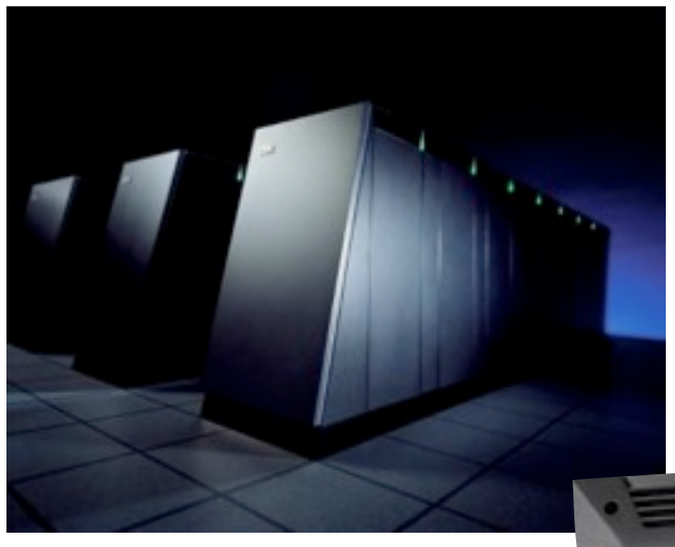
PGAS:
Partitioned Global Address Space

Nicolás Cardozo (ncardozo@vub.ac.be)
Software Languages Lab - Vrije Universiteit Brussel

