

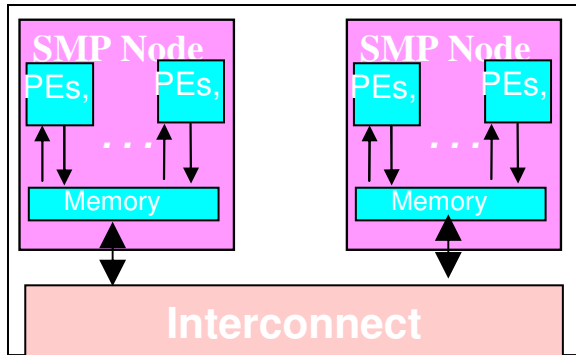


IBM Research

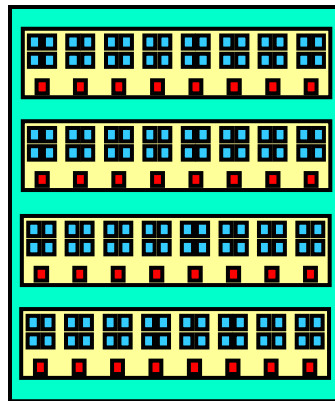
Performance and Productivity using PGAS languages

Calin Cascaval
cascaval@us.ibm.com

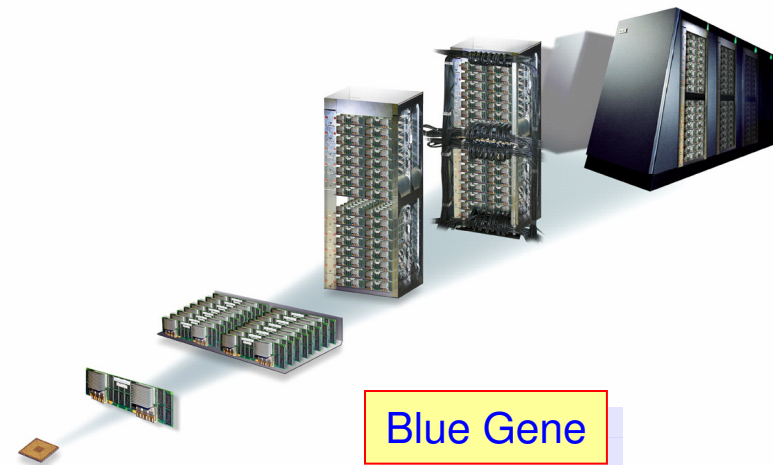
The current architectural landscape



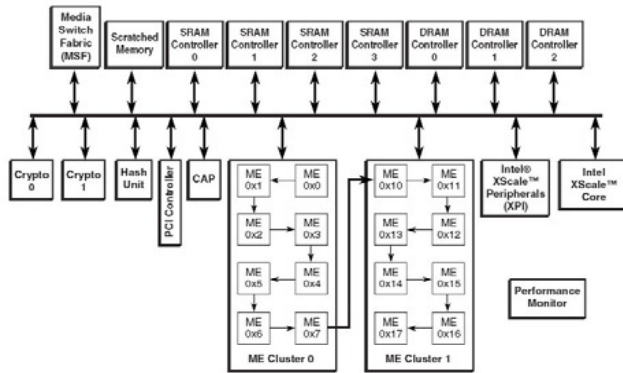
Power5 Clusters



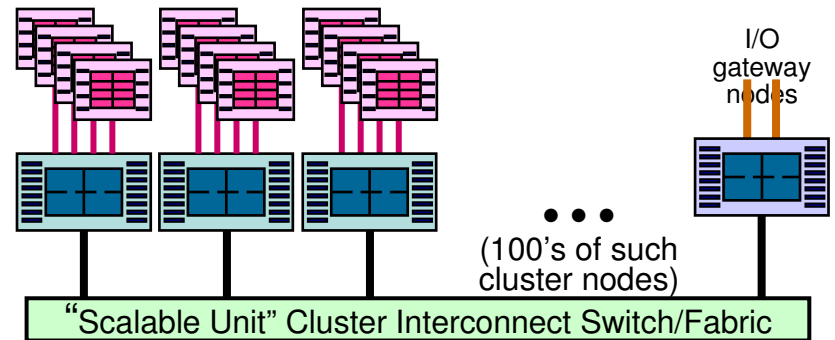
P7 supernode



Blue Gene



Multi-core w/ accelerators (*IXP 2850*)

