PGAS Languages (Partitioned Global Address Space)

Marc Snir

PARALLEL@ILLINOIS

www.parallel.illinois.edu

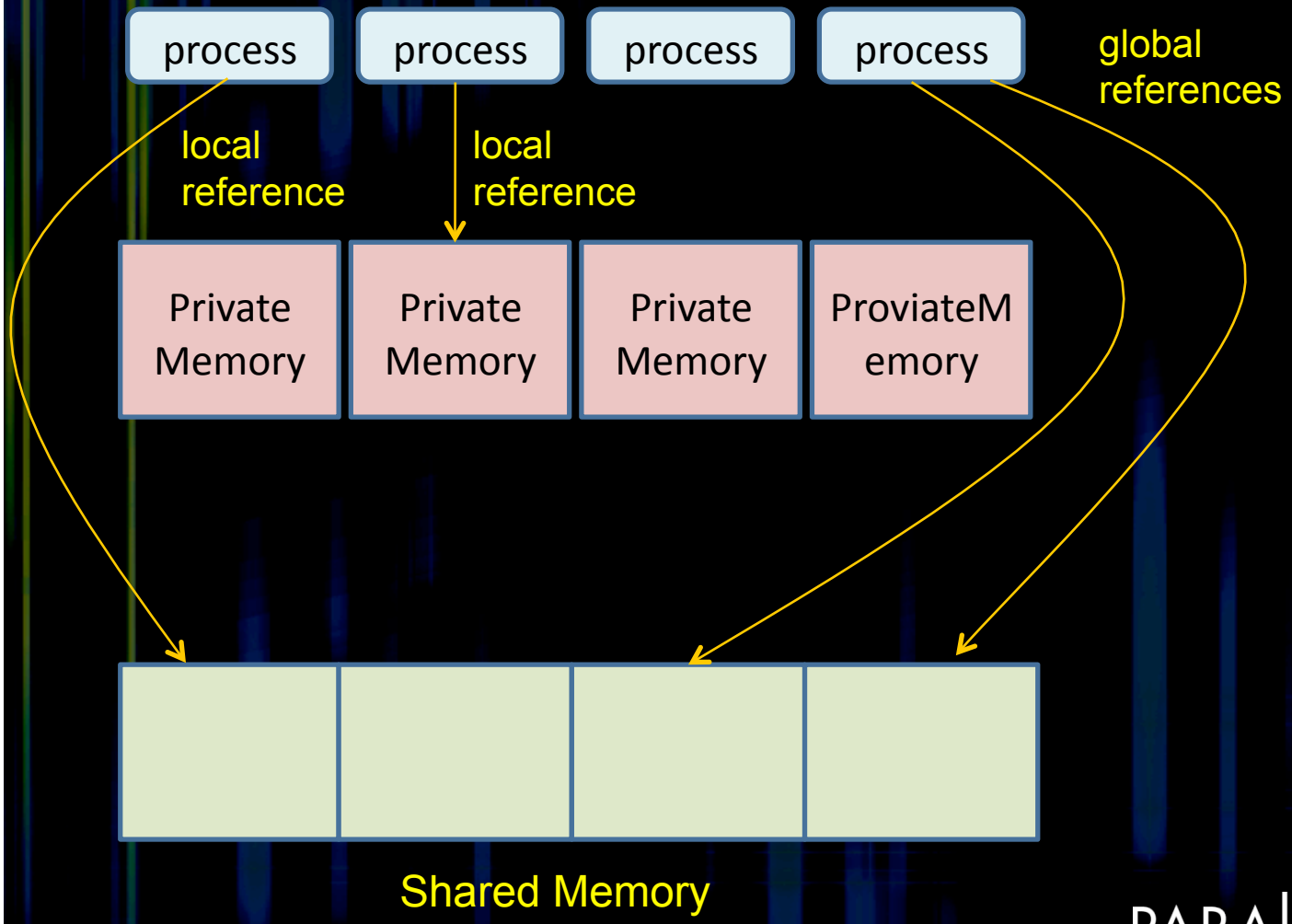
Goal

- Global address space is more convenient to users: OpenMP programs are simpler than MPI programs
- Languages such as OpenMP do not provide mechanisms to control locality; implementations on top of distributed memory are very inefficient
  - each load/store can become a get/put (or send/receive)
  - PGAS: design for providing global shared memory while controlling locality.

Design Principles

- Local references are syntactically distinct from global references (local and global pointers)
  - No overhead for use of local variables
- Good support for owner computer (computer where data is)
  - forall constructs

PGAS Memory Model



Unified Parallel C (UPC)

- UPC V1.0 was published in 2001; last standard is V1.2. May 2005
  - Developed by consortium under strong influence of govt. agencies.
  - Both public domain and commercial implementations available

UPC Memory Model

- Variables are private, by default

```
int i, x[100]
```

  - each process has an instance that only it can access

```
shared int j, y[100]
```

  - only one global instance that can be accessed by all processes

- References:

```
int *p;                  // private to private
shared int *q;           // private to shared
int *shared r;           // shared to private
shared int *shared s; // shared to shared
```

Where is a Shared Variable?

- Scalars are stored on process 0
- Arrays are stored round robin across all processes.

```
shared int i, j, x[10]
```

i j x[0] x[4] x[8]	x[1] x[5] x[9]	x[2] x[6]	x[3] x[7]

- Default layout is (almost always) bad!