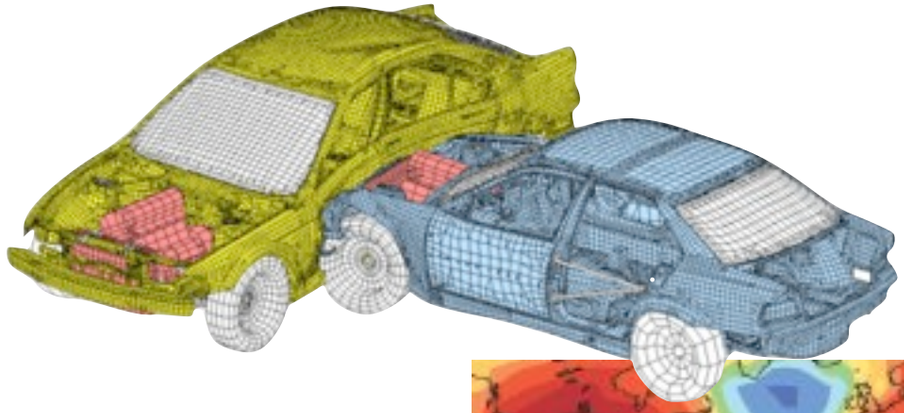
Context

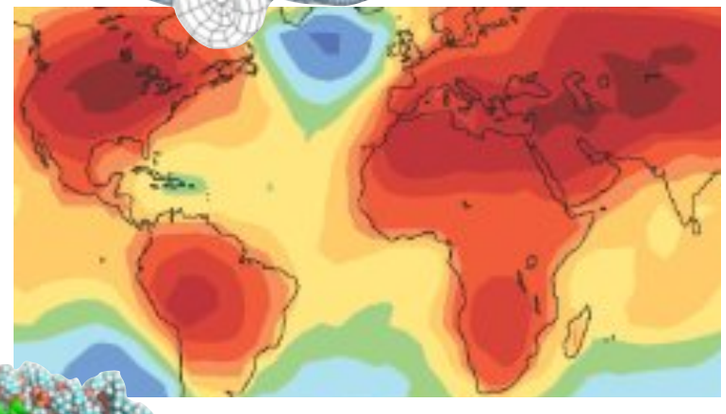
High-Performance Computing

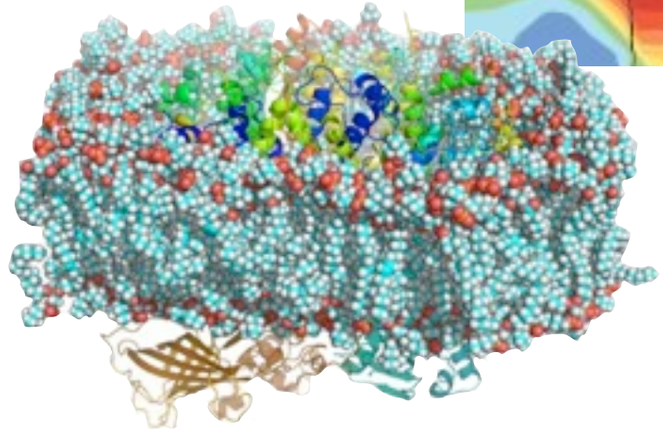
Supercomputers



Applications







Outline

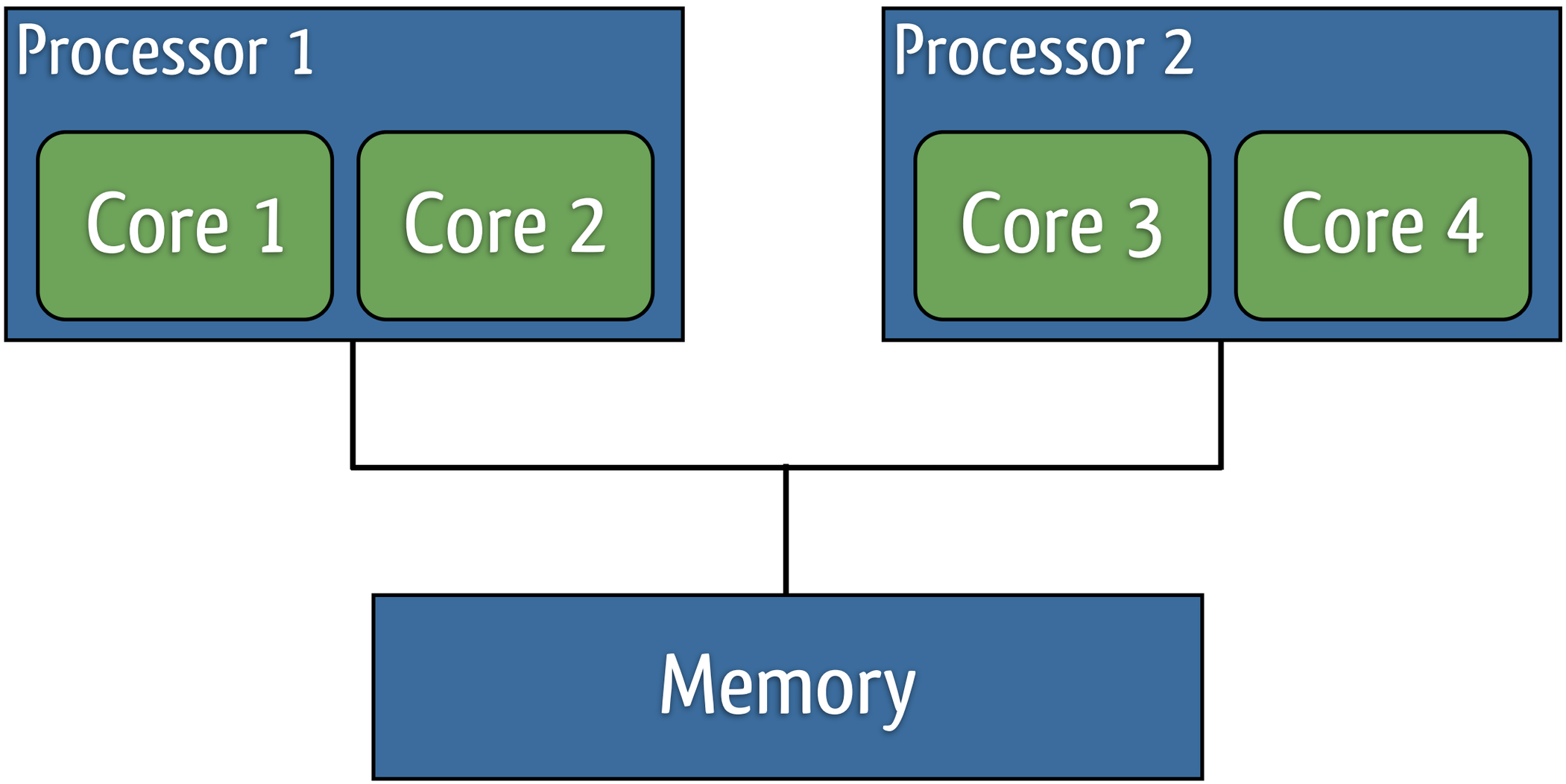
- Parallel Computer Architectures
- Parallel Programming Models
- Concrete (PGAS) approaches:
  - MPI
  - UPC
  - Global Arrays