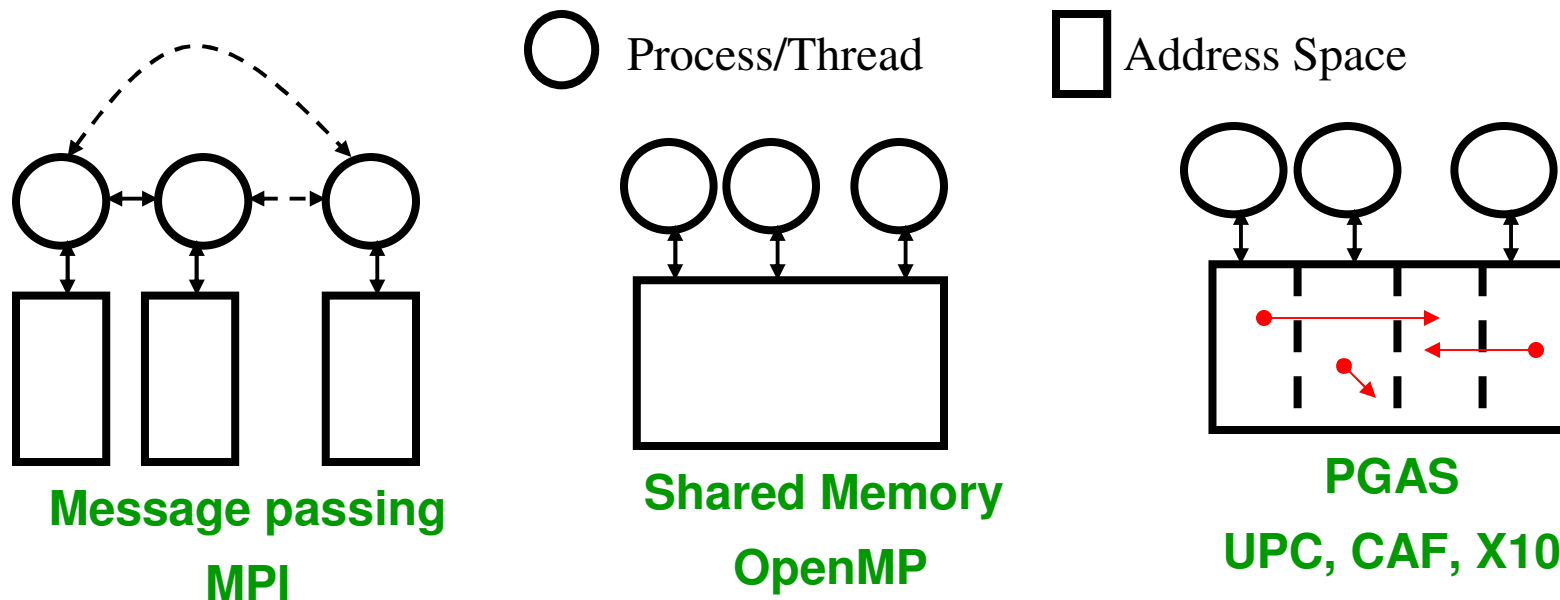


Road Runner: Cell-accelerated Opteron

How are all these systems programmed?

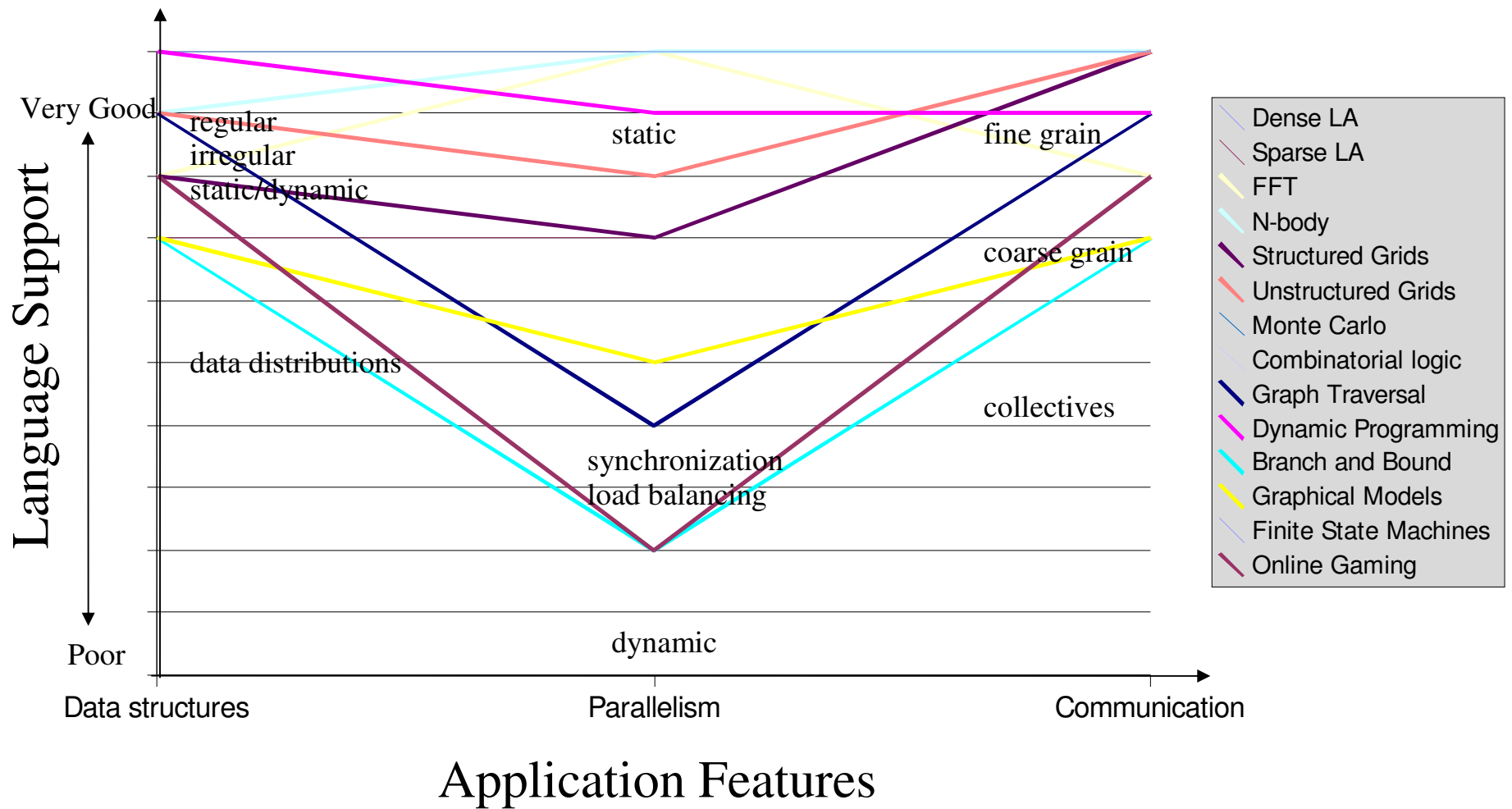
- Parallel programming techniques:
 - Automatic parallelization of sequential codes
 - Polaris, xlc -qsmp=auto, etc.
 - Successful for a limited application domain and relatively small scale
 - MPI (+ OpenMP or pthreads)
 - The dominant model today, scales well to large numbers of processors
 - Increasingly considered too complex to program
 - Parallel libraries
 - Parallel ESSL, PLAPACK, ScaLAPACK, STAPL, HTA, Intel TBB
 - Composability
 - Explicit parallel languages or parallel language extensions
 - OpenMP – small scale (hundreds of threads)
 - PGAS: UPC, CoArray Fortran, Titanium, X10, Chapel
 - Fortress

What is Partitioned Global Address Space (PGAS)?

