IBM TJ Watson Research Center

Shared Memory Programming for Large Scale Machines

Kit Barton, Călin Caşcaval, George Almási, Yili Zheng, Montse Farreras, Siddhartha Chatterjee, and José Nelson Amaral

Motivation

- **Large scale machines (such as Blue Gene and large clusters) and parallelism (such as multi-core chips) are becoming ubiquitous**
- **Shared memory programming is accepted as an easier programming model, but MPI is still the prevalent programming paradigm**

Why?

- **Because typically the performance of the shared memory programs lags behind and does not scale as well as the performance of MPI codes**

Overview

- **Demonstrate that performance on large scale systems can be obtained without completely overwhelming the programmer by:**
  - Taking Unified Parallel C (UPC), a Partitioned Global Address Space (PGAS) language, that presents a shared memory programming paradigm
  - Using a combination of runtime system design and compiler optimizations
  - Running on Blue Gene/L, a distributed memory machine and the largest supercomputer available today

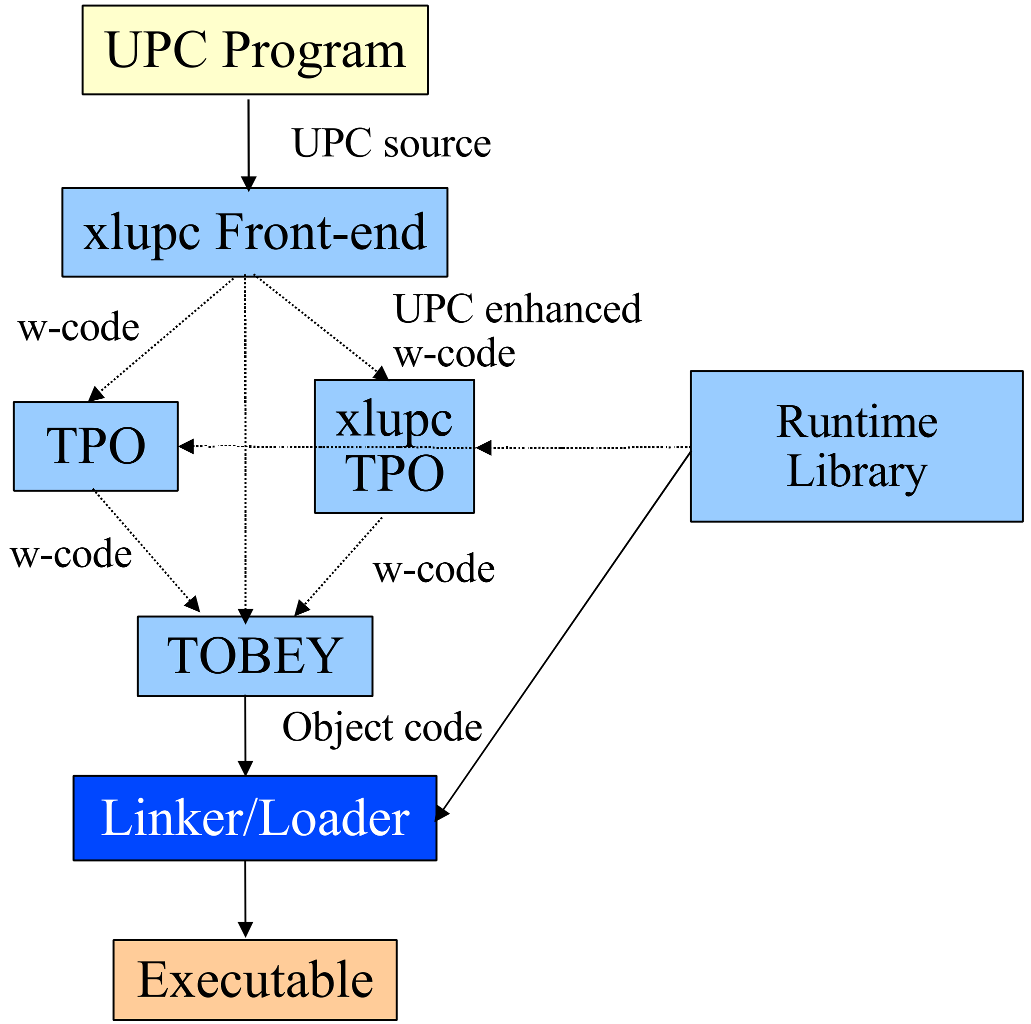
Outline

- Brief overview of UPC features
- The IBM xlupc compiler and run-time system
- Brief overview of the Blue Gene/L system
- Compiler optimizations
- Experimental results
- Conclusions

