



IBM Research

High productivity at what price?

Rajesh Nishtala
George Almasi
Calin Cascaval

Introduction

- **High performance library generators common for single-processor, on the way for multicore**
- **Truly high performance machines: one-of-a-kind**
 - Need 99.99...9% of the available performance
 - Programmers **still** cheaper than machine
 - HPL for Blue Gene/L: 100k lines of code, ~3PY development

Simple linear algebra on large parallel machines

- **Assume serial problem solved**
 - ESSL
- **Assume communication problems solved**
 - Blue Gene/L communication library (DCMF)
- **Assume extensible UPC implementation**
- **How much extra complexity?**

Scalable Cholesky factorization, FFT written in UPC

■ Required:

- Intuitive source code
- ~ 100 lines

■ Problems:

- 2D blocked data structures
- Processor distributions
- Communication patterns

■ Todos:

- Add multi-blocked data structures to UPC
- Add processor layouts
- Expand UPC collective framework as needed

UPC on Blue Gene/L

■ **Compiler:**

- 1.2 spec compliant
- Based on IBM XL suite
 - PowerPC + AIX,
 - PowerPC + Linux
- PERCS deliverable
- Cross-compiles for BG/L

■ **Runtime:**

- Shared memory, distributed, hybrid
- Low level transport for LAPI, sockets, Blue Gene DCMF.

Extending UPC with n-dimensional blocked arrays

■ Basic UPC syntax:

```
shared [b] <type> arrayname [d1][d2] ... [dn]
```

■ Extended syntax:

```
shared [b1][b2] ... [bm] <type> arrayname [d1][d2] ... [dn]
```

- where $m < n$
- blocking factors are not required to divide dimensions
- UPC pointer arithmetic and casts can be expanded to handle multi-blocked pointer-to-shared

Parallel Blocked Cholesky Factorization

```

void cholesky_mb (shared [B][B] double A[N][N], int N, int B) {

    int kk, info;
    for (kk=0; kk<N; kk+=B) {
        int k1 = min (B, N-kk);
        if (upc_threadof(A[kk][kk]) == MYTHREAD) {
            double * a_local = &A[kk][kk];
            dpotrf ("Upper", &k1, a_local, &B, &info); assert (!info);
        }
        upc_barrier;
        if (kk + B < N) {
            trisolve_mb (A, N, B, kk, kk+k1);
            update_mb (A, N, B, kk, kk+k1);
        }
    }
}