Motivation: Parallel Computer Architectures

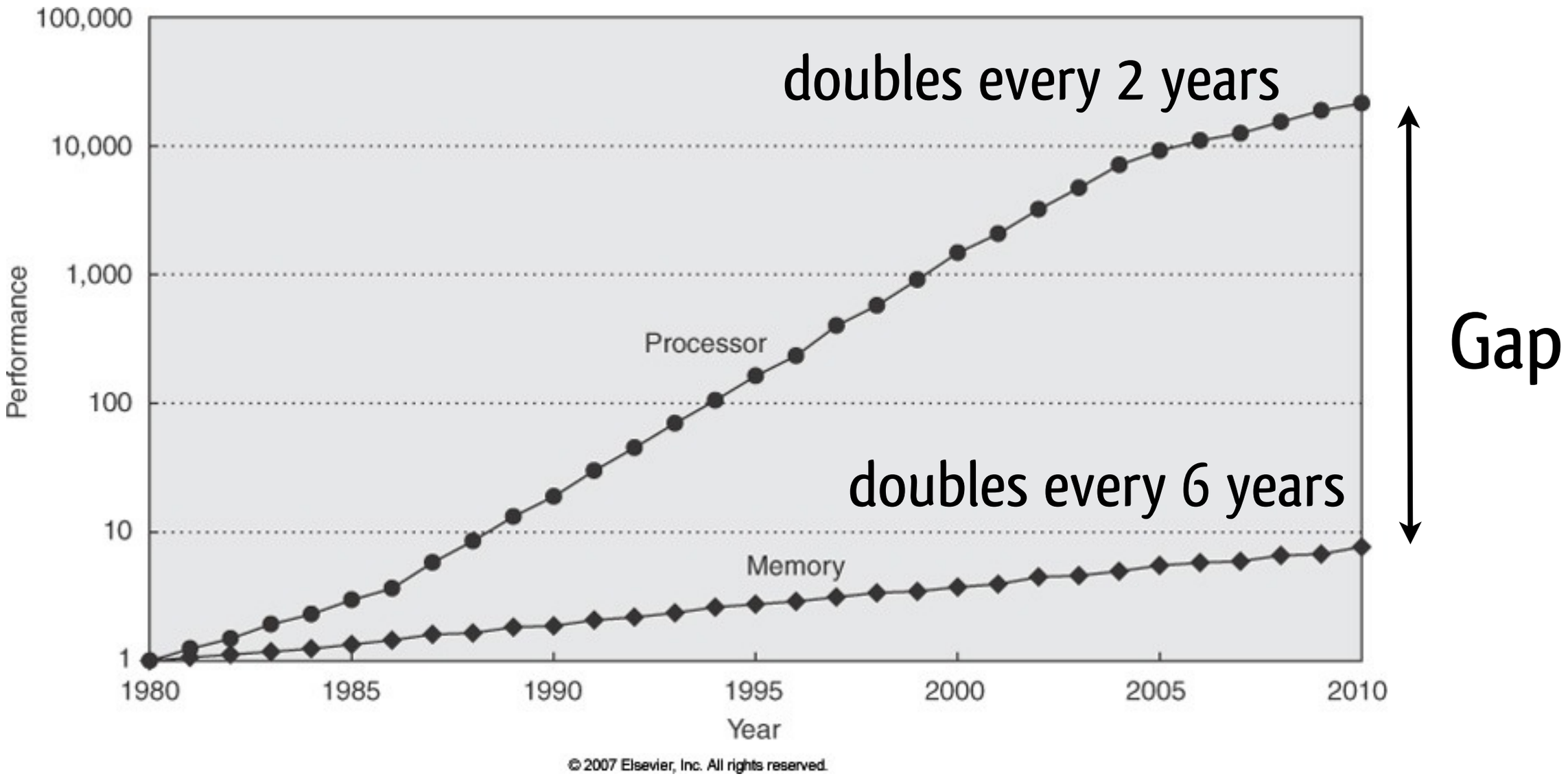
Parallel speedup $S = \frac{T_1}{T_P}$

- Suppose that I execute a program using P processing cores, what speedup can I expect?
- Amdahl's law: theoretical speedup is limited by the sequential fraction of the program
- E.g.: If 5% of the program is serial, the parallel runtime will always be at least 5% of the serial runtime
  - Maximal speedup factor: 20x $P = \frac{T_1}{T_\infty}$
- However, even with sufficient parallelism, there are many reasons for non-linear speedup in computer architectures!