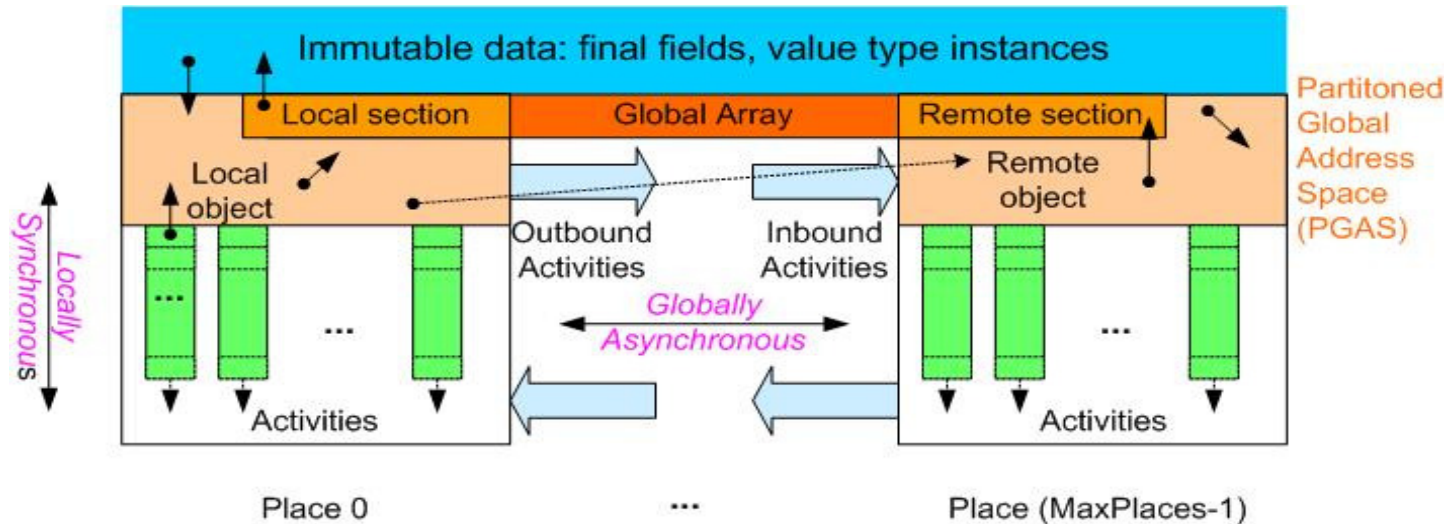


- Computation is performed in multiple **places**.
- A place contains data that can be operated on remotely.
- Data lives in the place it was created, for its lifetime.
- A datum in one place may reference a datum in another place.
- Data-structures (e.g. arrays) may be distributed across many places.
- Places may have different computational properties

UPC – Language Expressivity



Asynchronous PGAS



- **Asynchrony**
 - Simple explicitly concurrent model for the user: **async (p) S** runs statement S “in parallel” at place p
 - Controlled through **finish**, and local (conditional) **atomic**
- Used for active messaging (remote asyncs), DMAs, fine-grained concurrency, fork/join concurrency, do-all/do-across parallelism
 - SPMD is a special case

Concurrency is made explicit and programmable.

UPC Performance Gaps

- **Data distributions**
 - Express data locality and distribution
- **Efficient single thread performance**
 - Exploit existing, optimized serial libraries
 - Compiler optimizations: parallel loop, privatization
- **Efficient and scalable communication**
 - Collective operations
 - Compiler optimizations: communication scheduling and aggregation, hw exploit
- Fine grain threading for load balancing
- Synchronization
- Parallel I/O

Combination of system, runtime and compiler opts.

Data Layout Extensions to UPC

- Proposed Extensions
 - Allow support for true shared multi-dimensional arrays
 - Allow creation of a processor topology with these arrays
- Ensure that the underlying data layout is suitable for use with existing optimized serial libraries
- Checkerboard layout critical for load balancing dense factorization methods

0	1	0	1	0	1	0	1
2	3	2	3	2	3	2	3
0	1	0	1	0	1	0	1
2	3	2	3	2	3	2	3
0	1	0	1	0	1	0	1
2	3	2	3	2	3	2	3
0	1	0	1	0	1	0	1
2	3	2	3	2	3	2	3

```
#pragma processors MyDist(2,2)
shared [2][2] (MyDist) double A[16][16]
```

