Content

- Programming Models
- Shared Memory
- Message Passing
Familiar Aspects

- SPMD model - Fortran, C, C++ with MPI (MPI1 + subset of MPI2)
  - Full language support
  - Automatic SIMD FPU exploitation
- Linux development environment
  - User interacts with system through front-end nodes running Linux – compilation, job submission, debugging
  - Compute Node Kernel provides look and feel of a Linux environment
    - POSIX system calls (with some restrictions)
    - BG/P adds pthread support, additional socket support,
  - Tools – support for debuggers, MPI tracer, profiler, hardware performance monitors, visualizer (HPC Toolkit), PAPI
  - Dynamic libraries
  - Python 2.5

Aggregate Remote Memory Copy (ARMCI), Global Arrays (GA), UPC, ...

Restrictions (lead to significant scalability benefits)

- Space sharing - one parallel job (user) per partition of machine, one process per processor of compute node
- Virtual memory constrained to physical memory size
  - Implies no demand paging, but on-demand linking
- MPMD model limitations
Parallel Architectures

SMP / Share Everything

MPP / Share Nothing
Shared Memory Parallelism

- **Parallelism at « SMP level » within a single node**
  - Limited to the number of cores within a Node (4 on BG/P)
  - Pretty good scalability on BG/P

- **Invocation Options**
  - Automatically by using compiler options (-qsmp=auto)
    - In general not very efficient
  - By adding OpenMP compiler directives in the source code
  - By explicitly using POSIX Threads
Distributed Memory Parallelism

- Parallelism across N SMPs (often Compute Node, CN)
  - Each “CN” can only address its own (local) memory,
  - This implies modifications/re-writting of portion of codes and the use of « message passing » libraries such as MPI (or DCMF, SPI)

- « Unlimited » Scalability
  - Can scale up to thousands of processors
Mixed Mode Programming

- **Shared & Distributed Memory at the same time**
  - MPI programming style for inter-Nodes communications
  - OpenMP programming style within the Nodes

- **This has been proved to be very efficient when each node solves a « local problem » that can parallelized by using a mathematical library (ESSLSMP)**
  - In the coming years the performance processor improvement will happen through an increase of the number of cores
Terminology

- **Scalability**
  - What are the performance improvements when increasing the number of processors?

- **Bandwidth**
  - How many Mbytes can be exchanged per second between the nodes

- **Latency**
  - « Minimum time needed to send a zero length message to a remote processor »
Common Programming Models

- **MPI only – “quad mode” with enhancements**
  - Separate MPI process for each processor in the compute node
  - DMA support for each MPI process
    - Ensure network does not block when processor is computing
    - Drive network harder
  - Sharing of read-only or write-once data on each node
    - Need programming language extension to identify read-only data
    - Allow applications to overcome memory limits of quad mode

- **MPI + OpenMP**
  - OpenMP within each node – relies on cache coherence support
  - Only master thread on each node initiates communication
    - Get benefits of message aggregation
    - Exploit multiple processors to service MPI call
The Blue Gene/P MPI implementation uses the Deep Computing Messaging Framework (DCMF) as a low-level messaging interface. The Blue Gene/P DCMF implementation directly accesses the Blue Gene/P hardware through the DMA SPI interface. The MPI, DCMF, and SPI interfaces are public, supported interfaces on Blue Gene/P, and all can be used by an application to perform communication operations.

http://dcmf.anl-external.org/wiki
http://wiki.bg.anl-external.org/index.php/Base

**Multiple programming paradigms supported:** MPI, Charm++, ARMCI, GA, UPC
BG/P Programming Models (2)

**ARMCI: Aggregate Remote Memory Copy Interface**

General-purpose, efficient, and widely portable remote memory access (RMA) operations (one-sided communication) optimized for contiguous and noncontiguous data transfers.

http://www.emsl.pnl.gov/docs/parsoft/armci/

**Global Arrays**

Efficient and portable “shared-memory” programming interface for distributed-memory computers.

http://www.emsl.pnl.gov/docs/global/

**Charm++**

Explicitly parallel language based on C++ with a runtime library for supporting parallel computation called the Charm kernel. It provides a clear separation between sequential and parallel objects.

http://charm.cs.uiuc.edu/

**Berkeley Unified Parallel C**

Uniform programming model for both shared and distributed memory hardware. Single shared, partitioned address space, where variables can be directly read and written by any processor, but each variable is physically associated with a single processor.