Unified Parallel C (UPC)

- **Parallel extension to C**
  - Programmer specifies shared data
  - Programmer specifies shared data distribution (block-cyclic)
  - Parallel loops (upc_forall) are used to distribute loop iterations across all processors
  - Synchronization primitives: barriers, fences, and locks
- **Data can be private, shared local and shared remote data**
  - Latency to access *local* shared data is typically lower than latency to access *remote* shared data
- **Flat threading model – all threads execute in SPMD style**

xlupc Compiler Architecture

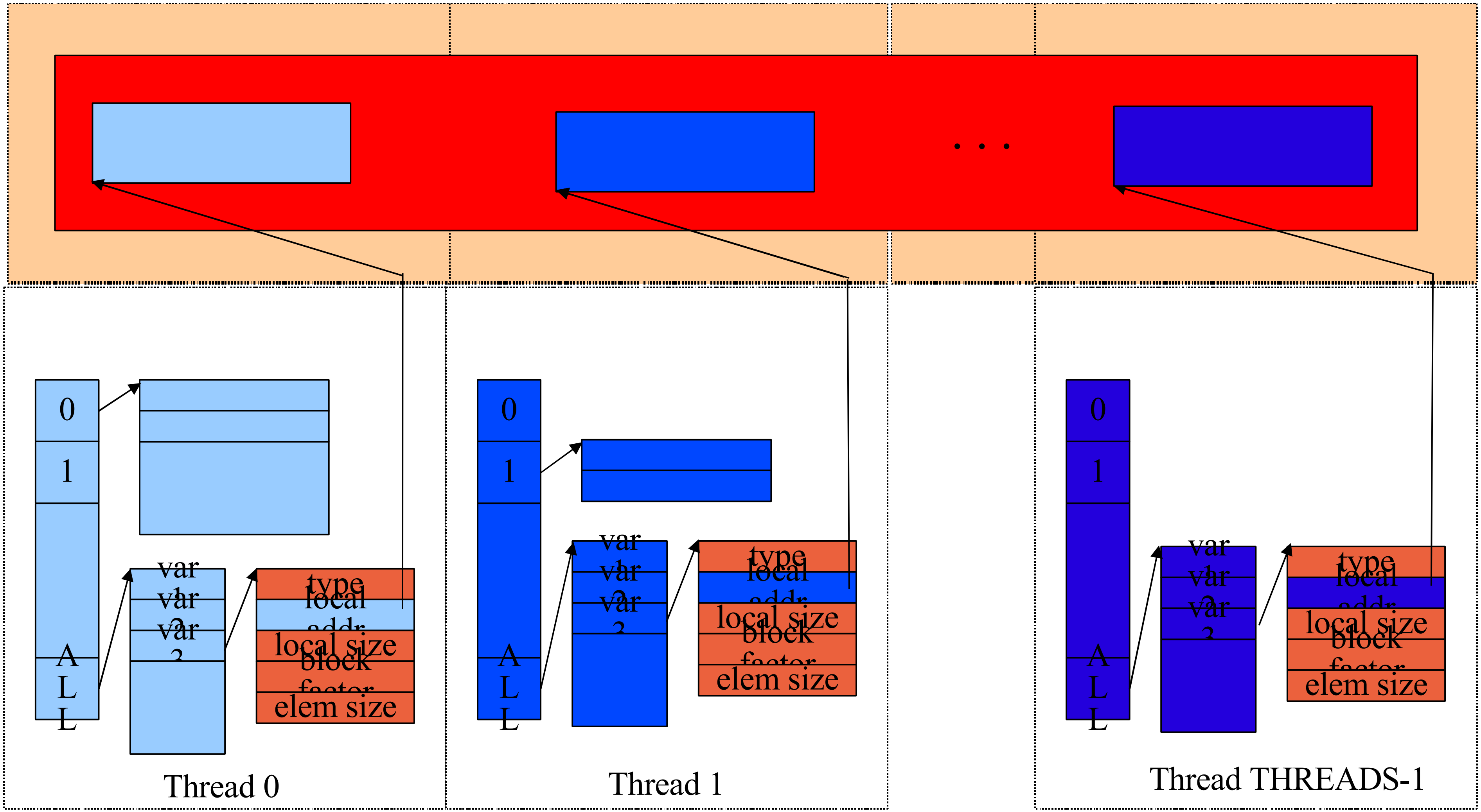


xlupc Runtime System

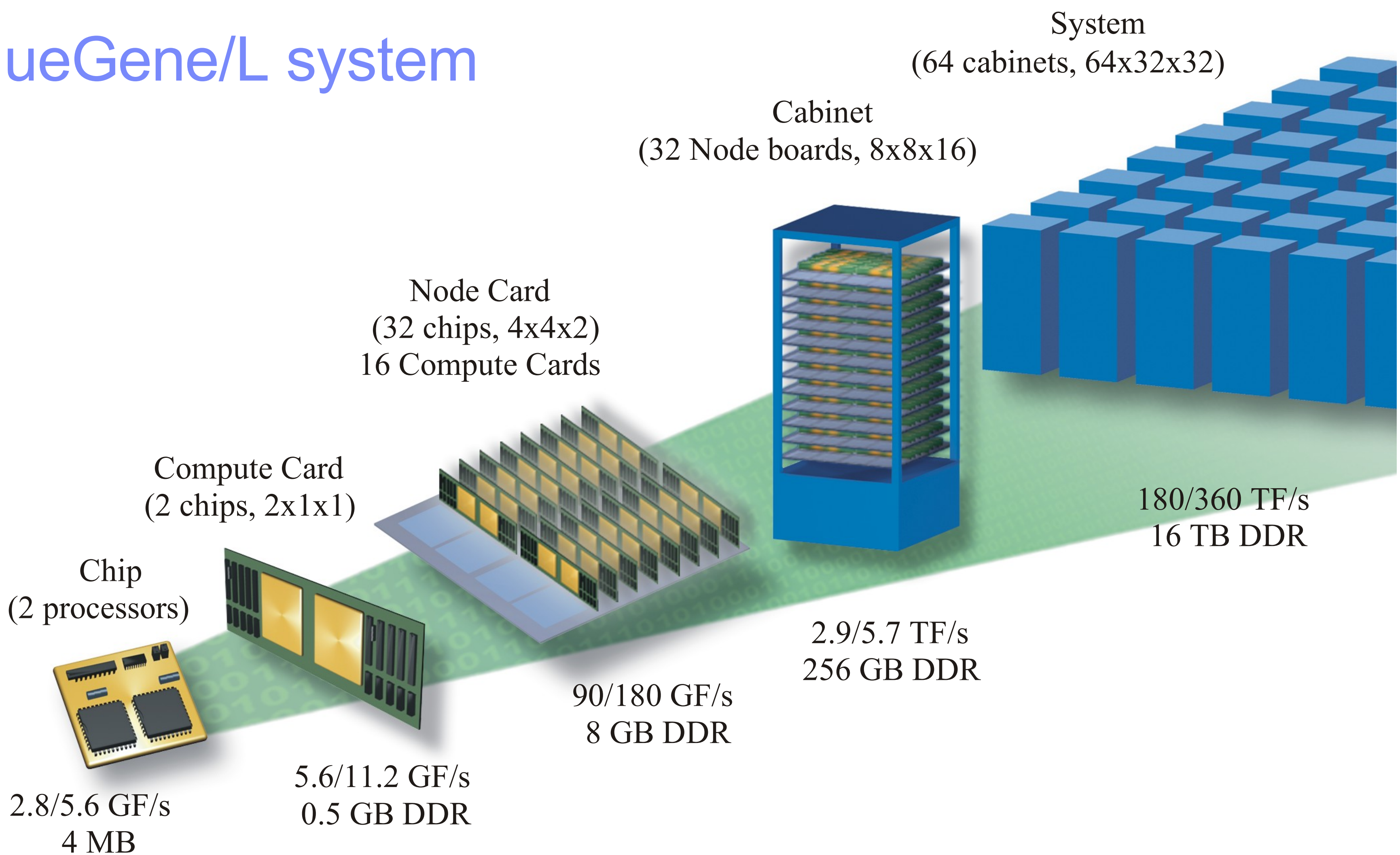
- **Designed for scalability**
- **Implementations available for**
  – SMP using pthreads
  – Clusters using LAPI
  – BlueGene/L using the BG/L message layer
- **Provides a unique API to the compiler for all the above implementations**
- **Provides management of and access to shared data in a scalable manner using the Shared Variable Directory**