UPC matrix multiplication with multidimensional tiling

```
#define BLK_SIZE (sizeof(double)*b*b)
shared [b][b] double A[M][P], B[P][N], C[M][N];

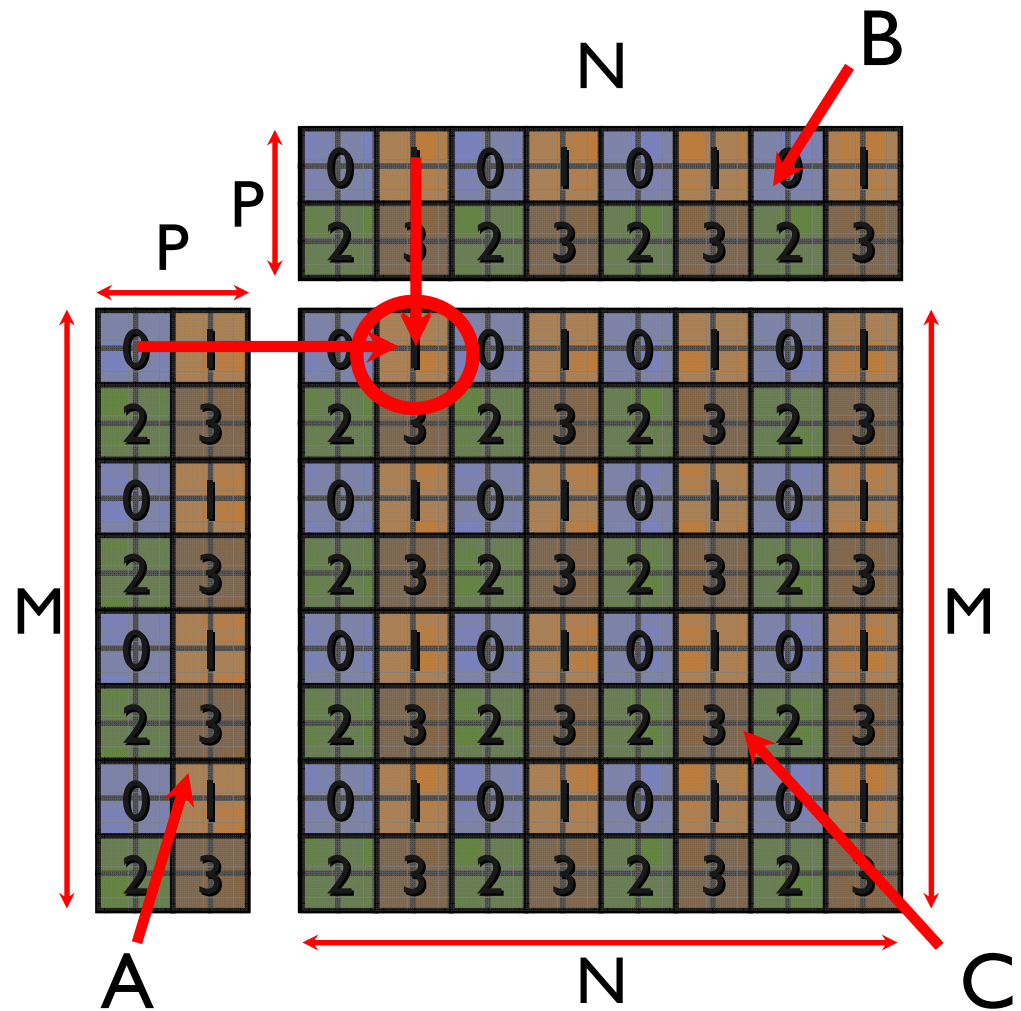
upc_forall(int ii = 0; ii < M; ii += BS; continue)
  upc_forall(int jj = 0; jj < N; jj += BS; &C[ii][jj]) {
    double scratchA[b][b], scratchB[b][b];

    upc_memget(scratchA, &A[ii][jj], BLK_SIZE);
    upc_memget(scratchB, &B[ii][jj], BLK_SIZE);

    dgemm(&C[ii][jj], &scratchA, &scratchB, b, b, b);
  }
```

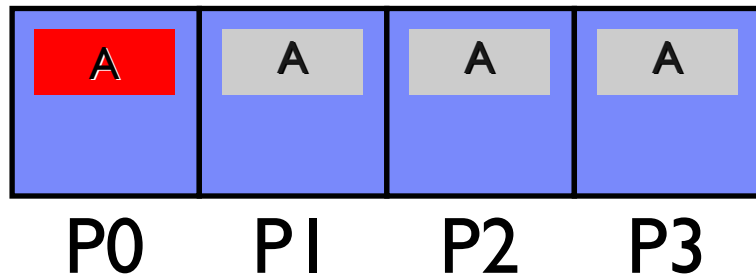
Algorithm Example: DGEMM

- $C = A * B$
- Use one-sided communication to handle data transfers
- Initial implementation used uncoordinated gets
- Ex: Calculate $C[0:1][2:3]$
- owned by processor 1
- Need data owned by 0 and 1 in first round
- Need data owned by 1 and 3 in second round
- Non-scalable approach due to $O(P * M * N)$ uncoordinated gets
- Can we do better?



Thread vs. Data-Centric Communication

- Example: Send P0's version of A to P1



■ MPI Code (Thread-Centric):

```
double A;
MPI_Status stat;
if(myrank == 0) {
    A = 42.0;
    MPI_Send(&A, 1, MPI_DOUBLE, 1, 0,
             MPI_COMM_WORLD);
} else if(myrank == 1)
    MPI_Recv(&A, 1, MPI_DOUBLE, 0,
             MPI_ANY_TAG, MPI_COMM_WORLD, &stat);
```

■ UPC Code (Data-Centric):

```
shared [1] double A[4];
if(MYTHREAD == upc_threadof(&A[0])) {
    A[0] = 42.0;
    upc_memput(&A[1], &A[0], sizeof(double));
}
```

Collective Communication

- An operation called by all processes together to perform globally coordinated communication
 - May involve a modest amount of computation, e.g. to combine values as they are reduced
 - Can be extended to teams (or communicators) in which they operate on a subset of the processors
- Defines an abstraction for communication
 - Pass the responsibility of tuning the communication schedule to the runtime system

Team Construction

- MPI requires explicit team (communicator) objects
 - They are heavy weight objects that must be constructed outside the critical path
 - They require the user to explicitly specify the processor ranks that are involved in the teams
- Logical extension to MPI's process-centric programming model
- PGAS collectives take advantage of distributed arrays and present a more data-centric programming model
 - Can we create a novel collective interface to go along with this programming model?
 - Can we make it scale?
- Our approach: Have the user specify the blocks of the shared data to operate on
 - Let the runtime figure out the mapping of data to processor and dynamically construct the teams