http://upc.lbl.gov/

**GASNET: Global-Address Space Networking**

language-independent, low-level networking layer that provides network-independent, high-performance communication primitives tailored for implementing parallel global address space SPMD languages.

http://gasnet.cs.berkeley.edu/
OpenMP

- OpenMP (Open Multi-Processing) is an application programming interface (API) that supports multi-platform shared memory multiprocessing programming in C/C++ and Fortran on many architectures.
- It consists of a set of compiler directives, library routines, and environment variables that influence run-time behavior.
- Jointly defined by a group of major computer hardware and software vendors, OpenMP is a portable, scalable model that gives programmers a simple and flexible interface for developing parallel applications.
- For Blue Gene/P, a hybrid-model for parallel programming can be used:
  - OpenMP for parallelism on the node
  - MPI (Message Passing Interface) for communication between nodes
OpenMP

- OpenMP is an implementation of multithreading, a method of parallelization whereby the master thread "forks" a specified number of slave threads and a task is divided among them.
- The threads then run concurrently, with the runtime environment allocating threads to different processors.
- The section of code that is meant to run in parallel is marked accordingly, with a preprocessor directive that will cause the threads to form before the section is executed.
  - For some programs, the original code may remain intact, and OpenMP directives are added around it.
  - For most programs, some code restructuring will also be necessary for optimal execution.
OpenMP (continued)

- Each thread has an "id" attached to it which can be obtained using a function (called `omp_get_thread_num()` in C/C+)

- The thread id is an integer, and the master thread has an id of "0". After the execution of the parallelized code, the threads "join" back into the master thread, which continues onward to the end of the program

- By default, each thread executes independently

- The runtime environment allocates threads to processors depending on usage, machine load and other factors

- The number of threads can be assigned by the runtime environment based on environment variables or in code using functions
OpenMP Loop Example: Global Weighted Sum

#define N 10000 /*size of a*/
long w;
double a[N];
...

sum = 0.0;

/*fork off the threads and start the work-sharing construct*/

#pragma omp parallel for private(w) reduction(+:sum) schedule(static,1)

for(i = 0; i < N; i++) {
    w = i*i;
    sum = sum + w*a[i];
}

printf("\n %lf",sum);
Communication Libraries

- **MPI**
  - MPICH2 1.0.4p2
    - A Blue Gene/P driver has been added that implements the MPICH2 abstract device interface (ADI).
    - Optimized versions of the Cartesian functions exist (MPI_Dims_create(), MPI_Cart_create(), MPI_Cart_map()).
    - MPIX functions create hardware-specific MPI extensions.

- **DCMF** *(Deep Computing Message Framework)*

- **SPI** *(System Programming Interfaces)*
BG/P Communication Networks

1. Global Interrupt Network
The global interrupt network connects all compute nodes and provides a low latency barrier operation.

2. Collective Network
The collective network connects all the Compute Nodes in the shape of a tree. Any node can be the tree root. The MPI implementation uses the collective network, which is more efficient than the torus network for collective communication on global communicators, such as MPI_COMM_WORLD.

3. Point-to-point network (3D Torus)
All MPI point-to-point and subcommunicator communication operations are carried out through the torus network.

DMA and the collective and GI networks: The collective and GI networks do not use DMA.
The route from a sender to a receiver on a torus network has two possible paths:
- Deterministic routing
- Adaptive routing

Selecting deterministic or adaptive routing depends on the protocol that is used for the communication.

The Blue Gene/P MPI implementation supports three different protocols:
- **MPI short protocol**
  - The MPI short protocol is used for short messages (less than 224 bytes), which consist of a single packet. These messages are always deterministically routed. The latency for eager messages is around 3.3 µs.
- **MPI eager protocol**
  - The MPI eager protocol is used for medium-sized messages. It sends a message to the receiver without negotiating with the receiving side that the other end is ready to receive the message. This protocol also uses deterministic routes for its packets.
- **MPI rendezvous protocol**
  - Large (greater than 1200 bytes) messages are sent using the MPI rendezvous protocol. In this case, an initial connection between the two partners is established. Only after that will the receiver use direct memory access (DMA) to obtain the data from the sender. This protocol uses adaptive routing and is optimized for maximum bandwidth. Naturally, the initial rendezvous handshake increases the latency.
  - There are two types of rendezvous protocols: *default* and *optimized*.