Shared Variable Directory

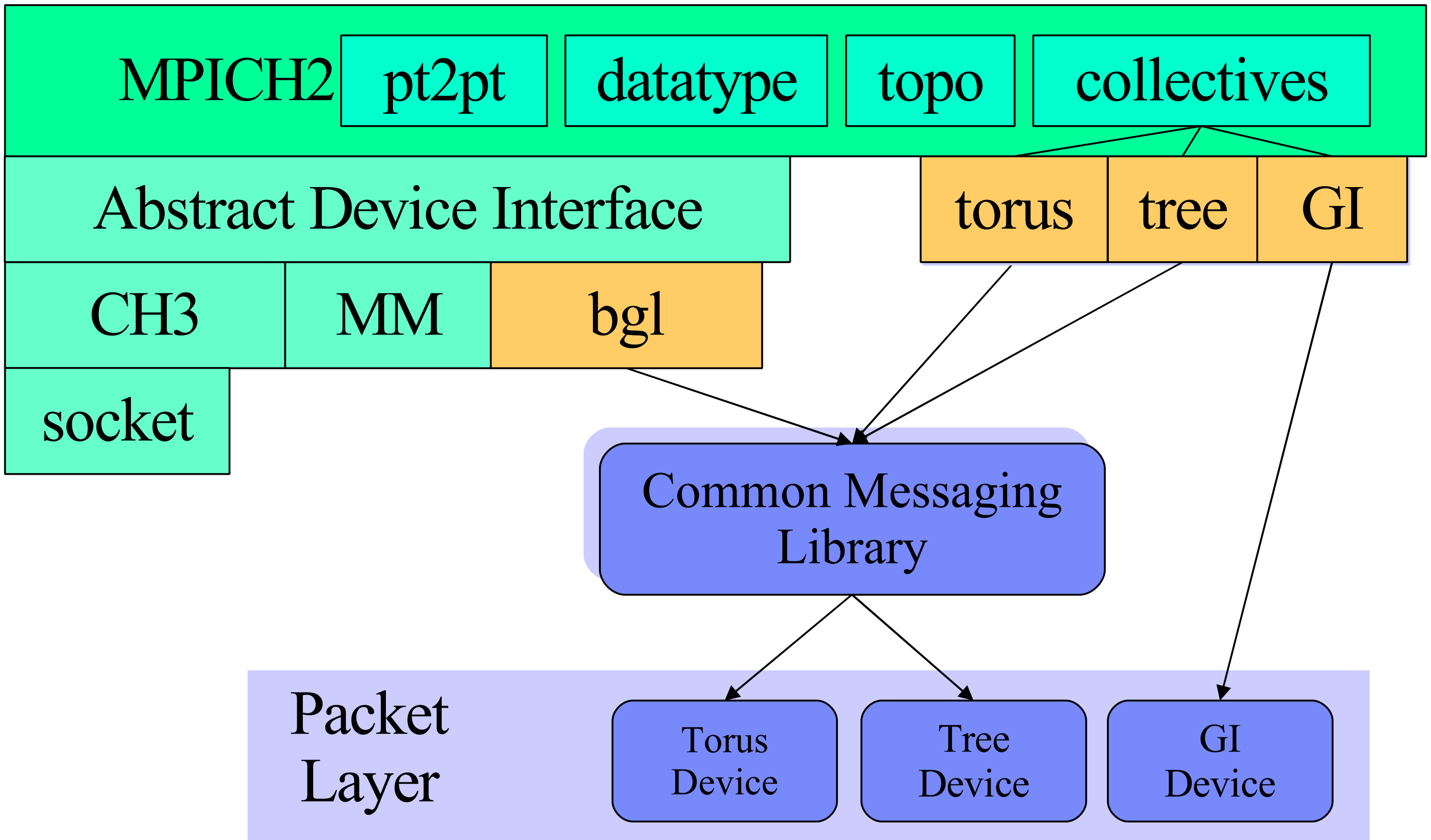
shared [SIZE/THREADS] int A[SIZE];