```



Cholesky parallel rank-k update (naive version)

```
void update_mb (shared [B][B] double A[N][N],
               int N, int B, int col0, int col1) {

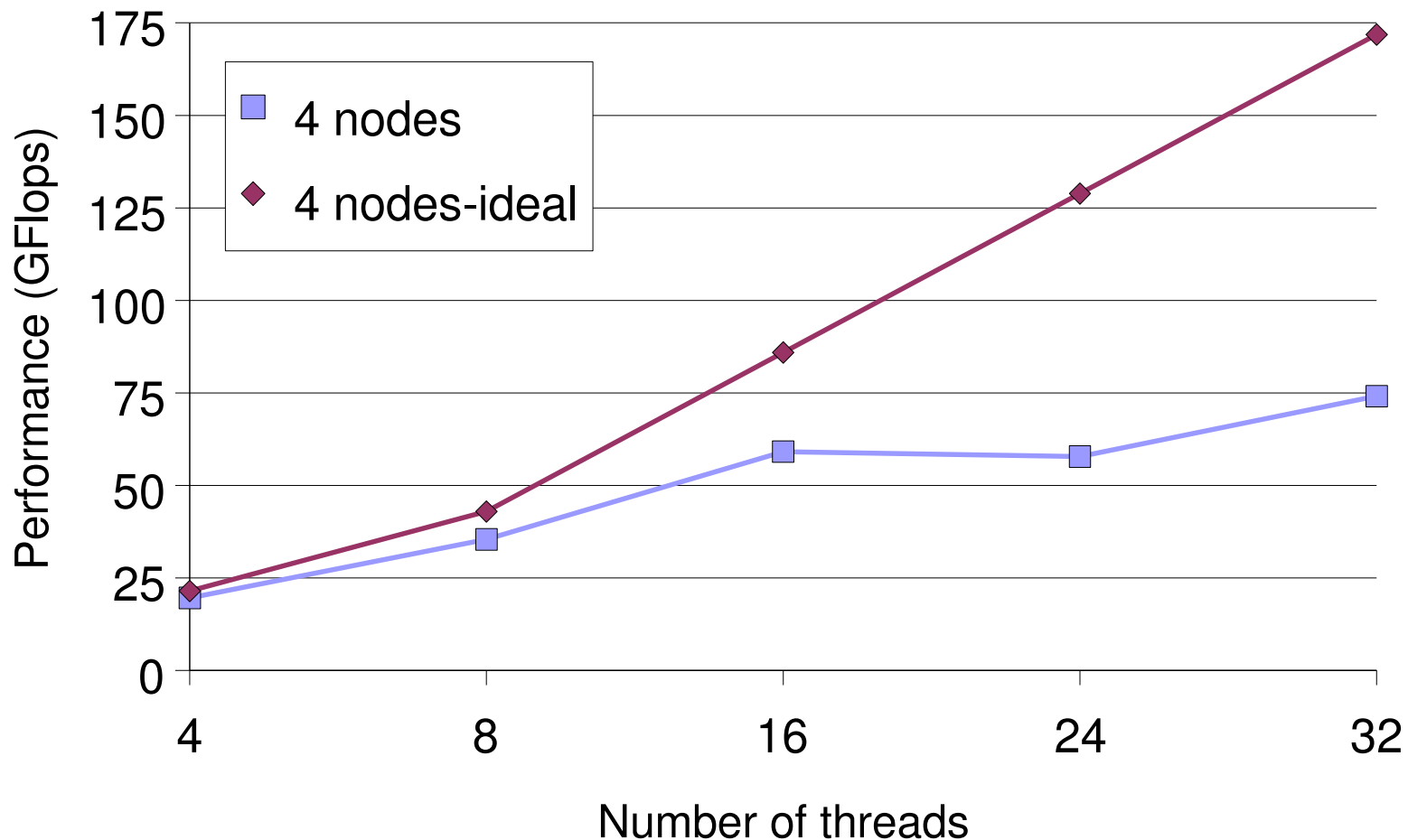
    double a_local[B*B], b_local[B*B];

    upc_forall(ii=col1; ii<N; ii+=B; continue)
        upc_forall(jj=col1; jj<ii+B; jj+=B; &A[ii][jj]) {

            upc_memget (a_local, &A[ii][col0], sizeof(double)*B*B);
            upc_memget (b_local, &A[jj][col0], sizeof(double)*B*B);

            double alpha = -1.0, beta = 1.0;
            int m         = min(N-ii,B);
            int n         = min(N-jj,B);
            int p         = col1-col0;
            dgemm ("T", "N", &n, &m, &p, &alpha, b_local, &B,
                 a_local, &B, &beta, (void *)&A[ii][jj], &B);
        }
    upc_barrier;
}
```

Performance results: Naive Cholesky scaling



The name of the problem is communication

GOOD

- **Simple approach (memget)**

- **Data-centric approach**

- **“Network neutrality”**

BAD

- **Point-to-point communication**

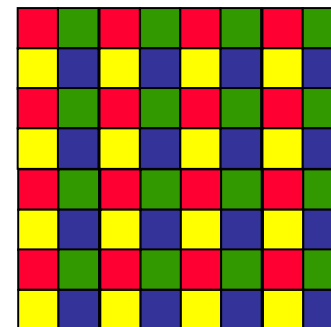
- **Same node, same data, multiple deliveries**
- **Load balance problems**

- **Unfavorable to network properties**

Solutions

■ Partially abandon data-centric approach

- Introduce processor distributions
 - Fixes problems with load balance
- Assign array blocks to processors