Symmetric Multiprocessing

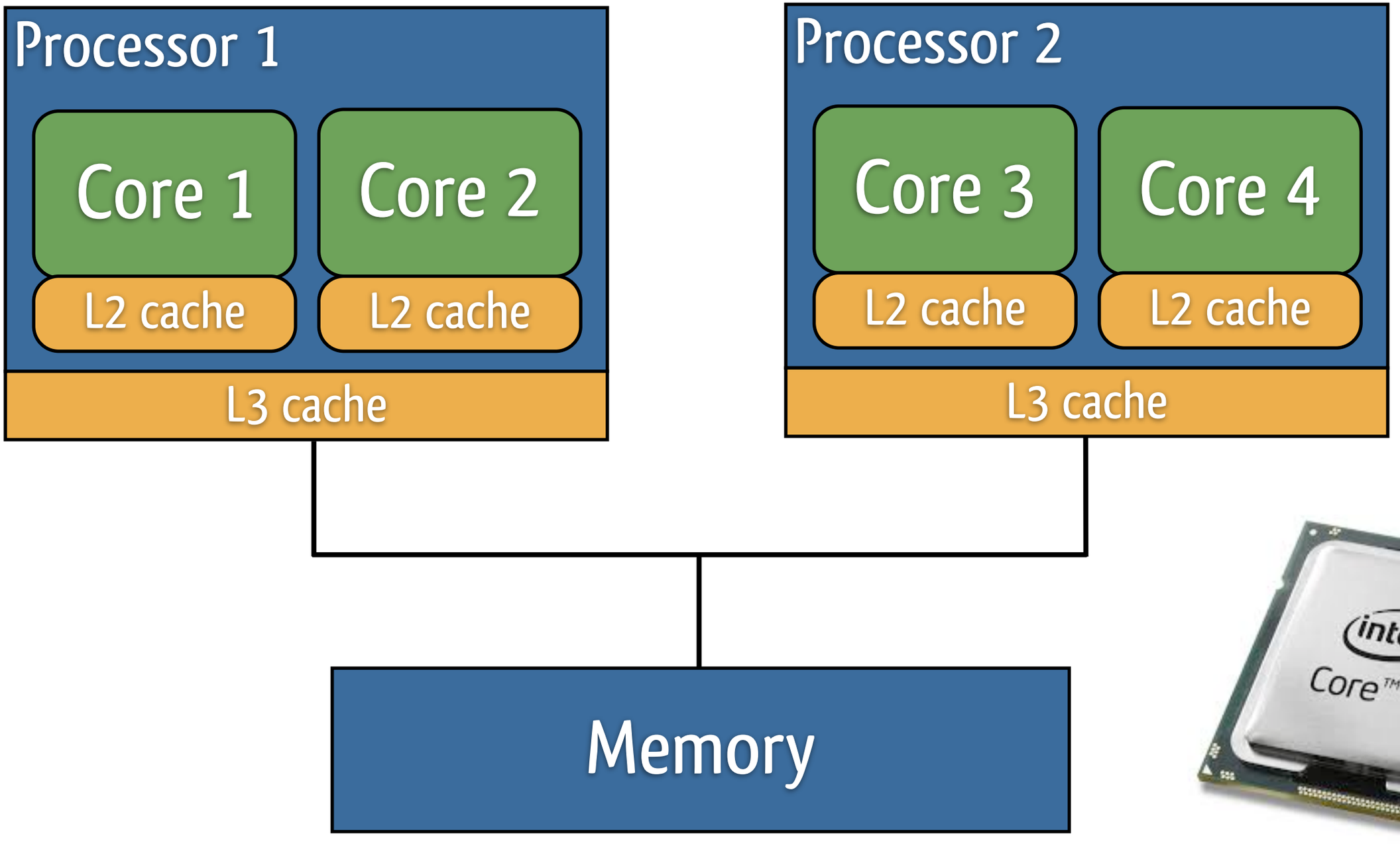


Processor-Memory Gap



Memory-intensive computations don't scale!