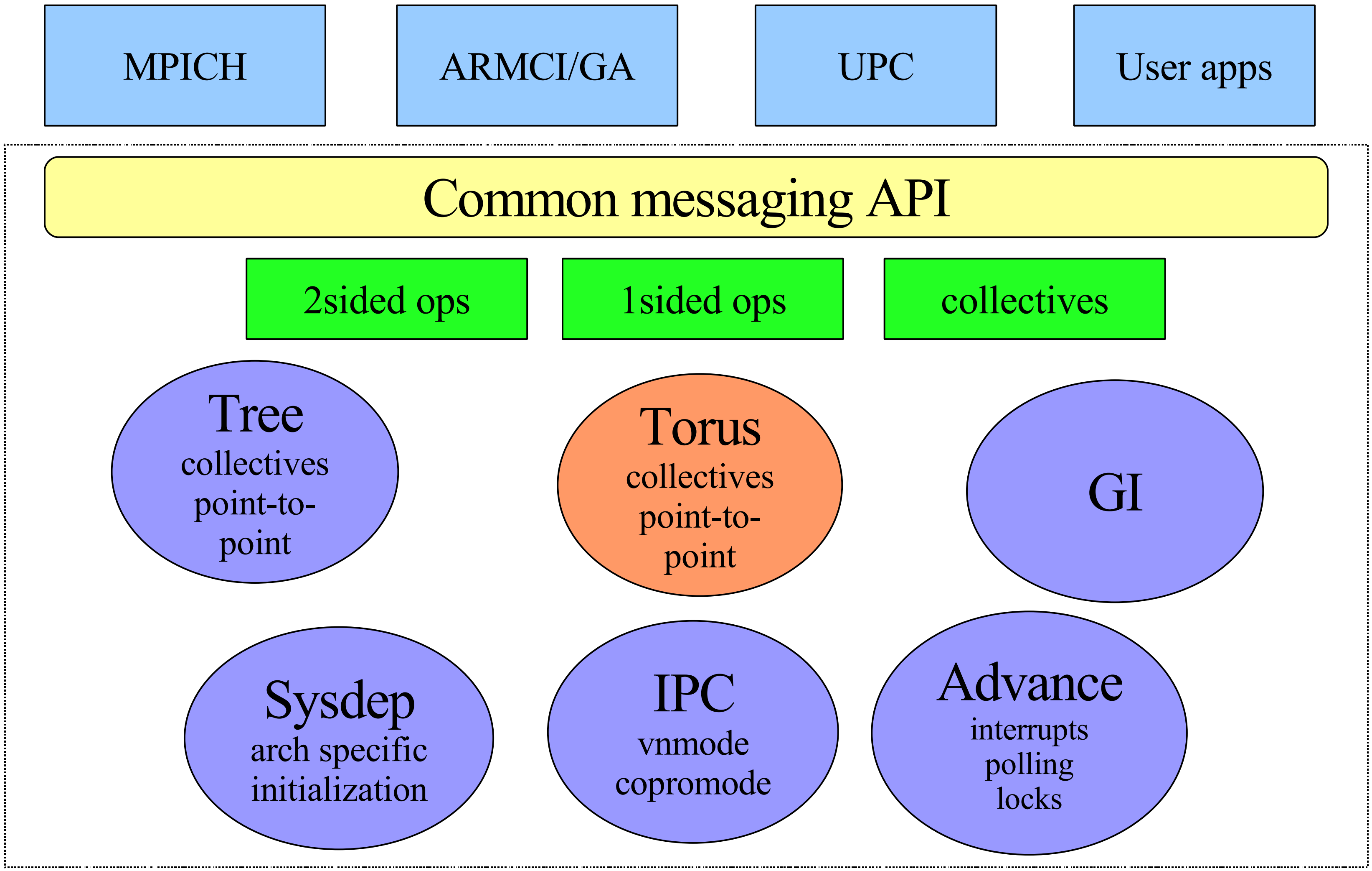
BlueGene/L system



BlueGene/L MPI Software Architecture



Common Messaging Library



Compiler Optimizations

- **Memory affinity analysis and optimization**
- **Parallel loop overhead removal**
- **Remote update operations**