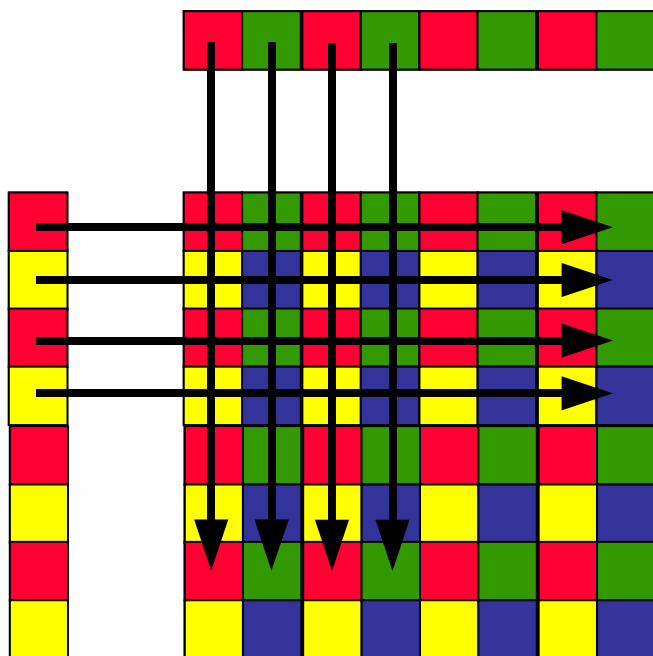


■ Effective and expressive collectives:

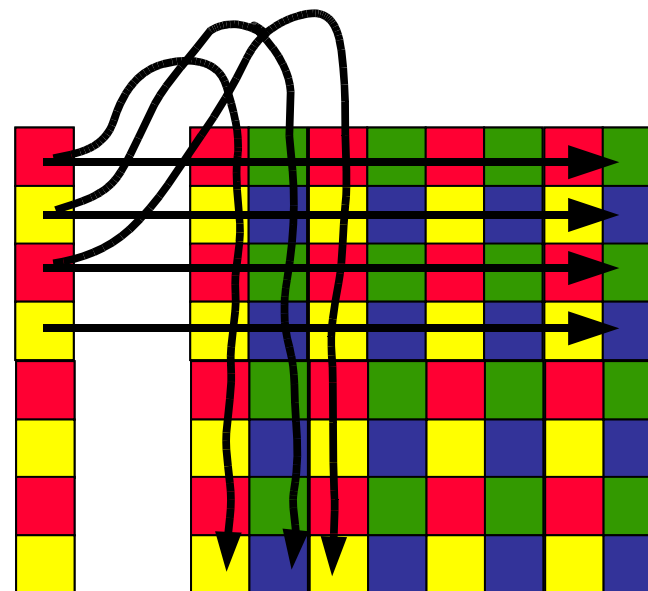
- Express communication requirements in 1 line of code
- Best bet: use Fortran-90-like notation (familiarity, compactness)

The broadcast pattern

Matrix multiply



Cholesky update



Requirements for collective communication

- **Multiple simultaneous collectives**
 - teams aka communicators
- **Data-centric instant teams**
 - Participation in collective depends on whether a CPU owns data in that collective
- **Nonblocking, overlapped**