Example: Broadcast

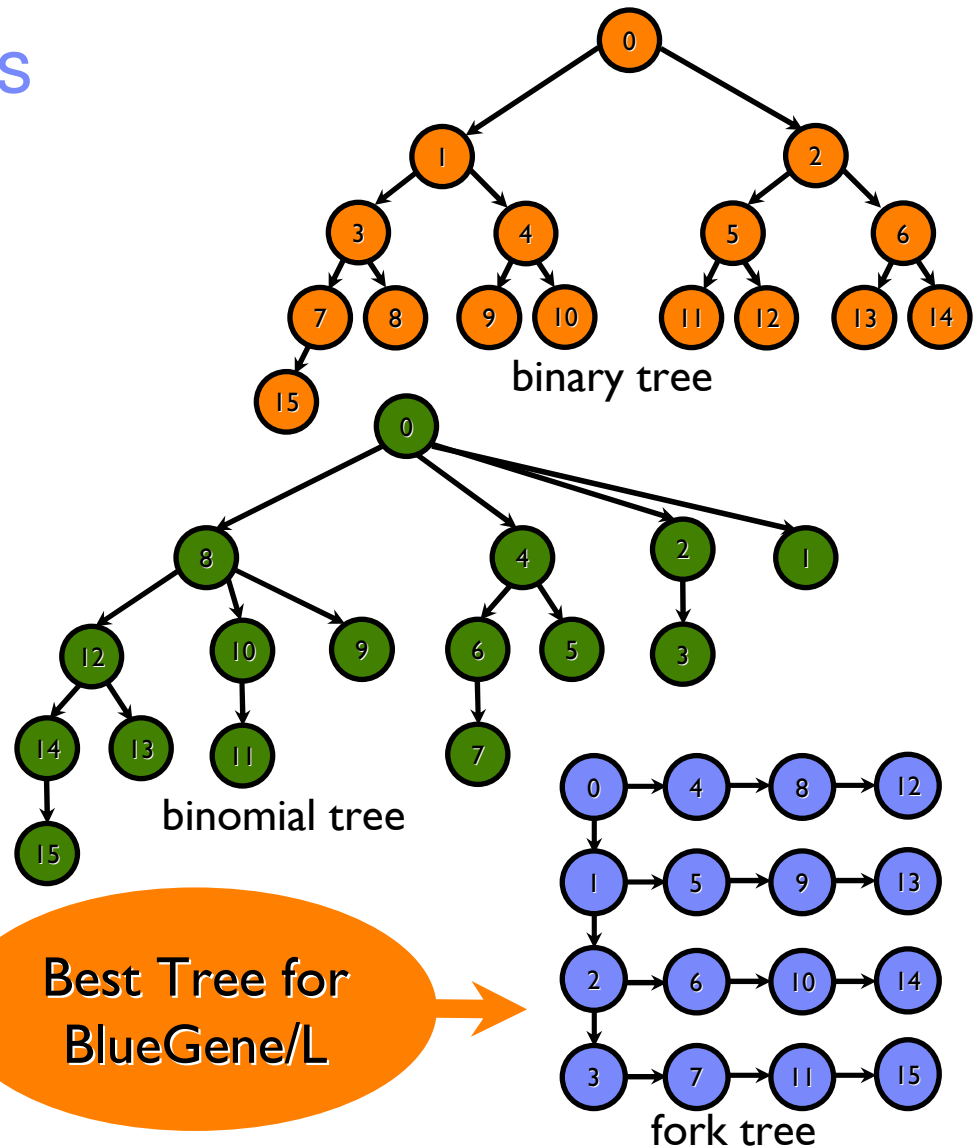
- Representative of one-to-many
- Broadcast one value to every other row and every other column
- Use Matlab style notation in the interface

```
shared [2][2] double dst[N][N];  
double even, odd;  
upc_stride_broadcast(dst<0:2:N-1,0:2:N-1>, even, sizeof(double));  
upc_stride_broadcast(dst<1:2:N-1,1:2:N-1>, odd, sizeof(double));
```

- Collective arguments are the same regardless of the the “shape” of the destination and the or the number of processors involved