Memory affinity

- **The UPC RTS uses the SVD to manage and access shared data, and thus there are several levels of indirection that impact performance**
- **If we can prove that the data is local, the shared memory access through SVD can be replaced with a direct memory access to the shared local section of memory of the thread**
- **For an affine array index expression of the form** $f(i_1, i_2, \ldots, i_n)$ **and a upc_forall affinity expression** g**, the condition for the array element to be local is:**

$$\frac{(f(i_1, i_2, \ldots, i_n))}{blocksize} \bmod MYTHREAD = g$$

Optimizing UPC Parallel Loops

```
shared [BF] int A[N];
```

affinity test

```
upc_forall(i=0; i < N; i++; &A[i])
{
    A[i] = ...;
}
```

branch

```
for (i=0; i < N; i++)
{
    if (upc_threadof(A[i]) == MYTHREAD)
        A[i] = ...;
}
```

```
for (i=MYTHREAD * BF; i < N; i+= THREADS*BF)  {
    for (j=i; j < i+BF; j++)  {
        A[j] = ...;
    }
}
```

Remote update optimization

- **Identify read-modify-write operations of the form**

  A[i] = A[i] OP value

- **Instead of issuing a *get* and then a *put* operation, we issue a *remote update* operation, where we send the address of A[i] and the value to the remote processor**
- **Take advantage of existing hardware and software primitives in Blue Gene/L:**
  - Small hardware packet size and network guaranteed delivery
  - Active message paradigm in the message layer

Experimental Environment

- **Blue Gene/L machines**
  - From a small 1 node-card (64 processors, 8 GB memory) up to 64 racks (128K processors, 32 TB memory)
- **Benchmarks:**
  - HPC Challenge: Random Access and EP STREAM Triad
  - NAS CG
- **Software:**
  - Experimental versions of the xlupc compiler, runtime system, messaging library and control system software

Random Access

```
u64Int ran = starts(NUPDATE/THREADS * MYTHREAD);
upc_forall (i = 0; i < NUPDATE; i++; i) {
    ran = (ran << 1) ^ (((s64Int) ran < 0) ? POLY : 0);
    Table[ran & (TableSize-1)] ^= ran;
}
```

- **Each update is a packet –** performance is limited by network latency
- **Important compiler optimization:**
  - Identify update operations and translate them to one sided update in messaging library

UPC Compiler Optimizations Impact

- **64 threads on a Blue Gene/L system**

		FE trans	TPO trans				
			No opt	Indexing	Update	Forall	All
Random Access	GUPS	0.0031090	0.0027048	0.0027248	0.0056082	0.0043764	0.0191830
	Time (s)	172.681	198.492	197.033	95.729	122.673	27.987
	Speedup	114.95	100.00	100.74	207.35	161.81	709.23
Stream Triad	GB/s	0.2028	0.1343	0.1769	0.1343	0.2831	32.3609
	Time (s)	23.665	35.730	27.129	35.730	16.952	0.148
	Speedup	150.98	100.00	131.71	100.00	210.77	24076.95

Theoretical GUPS limit on Blue Gene/L

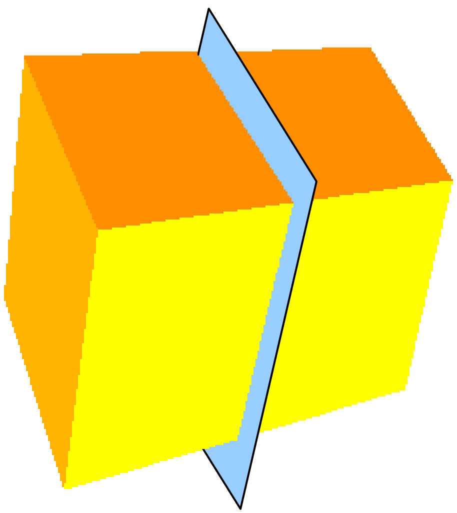
Pre-condition: one packet per update (naïve algorithm)

Update packets:

16 Byte header
4 Bytes target SVD
4 Bytes offset
4 Bytes op type,kind
8 Bytes update value

Packet size: 64 Bytes
75 Bytes on wire

$$\left\{ \begin{array}{l} \left\{ \begin{array}{l} \text{Packet size: 75 Bytes} \\ \text{Wire speed: } 4\,\dfrac{cycles}{Byte} \end{array} \right\} \rightarrow 300\,\dfrac{cycles}{packet} \\ \text{CPU speed: 700 MHz} = 1.4\,\dfrac{ns}{cycle} \end{array} \right\} \rightarrow \mathbf{2.38\cdot 10^6}\ \dfrac{packets}{second\cdot link}$$



Cross-section bandwidth:
64 x 32 x 32 torus:
2 wires/link x **32** x **32** x **2** (torus) = **4096** links
= $9.74\cdot 10^9$ packets/s