Block-Cyclic Distribution

```
shared [2] int a[9];
shared [12/THREADS] int b[6][2];
```

a[0]	a[2]	a[4]	a[6]
a[1]	a[3]	a[5]	a[7]
a[8]			
a[9]			
b[0][0]	b[1][1]	b[3][0]	b[4][1]
b[0][1]	b[2][0]	b[3][1]	b[5][0]
b[1][0]	b[2][1]	b[4][0]	b[5][1]

- Only one-dimensional distributions
  - Proposal for multidimensional distributions is being discussed

Shared Pointer Arithmetic

p: [phase, thread, addr]

```
shared [3] int a[10];
```

a[0]	a[3]	a[6]	a[9]
a[1]	a[4]	a[7]	a[10]
a[2]	a[5]	a[8]	

p →

```
upc_phaseof(p) = 1
upc_threadof(p) = 0
upc_addrfield(p)
```

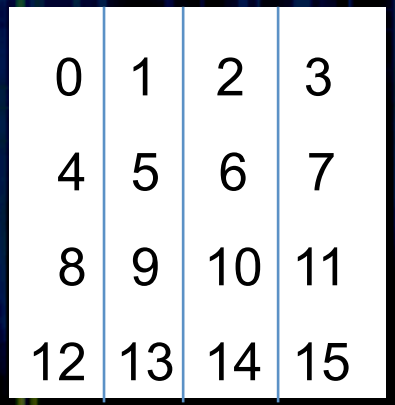
p++

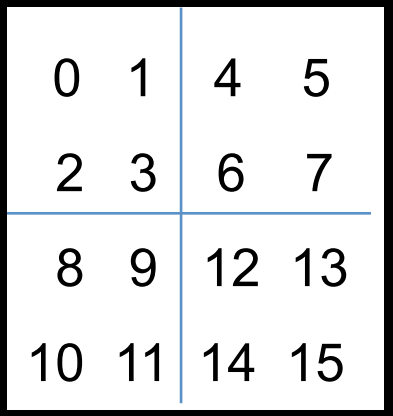
```
if (phase<blocksize-1) {
   phase++; addr+=sizeof(int);
   }
elseif (thread<NUMTHEADS) {
   thread++;  addr -= blocksize*sizoeof(int); phase=0;
   }
else {
   thread=0; addr += sizeof(int); phase=0;
   }
```

Jacobi Example

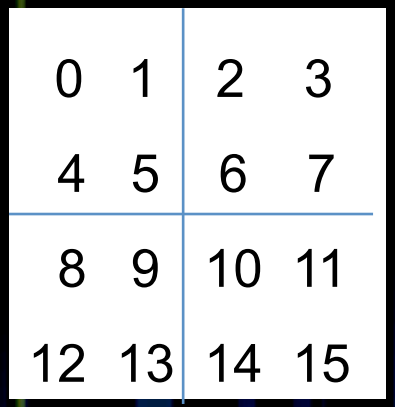
```
...
  for (i=1; i<=N; i++)
       for (j=1; j<=N; j++)
            b[i][j] = (4.0*a[i][j]+a[i-1][j]
                 +a[i+1][j]+a[i][j-1]
                 +a[i][j+1])*0.125;
...
```

Jacobi in UPC





- 1D partition
  - more communication



- 2D partition
  - not supported by UPC

- Use block-major numbering
  - more index arithmetic

  (more done by user, rather than compiler)

Jacobi Code

- Assume N+2 multiple of THREADS

```
…
shared [(N+2)*(N+2)/THREADS] double                    a[N
+2][N+2],b[N+2][N+2];
…
LB= (MYTHREAD == 0)? 1 : MYTHREAD*(N+2)/THREADS;
UB= (MYTHREAD == NTHREADS-1)? N :                  (MYTHREAD
+1)*(N+2)/THREADS-1;
for(i=LB; i<=UB; i++)
    for(j=1; j<=N; j++)
       b[i][j] = (4*a[i][j]+a[i-1][j]+a[i+1][j]+
            a[i][j-1]+a[i,j+1])*0.125;
upc_barrier;
…
```

- Each thread executes the code (SPMD model)

Jacobi Code -- Forall

```
…
shared [(N+2)*(N+2)/THREADS] double
      a[N+2][N+2],b[N+2][N+2];
…
upc_forall(i=1;i<N+1;i++;&a[i]);
   for(j=1;j<N+1;j++)
        b[i][j] = (4*a[i][j]+a[i-1][j]+a[i+1][j]
+
            a[i][j-1]+a[i][j+1])*0.125;
upc_barrier;
...
```

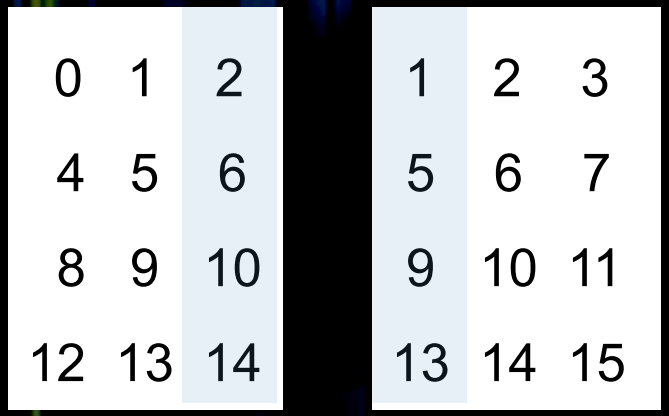
- ***Work sharing***: **each thread picks for execution values of i for which a[i] is local; call is *collective***