Cholesky factorization update with broadcasts

```

#pragma processors CHOL (TX, TY)
shared [B][B] (CHOL) double Ah[B*TX][B*TY];
shared [B][B] (CHOL) double Bh[B*TX][B*TY];
shared [B][B] (CHOL) double A[N][N];

void update_mb (shared [B][B] double A[N][N],
                int N, int B, int col0, int col1, int TX, int TY) {

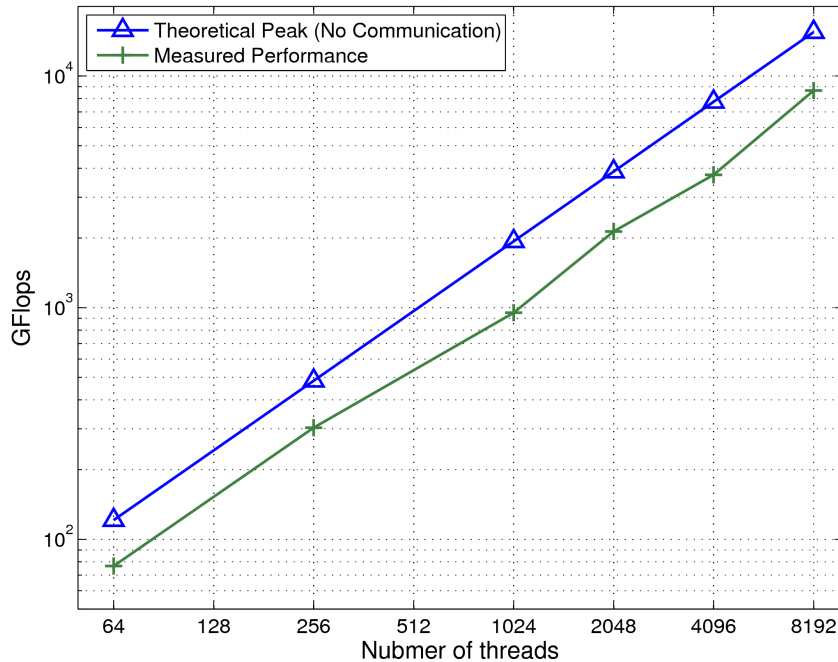
    int tx = MYTHREAD/TY, ty = MYTHREAD%TY;
    for (ii=col1; ii<N; ii+=B; continue)
        upc_stride_bcast (&A[ii][col0], &Ah, <<tx><:>>);
        for (jj=col1; jj<ii+B; jj+=B; &A[ii][jj]) {
            upc_stride_bcast (&A[jj][col0], &Bh, <<:><ty>>);

            if (upc_threadof(A[ii][jj]) != MYTHREAD) continue;
            double alpha = -1.0, beta = 1.0;
            int m = min(N-ii,B);
            int n = min(N-jj,B);
            int p = col1-col0;
            dgemm ("T", "N", &n, &m, &p, &alpha, &Bh[B*tx][B*ty], &B,
                &Ah[B*tx][B*ty], &B, &beta, (void *)&A[ii][jj], &B);
        }
    upc_barrier;
}

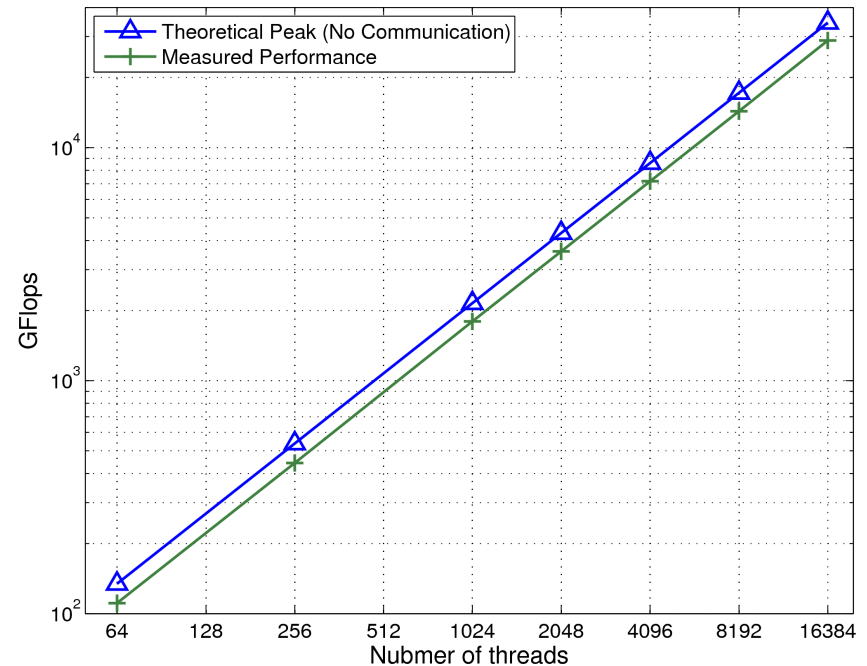
```