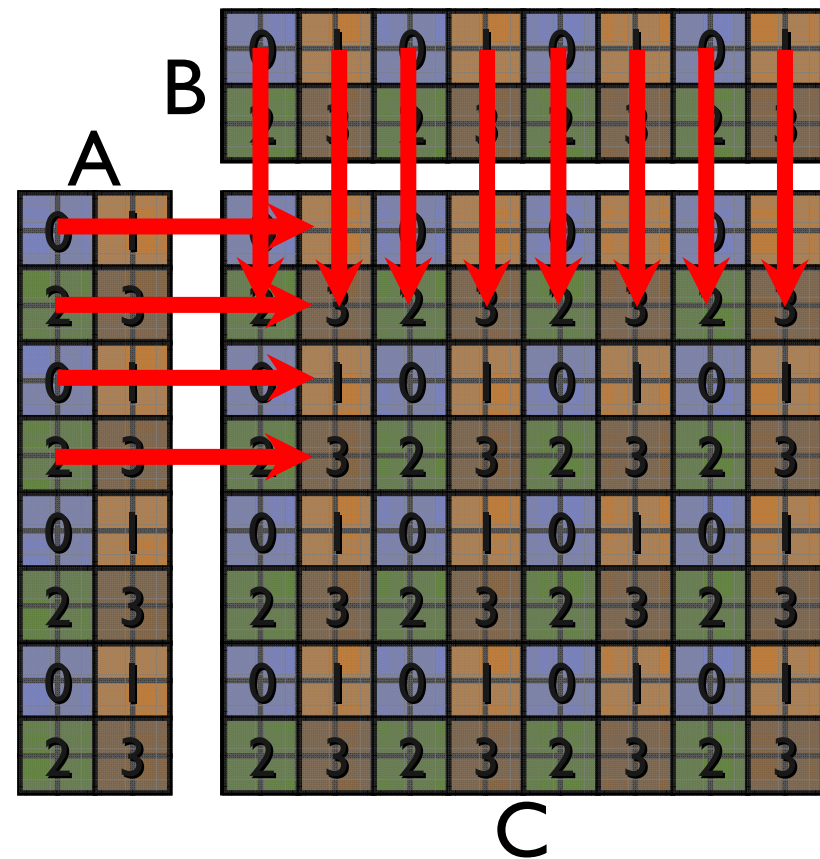
Optimizing the Collectives

- Tree topology for processor communication is critical for performance
- Overhead of injecting messages onto network can not be parallelized
 - Sending to intermediary nodes alleviates serial bottleneck and distributes work across machine



Back To DGEMM

- Implementation with uncoordinated gets did not scale
- Change the implementation to use scalable collectives
- Use simultaneous team broadcasts
 - First round goes across processor rows
 - Second round goes down processor columns



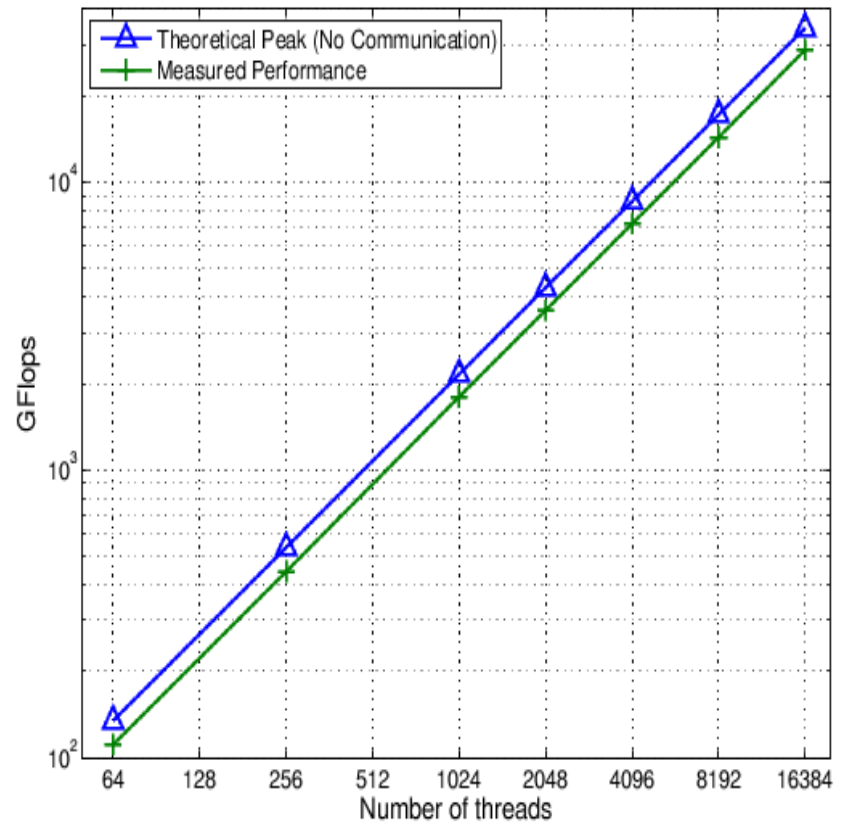
... And the code

```
#define BLK_SIZE (sizeof(double)*b*b)
shared [b][b] double A[M][P], B[P][N], C[M][N];
shared [b][b] double scratchA[b*Tx][b*Ty], scratchB[b*Tx][b*Ty];
int myrow=MYTHREAD/Ty;
int mycol=MYTHREAD%Ty;

for(k=0; k<P; k+=b) {
  for(i=0; i<M; i+=Tx*b) {
    upc_stride_broadcast(scratchA<myrow, :>, &A[i+myrow*b][k], BLK_SIZE, 0);
    for(j=0; j<N; j+=Ty*b) {
      upc_stride_broadcast(scratchB<:, mycol>, &B[k][j+mycol*b], BLK_SIZE, 0);
      /* matmult*/
      dgemm((double*) &C[i+myrow*b][j+myrow*b],
            (double*) &scratchA[myrow*b][mycol*b],
            (double*) &scratchB[myrow*b][mycol*b], b, b, b);
    }
  }
}
```

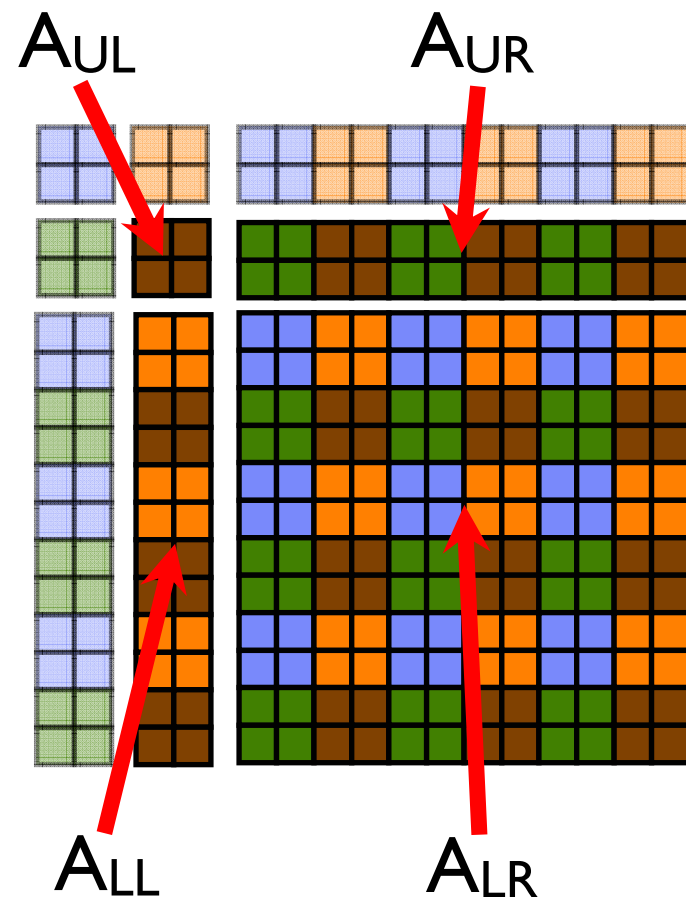
DGEMM Performance (Blue Gene/L 16 racks)