What the Compiler Should Do

```
upc_forall(i=1;i<N+1;i++;&a[i]);
   for(j=1;j<N+1;j++)
        b[i][j] = (4*a[i][j]+a[i-1][j]+a[i+1]
[j]+
             a[i][j-1]+a[i][j+1])*0.125;
```

– move the selection of local iterates out of loop (if possible)
–replace global references with local references (user can do this explicitly with casting)
– aggregate communication (if possible)
–allocate ghost cells (if possible)

What Compiler Should Do (2)



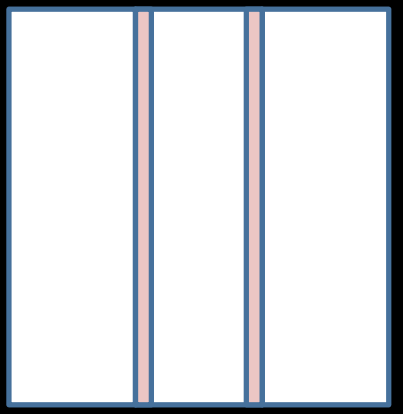
- Compute nested loop on local slice using local indices
- Copy boundary column to ghost column at neighbor
- Not always that simple...

```
upc_forall(i=1;i<N+1;i++;&a[i]);
  for(j=1;j<N+1;j++)
      b[i][j] = (4*a[i][j]+a[i-1][j]+a[i+1][j]+
           a[i][j-1]+a[i][j+1])*0.125;
```

Communication/Computation Overlap

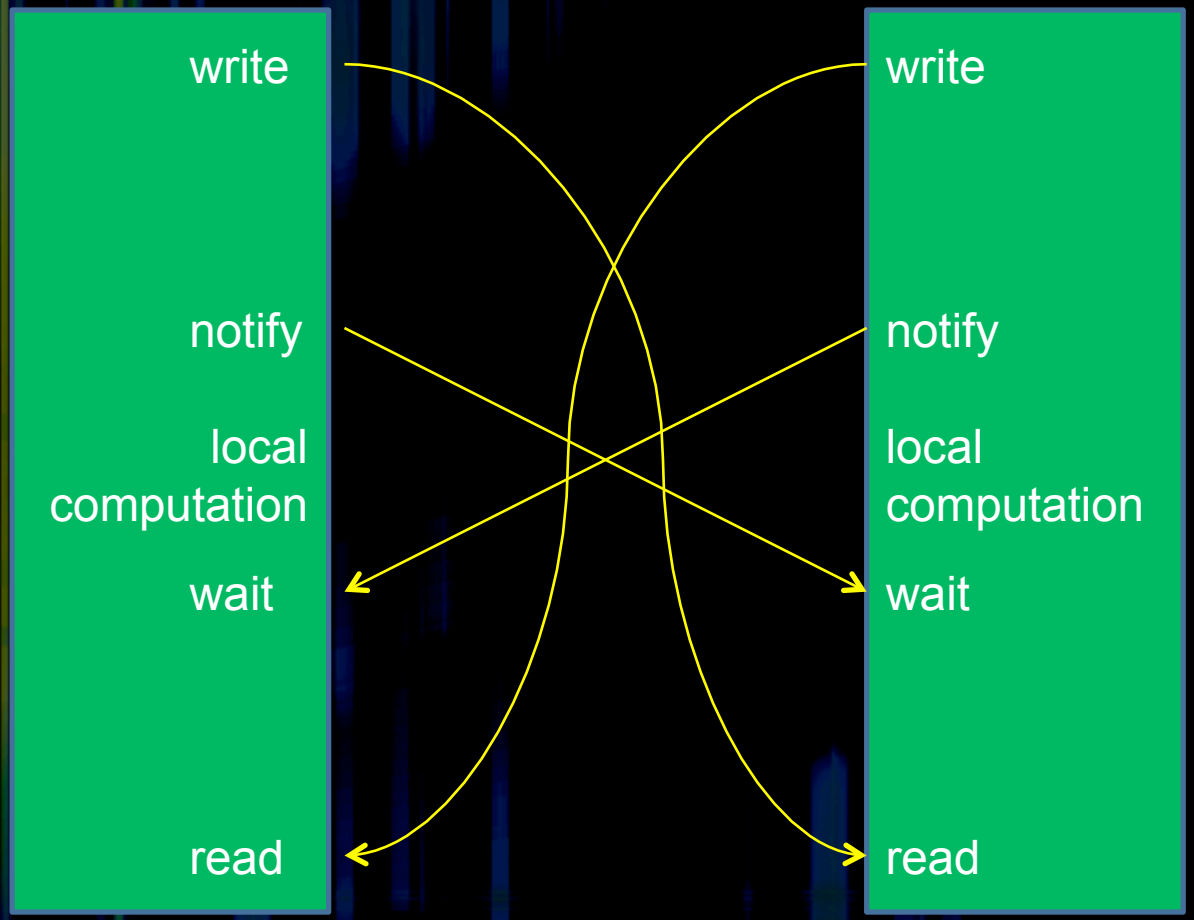
- **Split barrier – the logical equivalent of nonblocking communication**

```
…
shared [(N+2*(N+2)/THREADS] double a[N+2][N+2],b[N+2][N+2];
…
/* compute boundary */
if (MYTHREAD > 1) {
i = MYTHREAD*(N+2)/THREADS;
  for(j=1; j<N+1; j++)
      b[i][j] = …; }
if (MYTHREAD < THREADS) {
i = (MYTHREAD+1)*(N+2)/THREADS-1;
  for(j=1; j<N+1; j++)
      b[i][j] = …; }
upc_notify;  // done with my contribution
/* compute interior */
for(i=MYTHREAD*(N+2)/THREADS+1; i<(MYTHREAD+1)*(N+2)/THREADS-1;
i++)
  for(j=1; j<N+1; j++)
    b[i][j]=…
upc_wait; //wait for all other notifications
```



