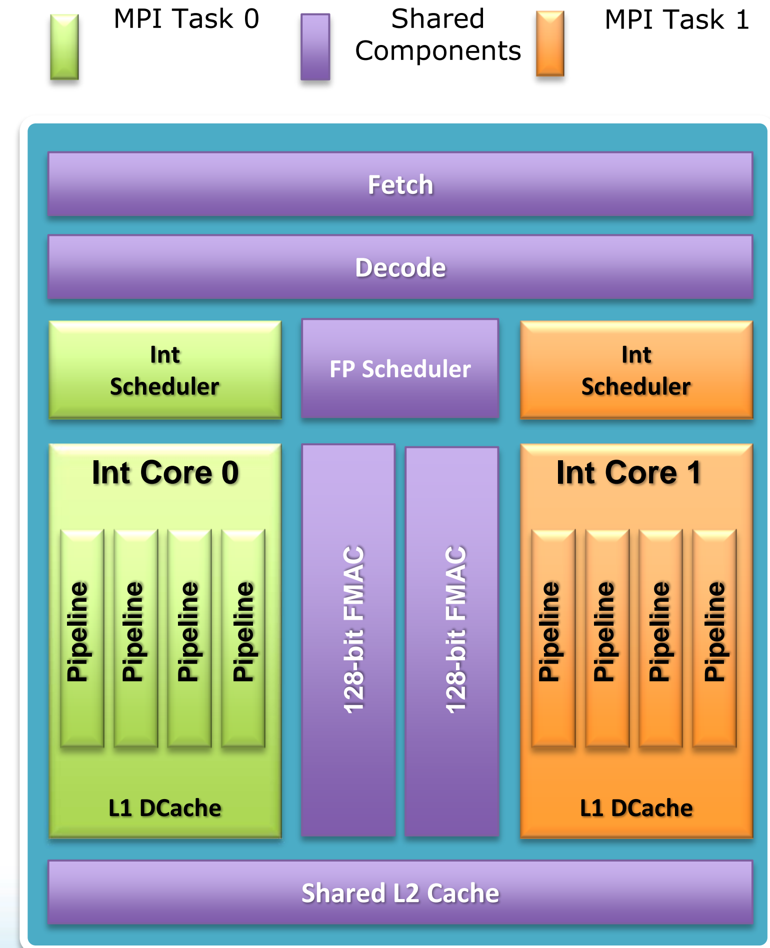
- a shared memory programming paradigm on the node
  - Cray compilers:
    - Default enabled: -h thread2
  - GNU compilers:
    - -fopenmp
  - PGI compilers:
    - -mp

MPI+OpenMP

- MPI+OpenMP is an efficient way to exploit multicore processors on Blue Waters.
  - Each OpenMP thread typically runs on one compute core (i.e. maximum 32 on BW).
- Thread safety
  - Required to specify the desired level of thread support
  - set environment variable MPICH_MAX_THREAD_SAFETY to different values to increase the thread safety.
    - MPI_THREAD_SINGLE (default)
    - MPI_THREAD_FUNNELED
    - MPI_THREAD_SERIALIZED
    - MPI_THREAD_MULTIPLE