- Linearized pairs of the 4-D torus dimension to get 2D processor grid
- Use ESSL for serial computation
- B^2 doubles broadcast for every $2B^3$ flops
- Experiments scale problem size w/ processor count
- 28.8 TFlops @ 16,384 processors with linear scaling

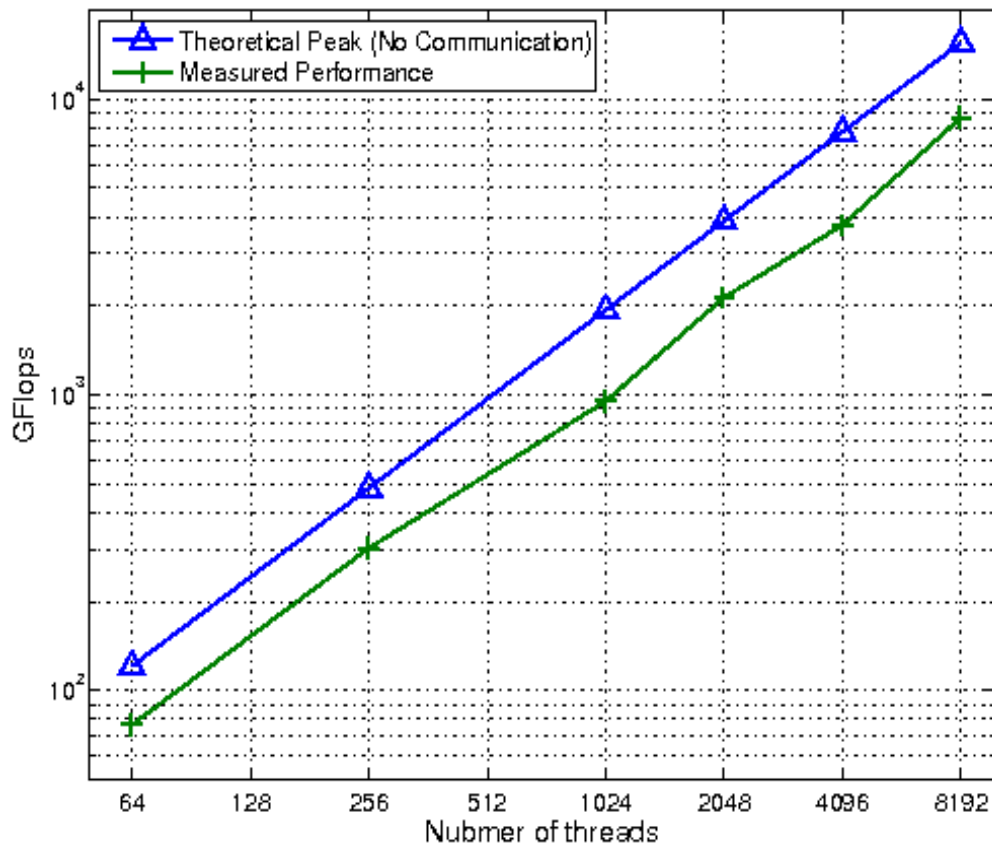


Dense Cholesky Factorization

- Factor A into $U^T U$
 - Relies on Team Broadcast
- Recursive algorithm
 - Factor Upper-Left corner of A (A_{UL})
 - Update A_{UR} using triangular solve
 - Outer product of A_{UR}^T and A_{UR} to get A_{LR}
- Full Code in PPoPP08 paper
- ESSL for local computation

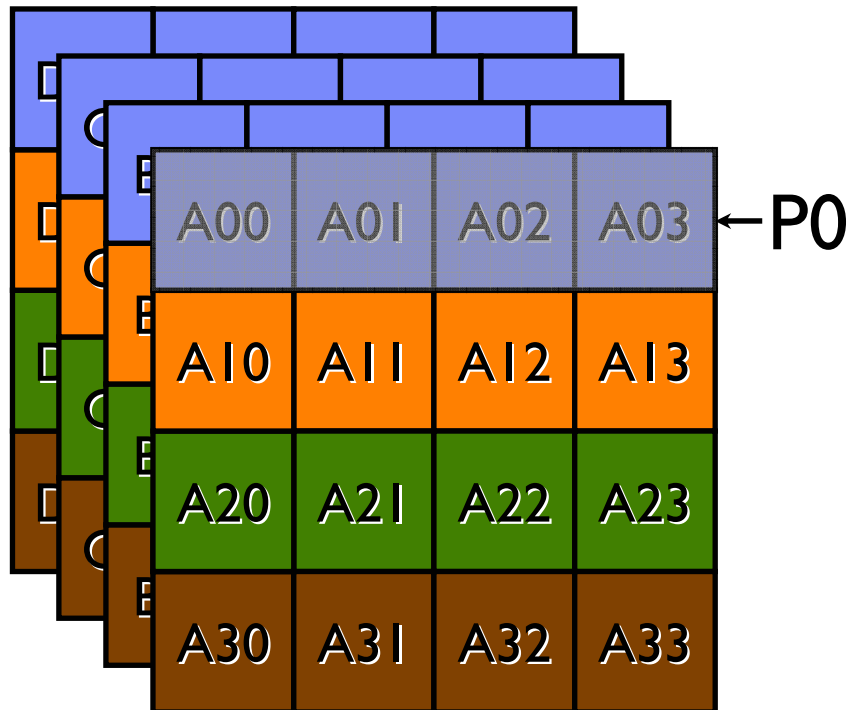


Cholesky Factorization Performance



- Processor layout identical to DGEMM
- Broadcast is no longer strictly along rows or columns of processor grid
- Rectangular processor grid over a square matrix leads to load balance issues
- 8.6 TFlops @ 8,192 processors w/ linear scaling