Split Barrier

- More variance tolerant



Dynamic Data Structures C

- Linked List code C

```
struct {
    char *data;
    node *next;
    } node;

void insert(char *a, node *after) {
 node *p;
 p = (*node)malloc(sizeof(node));
 p->next = after->next;
 p->data = a;
 after->next = p;
}
```

Dynamic Data Structures UPC

```
struct {
    char *data;
    shared node *shared next;
    } node;

void insert(char *a, shared node *shared after)
 {
 shared node *shared p;
 if (MYTHREAD==upc_threadof((node*)after)){
   p = (*shared node) upc_alloc(sizeof(node));
   p->next=after->next;
   p->data=a;
   after->next= p;
   }
upc_barrier;
 }
```

- Insertion executed only by thread that owns "after"
- Shared memory allocated at threads performing insertion

Dynamic Data Structures (3)

- OK to keep private pointer in node, if node is always manipulated by local thread

```
struct {
    char *data;
    shared node *next;
    } node;

void insert(char *a, shared node *shared after)
 {
 node *p;
 if (MYTHREAD==upc_threadof((node*)after)) {
   p = (*node) upc_alloc(sizeof(node));
   p->next=after->next;
   p->data=a;
   after->next=p;
   }
upc_barrier;
 }
```

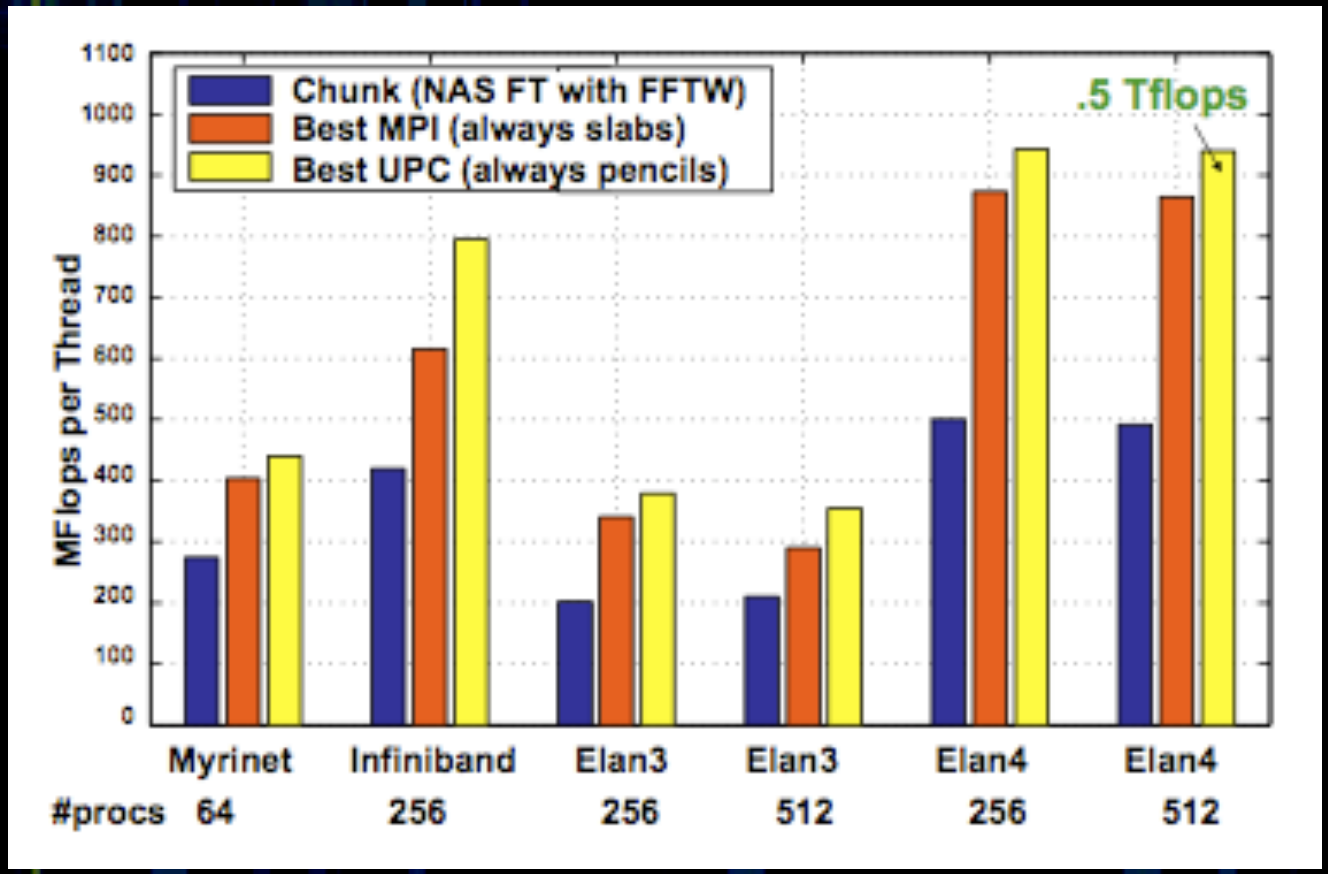
What Else in UPC?