Half of all packets travel across the cross-section;
GUPS limit = **19.5**

Blue Gene/L Performance Results

Random Access

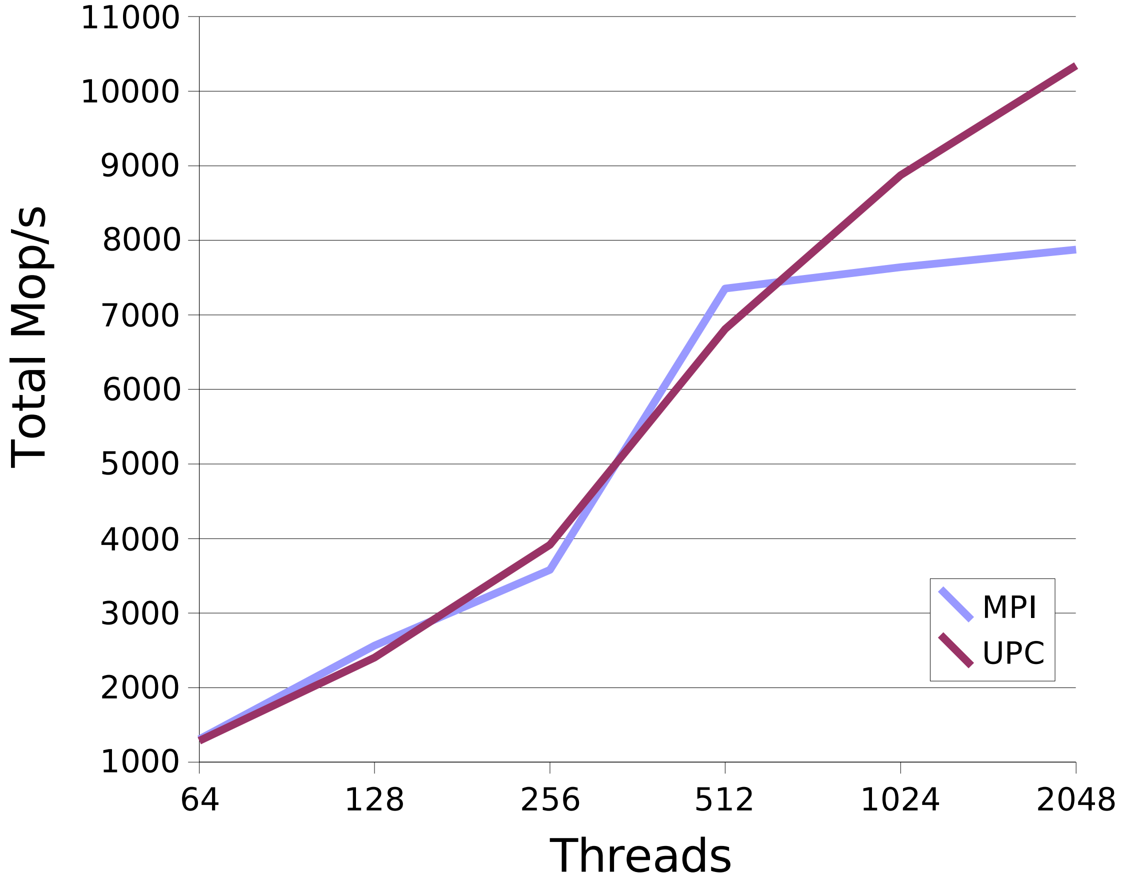
Processors	Problem Size 2^N	GUPS
1	22	0.00054
2	22	0.00078
64	27	0.02000
2048	35	0.56000
65536	40	11.54000
131072	41	16.72500

EP Stream Triad

Processors	Problem Size	GB/s
1	2,000,001	0.73
2	2,000,001	1.46
64	357,913,941	46.72
2048	11,453,246,122	1472.00
65536	366,503,875,925	47830.00
131072	733,007,751,850	95660.00

Won the HPC Challenge Productivity Award

NAS CG Class C



Conclusions

- **We have shown that scaling programs using the shared memory programing paradigm to a large number of processors is not a myth**
- **However, scaling to hundreds of thousands of threads is far from trivial; it requires:**
  - Careful design of the algorithm and data structures
  - Scalable design of run-time system and system software (communication libraries)
  - Support from the compiler
- **Lots of challenges on programming large scale machines are still waiting for compiler attention**