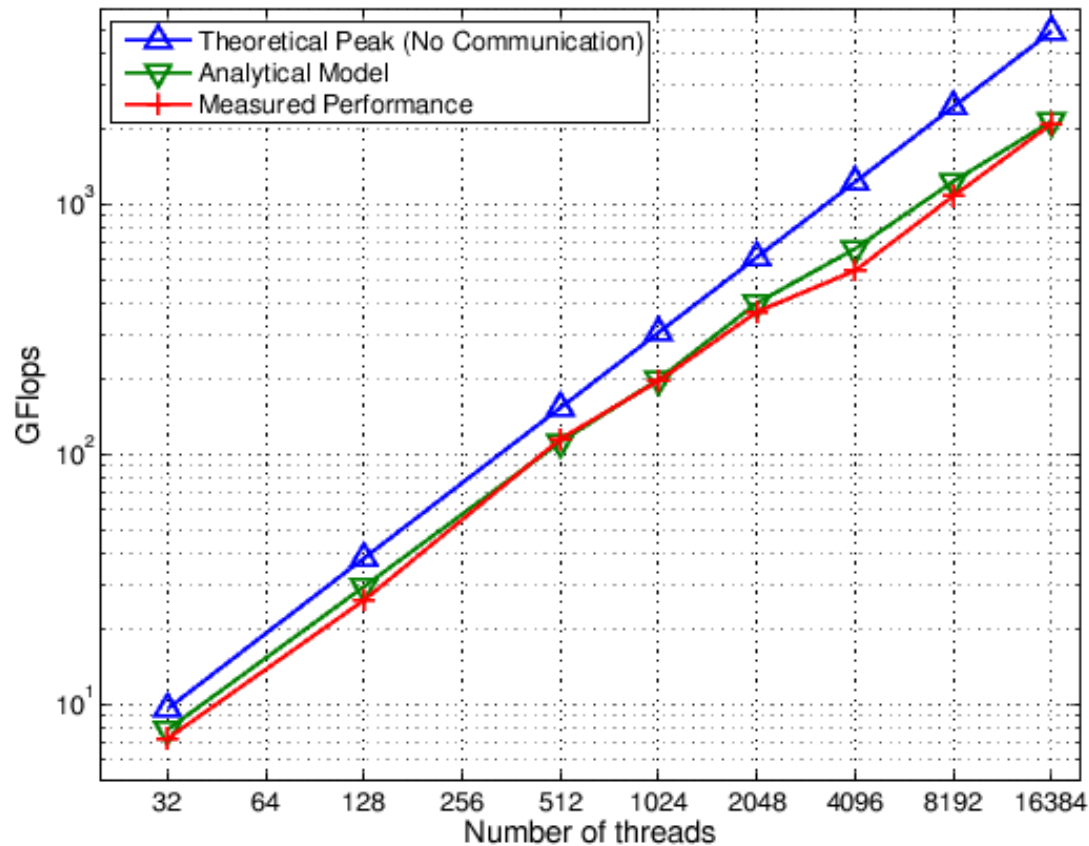
3D-FFT



Each processor owns
a row of 4 squares

- Perform a 3D FFT across a large rectangular prism
 - Perform an FFT in each of the 3 dimensions
 - Need to Team-Exchange for other 2 of the 3 dimensions for a 2-D processor layout
- Algorithm:
 - Perform FFT across the rows
 - Do an exchange within each plane
 - Perform FFT across the columns
 - Do an exchange across planes
 - Perform FFT across the last dimension

3D-FFT Performance



- Every processor exchanges all of its data in each round
- Network bandwidth limits performance
- Created performance model that exposes bisection bandwidth limits
- 2.1 TFlops @ 16,384 processors

Acknowledgments and further references

- Gheorghe Almasi
- Kit Barton
- Rajesh Nishtala
- Ettore Tiotto
- Philip Luk
- Jose Castanos
- Vijay Saraswat
- Montse Farreras
- Anthony Bolmarcich

■ Publications

- *Performance without Pain = Productivity: Data Layout and Collective Communication in UPC*, Rajesh Nishtala, George Almasi, and Calin Cascaval – PPOPP 2008
- *Multidimensional blocking in UPC*, Christopher Barton, Calin Cascaval, George Almasi, Rahul Garg, Jose Nelson Amaral, and Montse Farreras – LCPC 2007
- *Shared Memory Programming for Large Scale Machines*, Christopher Barton, Calin Cascaval, George Almasi, Yili Zheng, Montse Farreras, Siddhartha Chatterjee, Jose Nelson Amaral – PLDI 2006

www.alphaworks.ibm.com/tech/upccompiler

Conclusions

- Productivity and performance are tied together
- Compilers are not the magic wand
 - Work well for languages and patterns that we have studied for a long time
 - Have to handle general purpose languages, and therefore the number of patterns is very large
- Programmers don't necessarily expect miracles
 - They know the application and would be willing to express some of that knowledge – if we give them the right tools
- Languages, libraries, compilers, runtime systems, and architectures must interact and integrate better