- Locks
- Collective communication: broadcast, scatter, gather, allgather, exchange, permute, reduction, scan
- Strict and relaxed memory model

Missing Features

- Teams -- subsets of threads to be used as support for distributed arrays, and as target for work sharing and synchronization constructs
  - essential for modularity (e.g. multi physics code)
- All2all collective
- Multidimensional distributions (?)
- C++ support

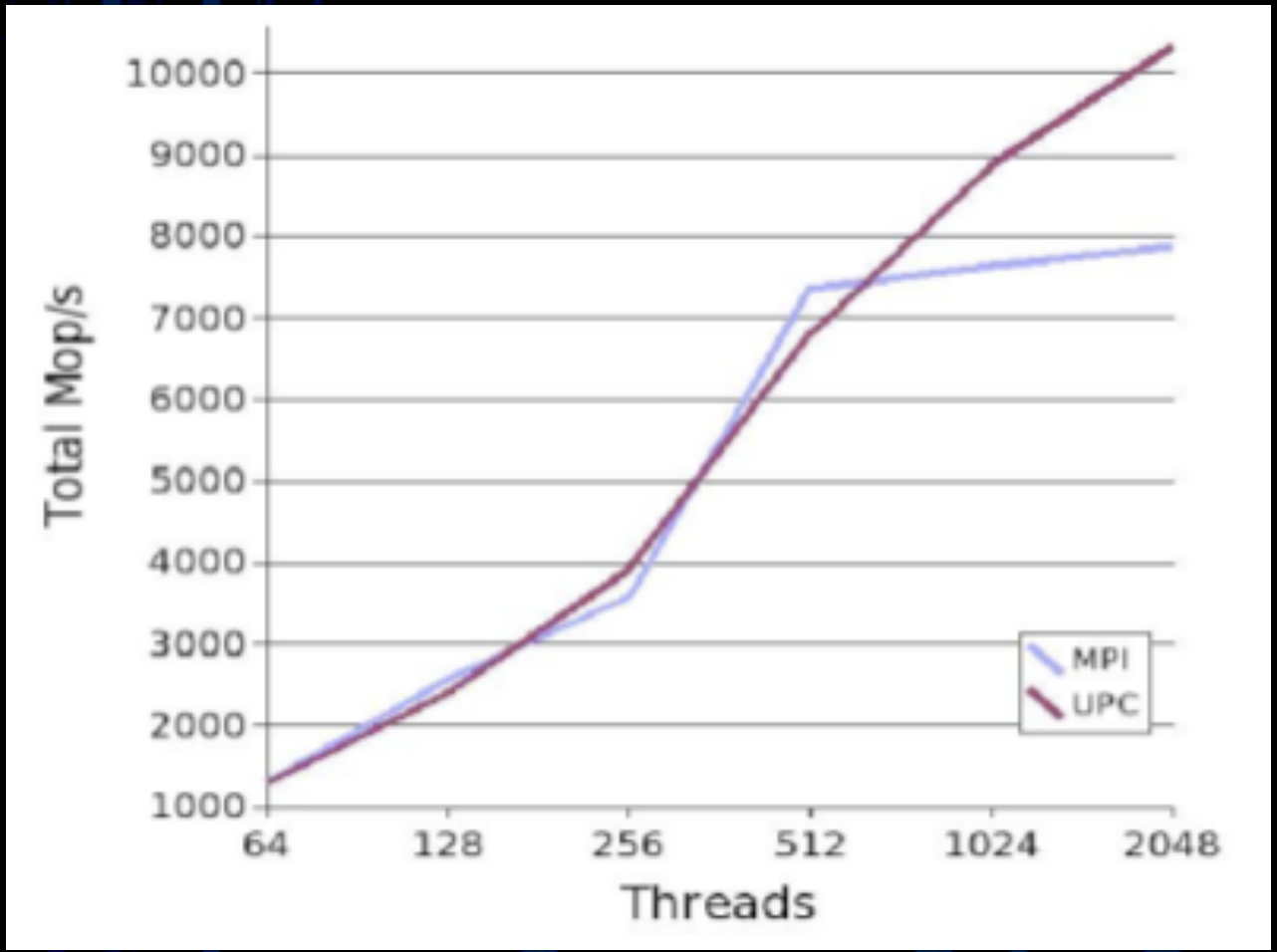
What Performance Do We Get? (1)



3D-FFT (Yellick et al., Berkeley compiler)