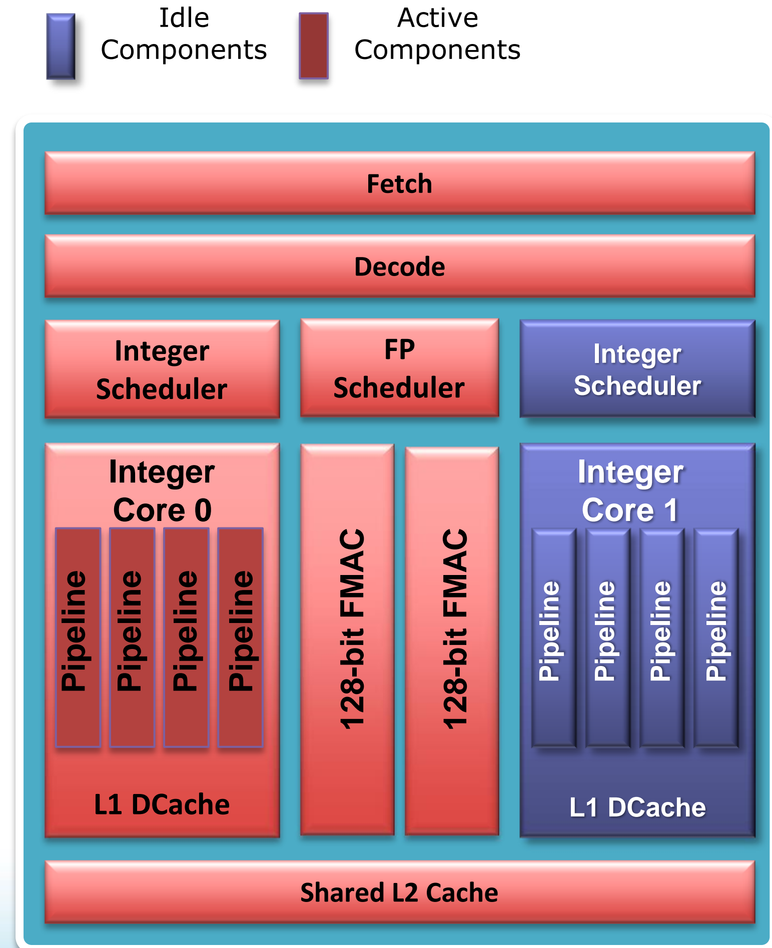
Two MPI Tasks on a Compute Unit ("Dual-Stream Mode")

- 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



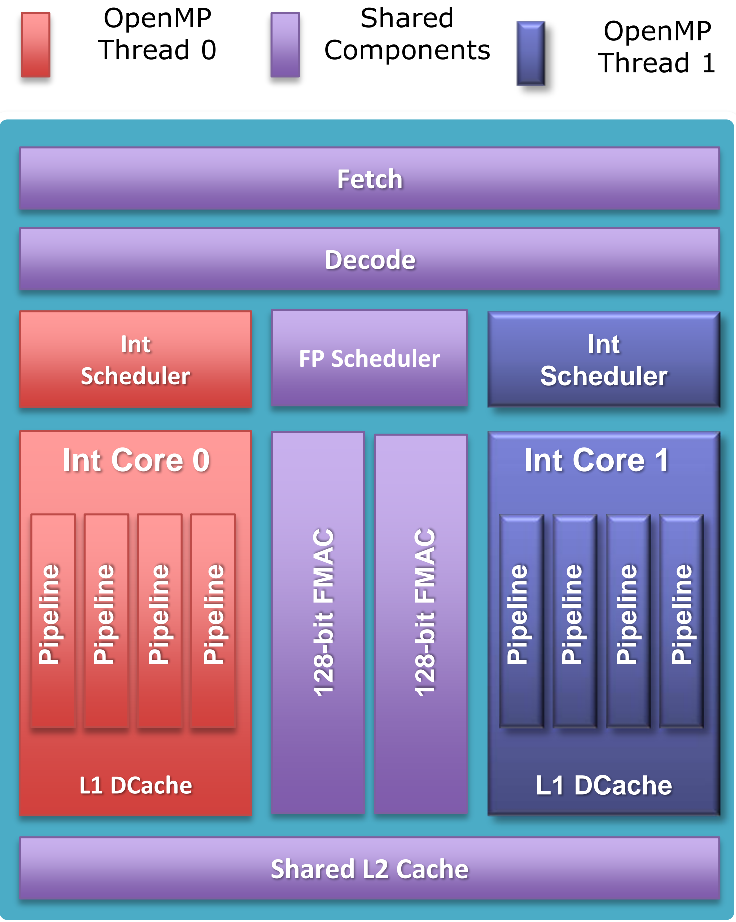
One MPI Task on a Compute Unit ("Single Stream Mode")

- 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



Running in Dual or Single-Stream modes

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

- Single-Stream mode is specified through the -j aprun option. Specifying -j 1 tells aprun to place 1 process or thread on each compute unit.

- When OpenMP threads are used, the -d option must be used to specify how many threads will be spawned per MPI process. See the aprun(1) man page for more details. The aprun –N option may be used to specify the number of MPI processes to assign per compute node or -S to specify the number of processes per Interlagos die. Also, the environment variable $OMP_NUM_THREADS needs to be set to the correct number of threads per process.