**Env variables**
- **DCMF_EAGER** in Bytes for eager protocol threshold
- **DCMF_OPTRZV** in Bytes optimized rendezvous threshold
  - DCMF_EAGER <= message_size < (DCMF_EAGER + DCMF_OPTRZV)
BG/P MPI Performance Guidelines (1)

- Avoid buffered and synchronous sends; post receives in advance
- Avoid vector data and non-contiguous data types
- Risk of forcing the MPI support to allocate too much memory, resulting in failure, because it forces excessive buffering of messages.

*MPI code that can cause excessive memory allocation*

```c
MPI_Isend(cpu2, tag1);
MPI_Isend(cpu2, tag2);
...
MPI_Isend(cpu2, tagn);
```

*MPI Memory Leaks*

```c
req = MPI_Isend( ... );
MPI_Test (req);
...
do something else; forget about req ...
```
BG/P MPI Performance Guidelines (2)

- Buffer alignment sensitivity

BG/P MPI Perf is sensitive to the alignment of the buffers that are being sent or received. The L1 cache and DMA are optimized on 32-byte boundaries, at least 16-bytes for SIMD FPU.

Instead of using malloc(), use the following statement and specify 32 for the alignment parameter:

```c
int posix_memalign(void **memptr, size_t alignment, size_t size)
```

```c
buffer_ptr_original = malloc(size + 32);
buffer_ptr = (char*)( ( (unsigned)buffer_ptr_original + 32 ) & 0xFFFFFFE0 );
```

```c
struct DataInfo
{
    unsigned int iarray[256];
    unsigned int count;
} data_info __attribute__((aligned (32)));
```

or

```c
unsigned int data __attribute__((aligned (32)));
```

or

```c
char data_array[512] __attribute__((aligned(32)));
```
Collectives

- Use the hardware support in the collective network and global interrupt networks
- Supported operations
  - Barrier
  - Broadcast
  - Allreduce
  - Alltoall
  - Allgather

Barrier

- Global barriers
  - GI, Tree
- Sub communicators
  - DMA barrier with point to point messages

GI barrier latency is about 1.2us
- Sub communicators use a logarithmic barrier which is about 2500 cycles/hop
## Optimized Collectives

<table>
<thead>
<tr>
<th>Collective</th>
<th>Torus via DMA</th>
<th>Collective</th>
<th>Network</th>
</tr>
</thead>
<tbody>
<tr>
<td>Barrier</td>
<td>Binomial algorithm</td>
<td>N/A</td>
<td>Uses Global Interrupt wires to determine when nodes have entered the barrier.</td>
</tr>
<tr>
<td>Sync Broadcast (BG/L style broadcast where all nodes need to reach the broadcast call before data is transmitted)</td>
<td>Rectangular algorithm</td>
<td>Binomial algorithm</td>
<td>Uses a Collective Broadcast via spanning class route. To prevent unexpected packets, broadcast is executed via global BOR.</td>
</tr>
<tr>
<td>All-to-All(v)</td>
<td>Each node sends messages in randomized permutations to keep the bisection busy.</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>Reduce</td>
<td>Rectangular algorithm</td>
<td>Same as Collective All-reduce, but with no store on non-root nodes.</td>
<td>N/A</td>
</tr>
<tr>
<td>All-reduce</td>
<td>Rectangular algorithm</td>
<td>Uses a Collective Broadcast via spanning class route. Native tree operations, single and double pass double precision floating point operations.</td>
<td>N/A</td>
</tr>
<tr>
<td>All-gather(v)</td>
<td>Broadcast, reduce, and all-to-all based algorithms. Algorithm used depends on geometry, partition size, and message size.</td>
<td>N/A</td>
<td>N/A</td>
</tr>
</tbody>
</table>
Alltoall

- DMA based alltoall
- Uses adaptive routing on the network
  - Optimized for latency and bandwidth (latency 0.9 us/destination)
  - 96% of peak throughput on a midplane
- Alltoall performance for large messages optimized by the all-to-all mode in the DMA device
  - DCMF_FIFOMODE=ALLTOALL (20% more)
- Alltoall is disabled for sub-communicators in MPI thread mode multiple in release 1
Blue Gene/P MPI extensions (1)

```c
int MPIX_Cart_comm_create (MPI_Comm *cart_comm);
```

4D Cartesian communicator that mimics the exact hardware on which it is run. The X, Y, and Z dimensions match those of the partition hardware, while the T dimension has cardinality 1 in SMP, 2 in Dual, and 4 in VN.
Blue Gene/P MPI extensions (2)

```c
int MPIX_Pset_same_comm_create (MPI_Comm *pset_comm);
```