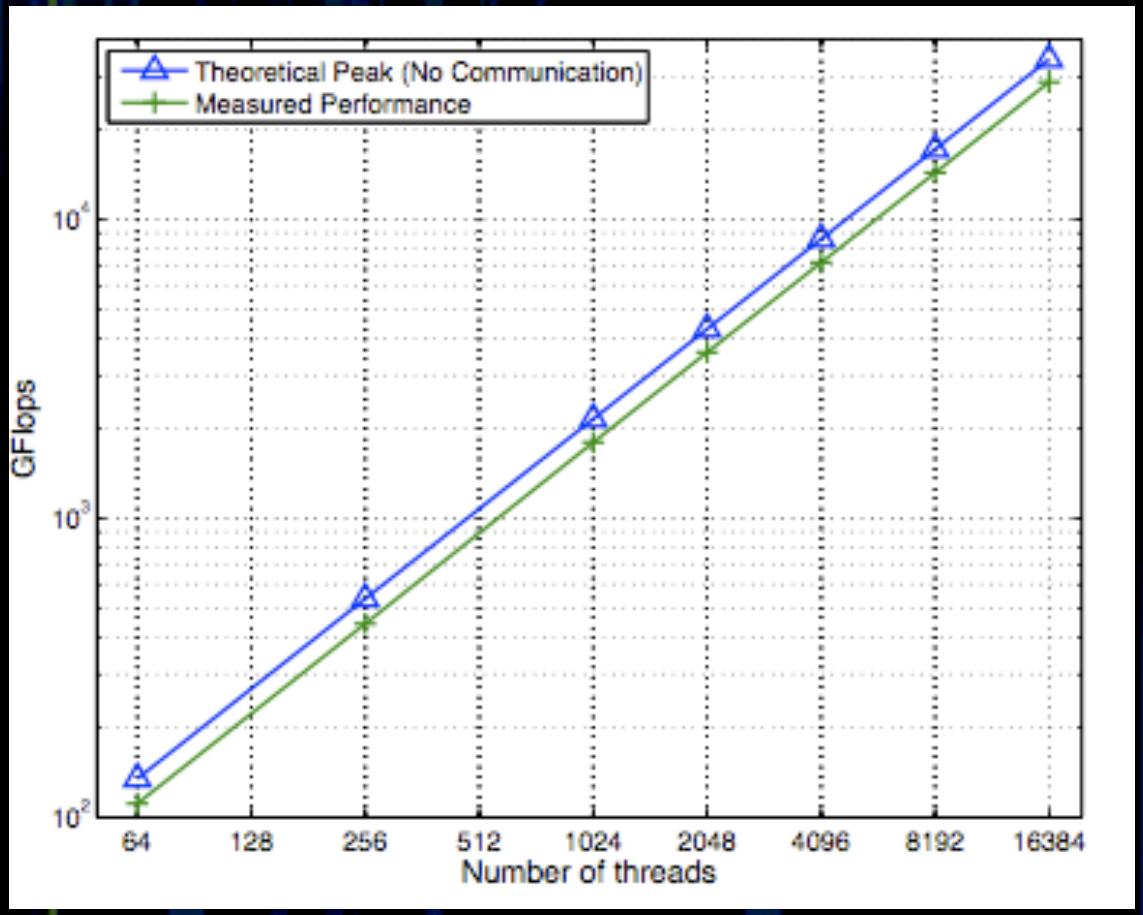
PARALLEL@ILLINOIS

What Performance Do We Get? (2)



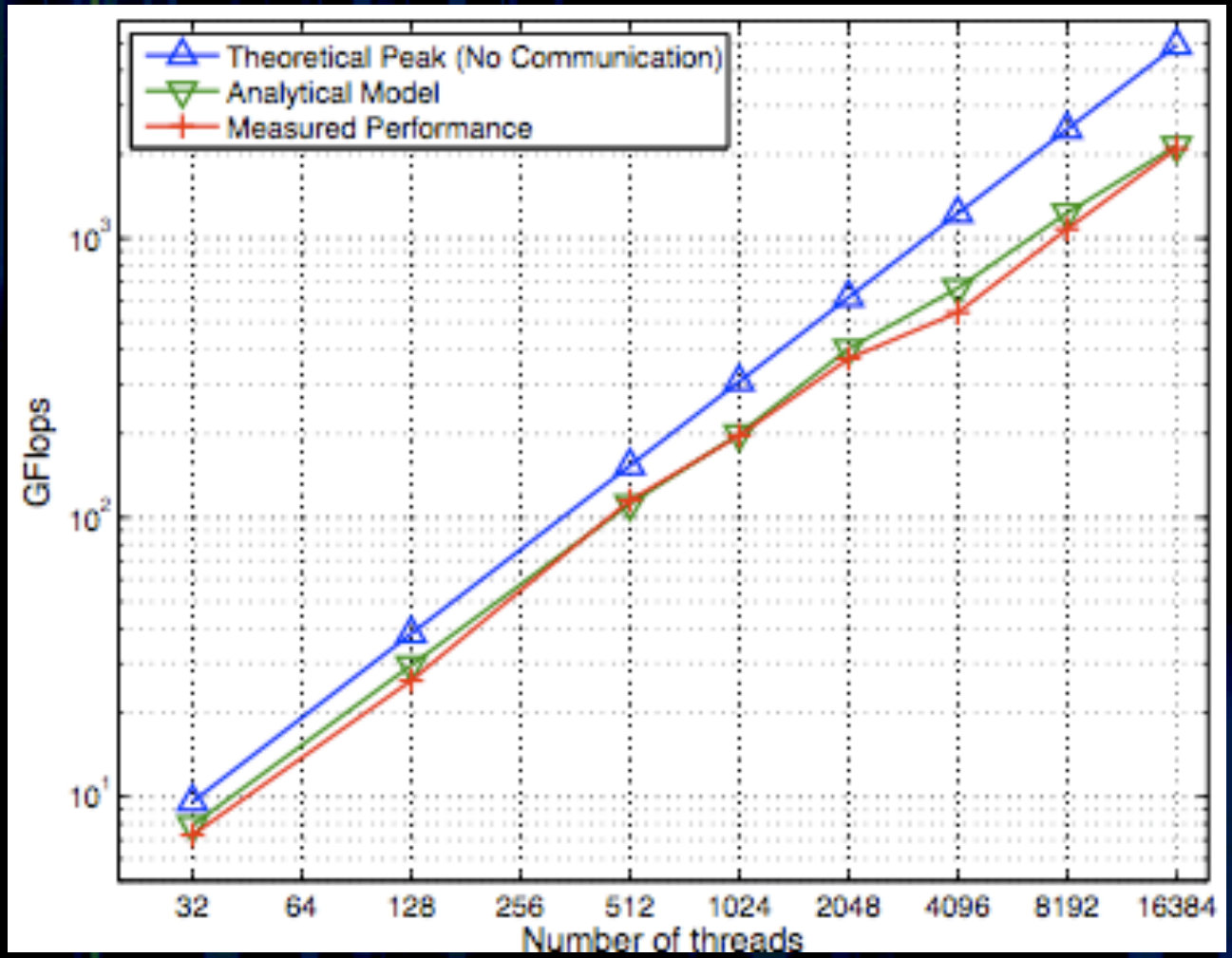
NAS Benchmark CG Class C (Barton et al., IBM compiler)