- For example, the following spawns 4 MPI processes, each with 8 threads, using 1 thread per compute unit.
  - OMP_NUM_THREADS=8
  - aprun -n 4 -d 8 -j 1 ./a.out

NUMA Considerations

- Each Interlagos processor has 2 NUMA memory domains, each with 4 Bulldozer Modules. Access to memory located in a remote NUMA domain is slower than access to local memory.
- 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.
- When running 1 process with threads over the NUMA domains, it's critical to initialize (not just allocate) memory from the thread that will use it in order to avoid NUMA side effects.

PGAS

- PGAS languages (UPC & Coarray Fortran) **fully optimized** and **integrated into the compiler**
  - UPC 1.2 and Fortran 2008 coarray support
  - No preprocessor involved
  - Target the network appropriately
  - Full debugger support with Allinea's DDT

Coarray Fortran (CAF)

- Coarray Fortran is a small set of extensions to Fortran for Single Program Multiple Data (SPMD) parallel programming
  - included in the current standard (Fortran 2008).
- Cray Fortran: -h caf (on by default)
- Gfortran:
  - -fcoarray=<keyword>

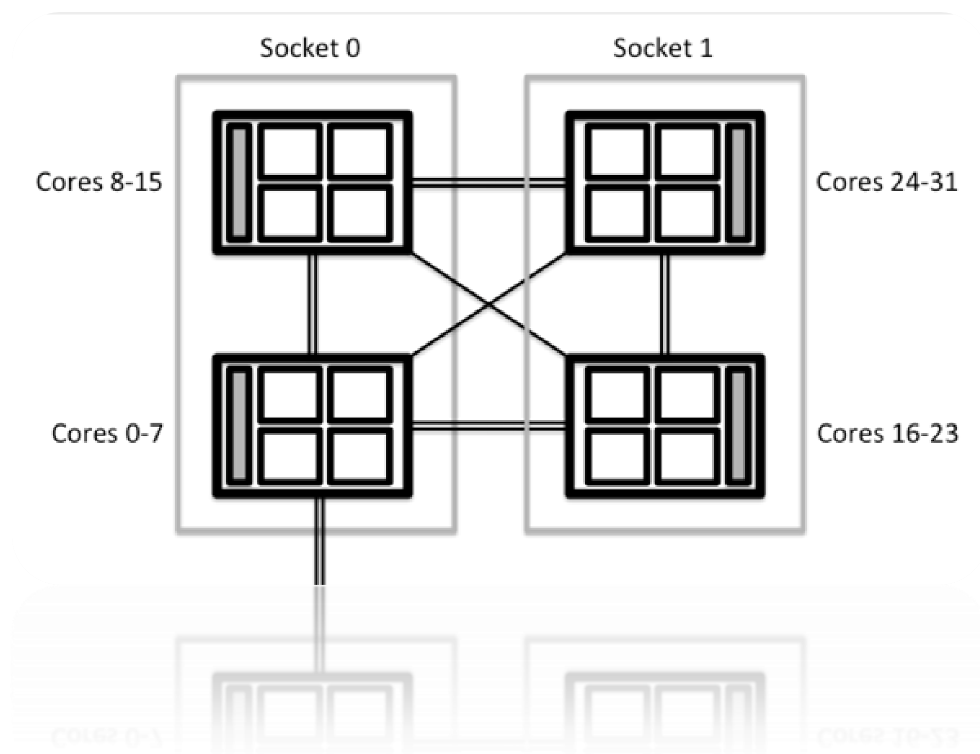
UPC

- An extension of C that supports a single shared, partitioned global address space
- UPC is fully integrated into the Cray C compiler, to enable:
  - -h upc

Charm++

- Charm++ provides processor virtualization
  - Object oriented C++ programming
  - Migratable object-based dynamic load balancing
  - Fault tolerance and many other features
- To build Charm++ on BW
  - ./build charm+++ gni-crayxe

Setting Process Affinity – BW XE node



- 32 integer cores
- 16 FPU's
- 4 numa nodes
- 2 sockets

Setting Process Affinity – aprun options

Common aprun options are:

- -n: Number of processing elements PEs for the application
- -N: Number of PEs to place per node
- -S: Number of PEs to place per NUMA node.
- -d: Number of CPU cores required for each PE and its threads
- -cc: Binds PEs to CPU cores.
- -r: Number of CPU cores to be used for core specialization
- -j: Dual or single stream/integer cores to use for a PE
- -ss: Enables strict memory containment per NUMA node

Setting Process Affinity – pure MPI code

Assume XE nodes are used in the following and proper resources are allocated, then:

- ``aprun -n 64" places 32 mpi processes on a XE node by default;
- ``aprun -n 64 -N 8" places 8 mpi processes on a XE node.
- ``aprun -n 64 -N 8 -S 2" places 8 mpi processes on a XE node using 4 numa nodes with 2 mpi processes per numa;

Setting Process Affinity – pure MPI code

Assume XE nodes are used in the following and again proper resources are allocated. To precisely control the placements, use -cc:

- ``aprun -n 64 -N 8 -cc 0,1,8,9,16,17,24,25" specifies actually where each of 8 mpi processes on a node will be placed.
- ``aprun -n 64 -N 8 -cc 0,4,8,12,16,20,24,28" specifies a different placement compare to the above;

Setting Process Affinity – MPI+openMP

Assume XE nodes are used in the following, then:

- ``aprun -n 64 -d 2" places 16 mpi processes on a XE node, 4 per numa along with their corresponding threads;
- ``aprun -n 64 -N 8 -d 2" places 8 mpi processes on a XE node using 2 numa nodes with 4 mpi processes per numa together with their corresponding threads;
- ``aprun -n 64 -N 8 -S 2 -d 2" places 8 mpi processes on a XE node using 4 numa nodes with 2 mpi processes per numa together with their threads;

Setting Process Affinity – MPI+openMP

Assume XE nodes are used in the following, the -cc option provides precise control on placements:

- ``aprun -n 64 -cc 0,1:2,3:8,9:10,11:16,17:18,19:24,25:26,27 puts 8 mpi processes on an XE node, with core 0,1 for 1st MPI process and its 2 threads; core 2,3 for 2nd MPI process and its 2 threads …
- ``aprun -n 64 -cc 0,1:4,5:8,9:12,13:16,17:20,21:24,25:28,29 puts 8 mpi processes on an XE node, with core 0,1 for 1st MPI process and its 2 threads; core 4,5 for 2nd MPI process and its 2 threads …

Running Applications

- Job submission
  - Prepare a bash script to run

```
#! /bin/bash

echo "Running a job on MOM node $(hostname)"
```

  - Submit the script with **qsub**

```
$ qsub -l nodes=4 -l walltime=1:00:00 -N example_job /path/to/script.pbs
```

  - Reservation options can also be put in the script as annotated comments

```
#! /bin/bash

#PBS -l nodes=4
#PBS -l walltime=0:01:00
#PBS -N example_job

echo "Running a job on MOM node $(hostname)"
```

Running Applications

- Interactive Jobs
  - User is placed on a MOM node with a shell prompt
  - No job script is necessary

```
$ qsub -I -X               \
       -l nodes=2:ppn=32   \
       -l walltime=0:30:00 \
       -N interactive_job
```

Running Applications

- MOM node
  - Manager node that runs the job script (does *not* participate in MPI applications)
  - Initiates parallel application launch using **`aprun`**
- aprun
  - Used in your job script to run your application binary
  - Used *instead* of **`mpirun`**
  - Handles placement of processes

```
$ aprun [placement_options] /path/to/binary
```

```
#! /bin/bash
#PBS -l nodes=4:ppn=32:xe
#PBS -l walltime=00:01:00
#PBS -N example_job

echo "Running a job on MOM node $(hostname)."
echo "using the following compute nodes:"
aprun -n 4 -N 1 "$(which hostname)"
```

Running Applications

- Common qsub reservation options
  - `-l nodes=N:ppn=P:xe`
  - `-l walltime=HH:MM:SS`
  - `-N JOB_NAME`
  - `-e STDERR_FILE`
  - `-o STDOUT_FILE`
  - `-j oe`
- Common aprun placement options
  - `-n TOTAL_PE`
  - `-N PE_PER_NODE`
  - `-S PE_PER_NUMA_DOMAIN`
  - `-d THREADS_PER_PE`
  - `-r NUM_SPECIAL_CORES`
  - `-cc CPU_BINDING_LISTS`