All nodes in a given communicator are part of the same pset (all share the same I/O node).

```c
int MPIX_Pset_diff_comm_create (MPI_Comm *pset_comm);
```

No two nodes in a given communicator are part of the same pset (all have different I/O Nodes).
MPI Environment Variables / MPICH2

- **DCMF_EAGER**
  - This value, passed through atoi(), is the smallest message that will be sent using the Rendezvous protocol. This is also one greater than the largest message sent using the Eager protocol.
  - (Synonyms: DCMF_RVZ, DCMF_RZV)

- **DCMF_COLLECTIVES**
  - When set to "0", this will disable the optimized collectives. When set to "1", this will enable the optimized collectives. Otherwise, this is left at the default.
  - (Synonyms: DCMF_COLLECTIVE)

- **DCMF_TOPOLOGY**
  - When set to "0", this will disable the optimized topology routines. When set to "1", this will enable the optimized topology routines. Otherwise, this is left at the default.

- **DCMF_ALLREDUCE**
  - Possible options: MPICH, BINOMIAL, RECTANGLE, TREE

- **DCMF_INTERRUPTS**
  - When set to "0", interrupts are disabled. Message progress occurs via polling. When set to "1", interrupts are enabled. Message progress occurs in the background via interrupts, and/or by polling. Default is "0".
  - (Synonyms: DCMF_INTERRUPT)
MPI Environment Variables / ARMCI

- **DCMF_INTERRUPTS**
  - When set to "0", interrupts are disabled. Message progress occurs via polling. When set to "1", interrupts are enabled. Message progress occurs in the background via interrupts, and/or by polling. Default is "1".
  - (Synonyms: DCMF_INTERRUPT)

- **DCMF Messaging**

- **DCMF_RECFIFO**
  - The size, in bytes, of each DMA reception FIFO. Incoming torus packets are stored in this fifo until DCMF Messaging can process them. Making this larger can reduce torus network congestion. Making this smaller leaves more memory available to the application. The default size is 8388608 bytes (8 megabytes), and DCMF Messaging uses one reception FIFO.

- etc …
Three Options

- **BG_MAPPING** environment variable
  - Equivalent to BGLMPI_MAPPING on BGL. Allows user to specify mapping as an environment variable. Options are: TXYZ, TXZY, TYXZ, TYZX, TZYX, TZX, XYZT, YZXT, ZXYT, ZYXT or a path to a mapping file
    - Rotations and point-mirroring operators (XYZT is default)

- **-mapfile** option of mpirun
  - `<CR>` separated list of physical core coordinates per task
    - x0 y0 z0 t0
    - X1 y1 z1 t1
    - ...
    - XN yN zN tN

- Use **cartesian communicators** and let BG MPI **reorder** the tasks
Mapping for Nearest-Neighbor-Communication

- **powers of 2 on BlueGene in partitioning are king!**
  - 1d -> 4d \((AxBxCxD)\) physical (smallest partition is \(4x4x2x(4)\))
    - Use 'slithering snake' mapping on small partitions (no torus)
    - TXYZ mapping on torus partitions probably fine
  - 2d \((N\times M)\) -> 4d physical \((AxBxCxD)\)
    - Try to decouple problem into two 1d -> 2d mappings, then use 'slithering snake'
  - 3d \((L\times N\times M)\) -> 4d physical \((AxBxCxD)\)
    - Try to map one \(L, N\) or \(M\) to a product of two physical dimensions then map the remaining dimensions one-to-one
    - What, if that's not possible
      - Try to split the \(D\) (intra-node dimension) into \(2x2\) and see if that works out
  - 4d -> 4d
    - Really only works well if all dimensions can be mapped one-to-one, then use the MAPPING variable
Mapping Examples (1a)

- 1d ring communicator mapping (default): worst case is 7 hops (32 nodes: 4x4x2); TXYZ
Mapping Examples (1b)

- 1d ring communicator mapping (slithering snake) -> single hop between neighbors
Mapping Examples (2)

- 8x4 2d communicator mapping (folded paper) -> single hop between neighbors
Mapping for collectives

- Collectives 'like' rectangular shaped communicators which are mapped out to a preferably rectangular 4d slab with minimum volume
  - Quite contradicting with nearest neighbor requirements
- Why?
  - Keep as many torus links busy as possible for the same collective at the same time
  - Minimize inter-collective congestion
- What if I have multiple communication patterns?
  - Try to find a layout that optimizes both
    - Rarely possible :-(
    - Optimize layout for the dominant pattern