What Performance Do We Get? (3)



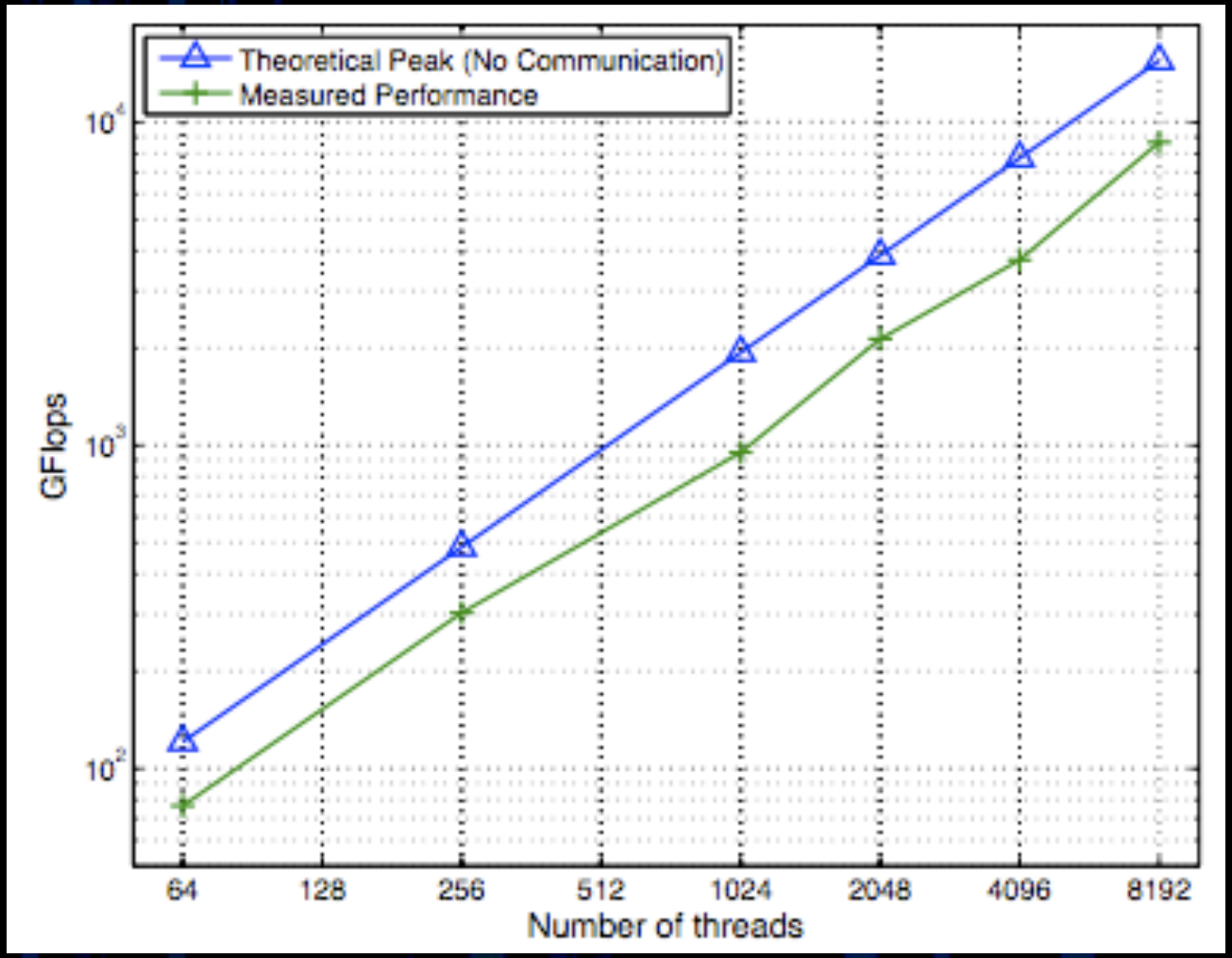
Matrix multiplication (Nishtala et al., IBM compiler)