Running Applications

- Example 1 – OpenMP on a single XE node

```
#! /bin/bash
#PBS -l nodes=1:ppn=32:xe
#PBS -l walltime=1:00:00
#PBS -N big_flops
#PBS -j oe

cd “$PBS_O_WORKDIR”
export OMP_NUM_THREADS=32
aprun -n 1 -d “$OMP_NUM_THREADS” ./big_flops
```

Running Applications

- Example 2 – FLOP heavy MPI code

```
#! /bin/bash
#PBS -l nodes=16:ppn=32:xe
#PBS -l walltime=3:00:00
#PBS -N bigger_flops
#PBS -j oe

cd “$PBS_O_WORKDIR”
aprun -n “$((16*16))” -N 16 -d 2 ./bigger_flops
```

Running Applications

- Example 3 – FLOP heavy MPI/OpenMP code

```
#! /bin/bash
#PBS -l nodes=64:ppn=32:xe
#PBS -l walltime=1:00:00
#PBS -N even_bigger_flops
#PBS -j oe

cd "$PBS_O_WORKDIR"
export OMP_NUM_THREADS=16
CPU_LIST="0,2,4,6,8,10,12,14,16,18,20,22,24,26,28,30"
CPU_LIST="$(seq -s , 0 2 31)" # SHORTCUT
aprun -n 64 -N 1                  \
      -d "$OMP_NUM_THREADS"       \
      -cc "$CPU_LIST"
```

Running Applications

- SPMD Mode
  - aprun feature for running different binaries with different options under a unified communicator
  - Option, binary sets are separated with a single colon

```
#! /bin/bash
#PBS -l nodes=8:ppn=32:xe+16:ppn=16:xk
#PBS -l walltime=2:22:22
#PBS -N mixed_flops
#PBS -j oe

cd “$PBS_O_WORKDIR”
aprun -n $((8*16)) -N 16 -d 2 ./xe_flops : \
      -n $((16*8)) -N  8 -d 2 ./xk_flops
```

  - There are limitations! See portal documentation for details.

Running Applications

- Cluster Compatibility Mode
  - For non-MPI applications that handle their own multiprocessing
  - Submit job with CCM reservation parameter

```
-l gres=ccm
```

  - Use **ccmrun** instead of **aprun**

```
$ module load ccm
$ ccmrun ./compatible_flops          \
         --host-file="$PBS_NODEFILE"
```

  - For interactive jobs, use **ccmlogin**
  - See portal documentation for more information.

Blue Waters Debugging Tools

- **DDT** - Aparallel debugger from Allinea Software, can be used for scalar, multi-threaded and large-scale parallel applications.
- **APT** - Abnormal Termination Processing from Cray, a utility for debugging. If an application takes a system trap, ATP performs analysis on the dying application.
- **STAT** - The Stack Trace Analysis Tool gathers and merges stack traces from a parallel application's processes. The tool produces call graphs. STAT is also capable of gathering stack traces with more fine-grained information, such as the program counter or the source file and line number of each frame

Blue Waters Debugging Tool - DDT

- How to use:
  - Set up for x11 forwarding: ssh -Y bw.ncsa.illinois.edu
  - Complile with the –g option: e.g. ftn -g test.f90 -o test
  - Starting a DDT debugging section with one of the following:
  - submit a job through DDT
  - manually launch a program with DDT
  - attach DDT to a running program
  - start a debug session from inside an interactive job
- The first three begin by launching ddt using the commands:
  - Module load ddt
  - ddt
- Details to follow in the tools' section

Blue Waters Debugging Tool - ATP

To use ATP for program abnormal terminations, do:

- Load atp module by ``module load atp”
- Recomplie and link the code
- Modify job script as follows:

…

module add atp export ATP_ENABLED=1 # or setenv ATP_ENABLED 1

… aprun …

- Sumbit the job
- More details to follow in the tools’ section

Blue Waters Debugging Tool - STAT

- How to use – See tools' section for details