Performance with collective communication

Cholesky factorization



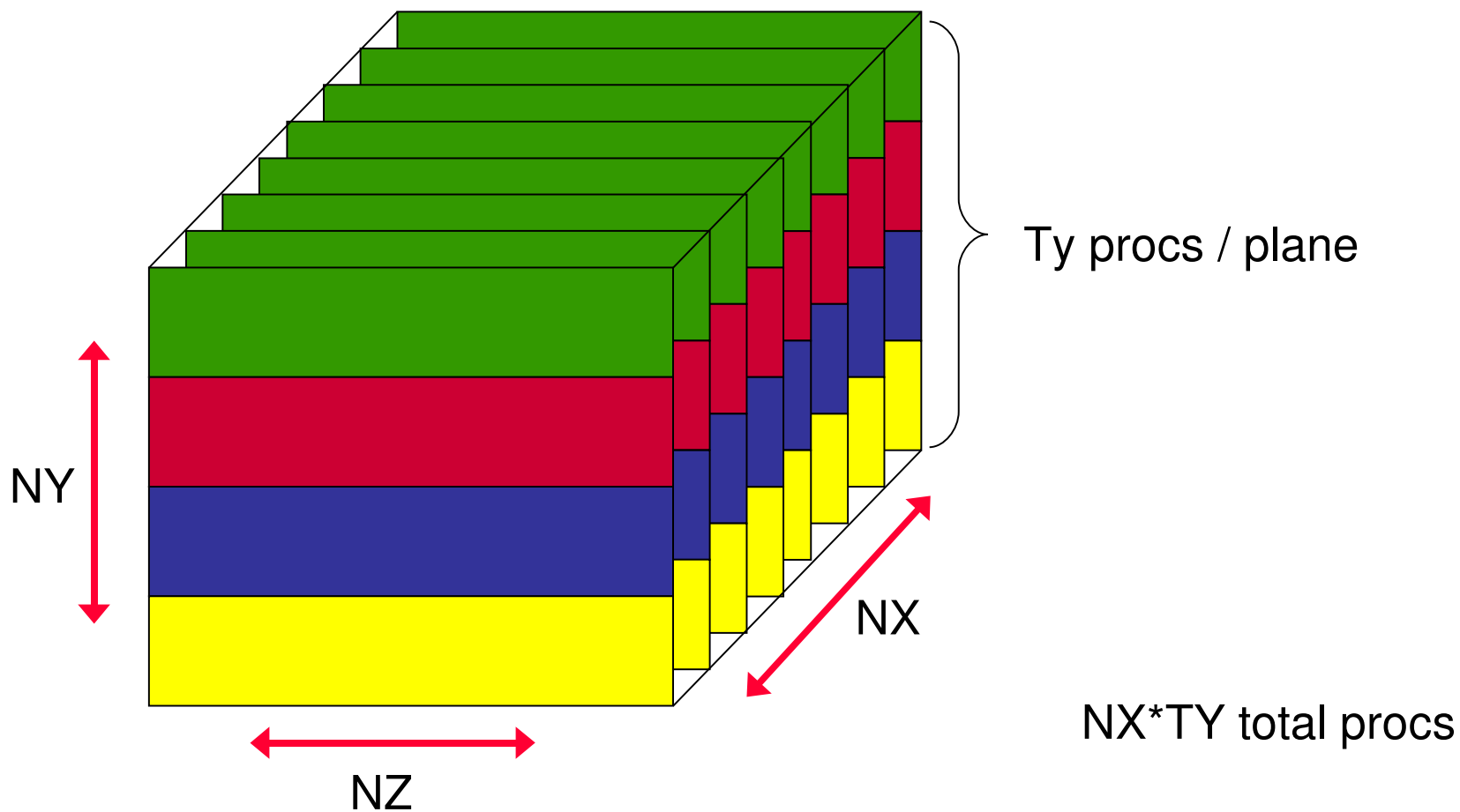
Matrix multiplication



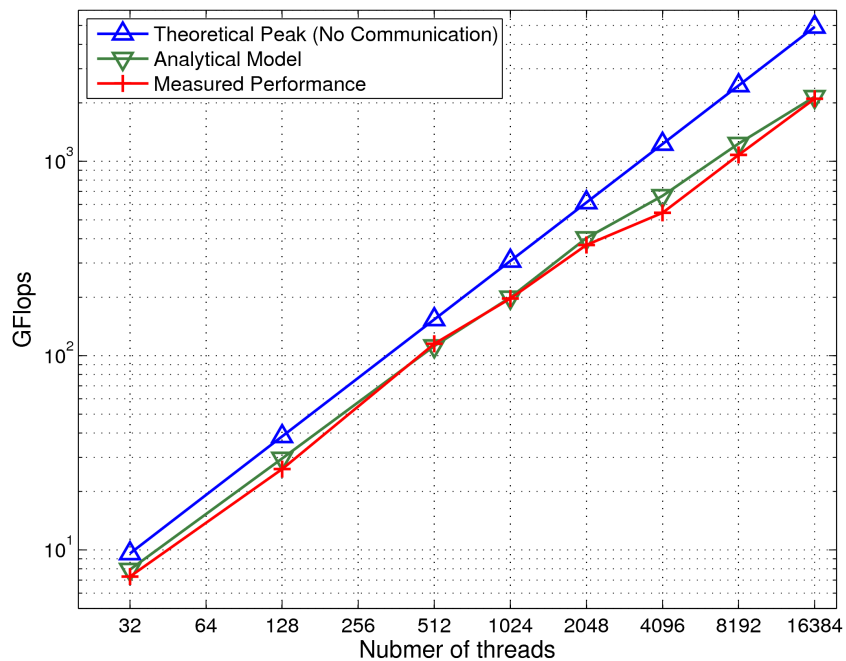
Performance loss:

15% serial ESSL performance; 20% communication; 15% load imbalance

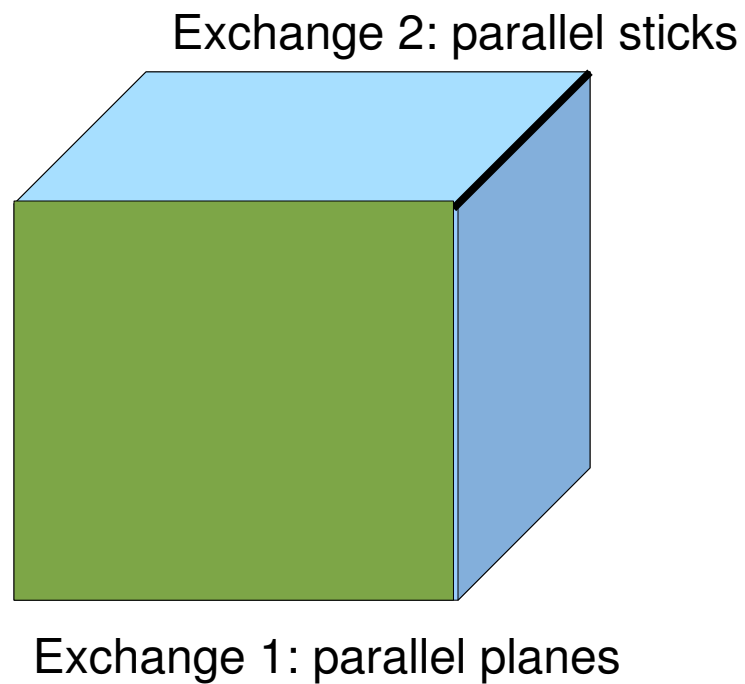
3D FFT using strided exchange collectives



3D FFT scaling



- **Analytical model takes bisection bandwidth into account**



Conclusions

- **So far so good:**
 - Good parallel linear algebra routine implementations do not have to be complicated.

- **Immediate goal:**
 - Scalable, high-performance, performance portable HPL in ~200 lines of code. Is that too much to ask?

- **UPC language extensions**
 - Can we get them adopted? Is anyone going to use them?