What Performance Do We Get? (4)



3D-FFT (ibid)

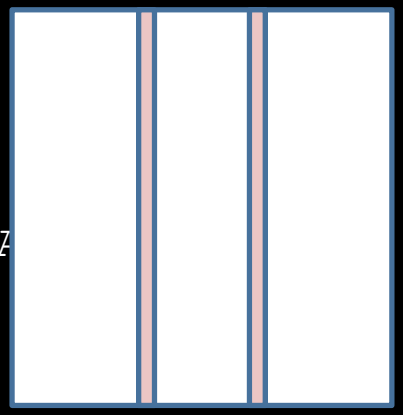
What Performance Do We Get? (5)



How Do We Get Performance?

- Communication aggregation: ensure that data is moved in large messages.
  - Aggregation by compiler
  - Aggregation by user: use memcopy to move data in big chunks

UPC "MPI Style"



```
…
shared [(N+2*THREADS)*(N+2)/THREADS] double a[N+2*THREA
+2*THREADS][N+2];
double *mya[][N+2]; double *myb[][N+2];
…
mya =(double*)a[MYTHREAD*(N/THREADS+2)];
myb =(double*)b[MYTHREAD*(N/THREADS+2)];
for(j=1;j<N+1;j++)
    (*myb)[1][j] = 0.125*(4*(*mya)[1][j]+(*mya)[0][j]+
          (*mya)[2][j]+(*mya)[1][j-1]+(*mya)[1][j+1]);
if (MYTHREAD > 0)
upc_memput(&(*myb)[i][1],&b[MYTHREAD*(N/THREADS+2)-1][1],
        N*sizeof(double));
for(j=1;j<N+1;j++)
    (*myb)[N-1,j] =…;
if (MYTHREAD < THREADS-1)
upc_memput(&m(*myb)[N-1][1],&b[(MYTHREAD+1)*(N/THREADS+2)][1],
        N*sizeof(double);
upc_notify;
for(i=2, i<(N+2)/THREADS; i++)
   for(j=1; j<N+1; j++)
       (*myb)[i][j]=…
upc_wait;
…
```

- With half-decent compiler this code should run as fast or faster than MPI
- Naïve code never runs fast, and can be surprisingly slow

Co-Array Fortran

- Developed by Numrich in 1998
- Supported by Cray and in an open source implementation
- Proposed for inclusion in Fortran 2008

- Same SPMD execution model as UPC
  - No shared work constructs
- Co-array (shared array): extra dimension(s)

CAF Co-Arrays

```
real, dimension(100,50)[*]:: x
```

- Allocates 5000 x *num_images* reals

```
real, dimension(-1:98)[3:7,5,*]:: y
```

- Allocates 100 x *num_images* reals; requires that number of processes is multiple of 25)

```
x(:,2)[2] = y(:)[4,2,1]
```

- Array assignment that involves communication

~~`x(1,3)[1:5] = y(0:4)[4,2,1]`~~

- Can gather, but cannot scatter

CAF "MPI Style"

```
…
real, dimensions (N+2,(N+2)/num_images())[*]:: a, b
…
b(2:N+1,2) = 0.125*(4*a(2:N+1,2)+a(2:N+1,1)+
      a(2:N+1,3)+a(1:N,2)+a(3:N+2,2))
if (this_image() > 1) then
  b(2:N+1,N+2)[this_image()-1] = b(2:N+1,2)
  NOTIFY(this_image()-1)
      end if
b(2:N+1,N+1) = 0.125*(4*a(2:N+1,N+1)+a(2:N+1,N)+
      a(2:N+1,N+2)+a(1:N,N+1)+a(3:N+2,N+1))
if (this_image()<num_images) then
b(2:N+1,1)[this_image()+1) = b(2:N+1,N+1)
  NOTIFY(this_image()+1)
      end if
b(2:N+1,3:N)=0.125*(4*a(2:N+1,3:N)+a(2:N+1,2:N-1)+
      a(2:N+1,4:N+1)+a(1:N,3:N)+a(3:N+2,3:N))
if (this_image()>1) then QUERY(this_image()-1) end if
if (this_image()<num_images) then QUERY(this_image()+1)
      end if
```

Titanium

- Developed by C. Yelick et al.
- Java based
  - No impediment for good performance with the right compiler!
- Many interesting features (e.g., good support for sparse arrays)

Migration from MPI to PGAS

- PGAS languages have same execution model as MPI (assuming no work-sharing used)
- Replace arrays with shared arrays (allocated independently on each thread)
- Replace send/receive with memcopy to/from shared arrays
- Use local references for local code
- If code is well-structured, should affect only small fraction of code
- Possible gain: better performance for fine-grain communication
  - Compiler can map directly to communication HW
  - Compiler can perform optimizations across multiple communications

PGAS Summary

- Languages still need to evolve
  - But are already in a useful state
- Do not really provide a higher-level programming level
- Can provide lower communication overhead
- Can better hide the shared-memory/distributed-memory distinction – well-suited to handle multi-core
- Main additions needed:
  - Teams, Collectives, Virtualization, *Good implementations*