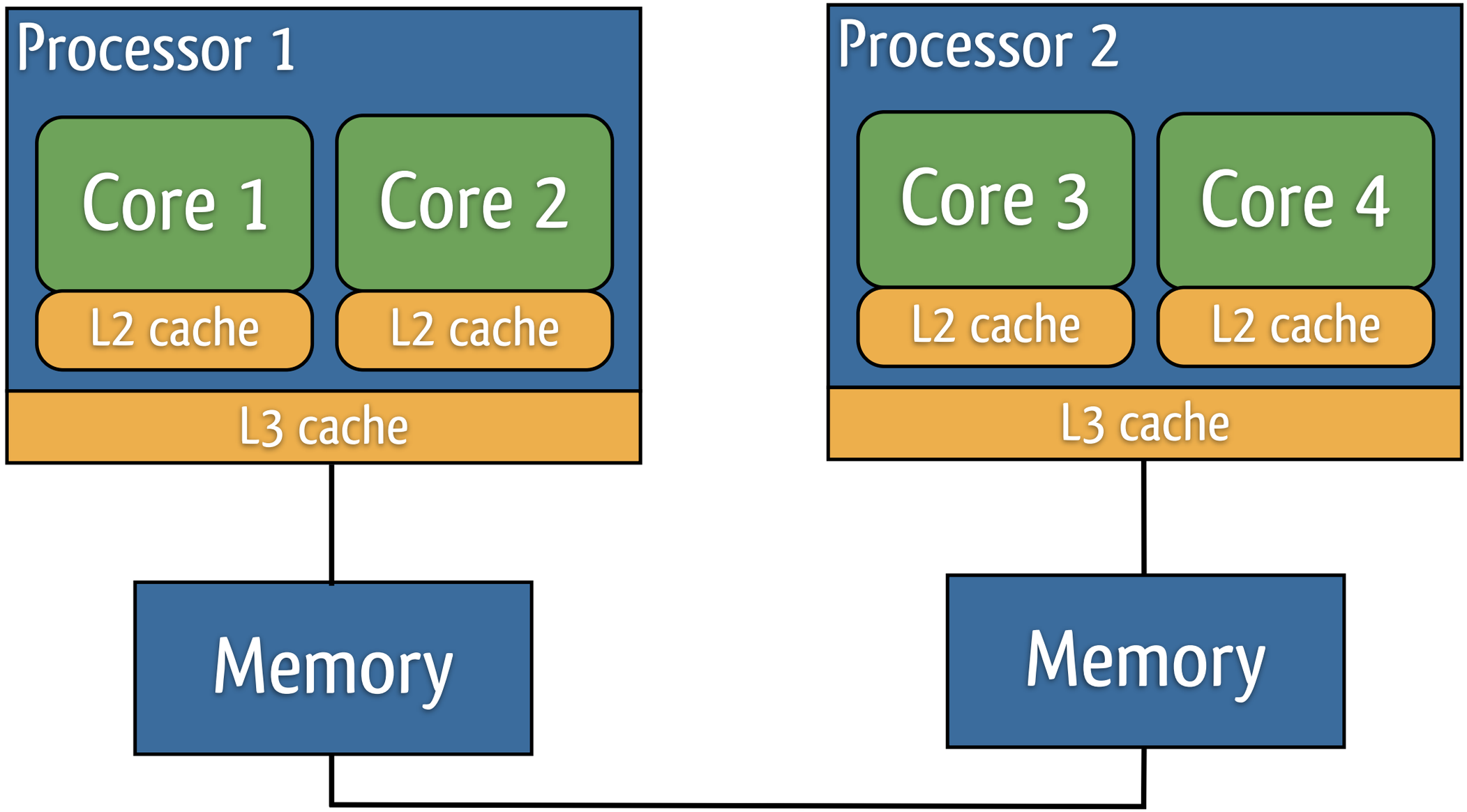
Solution: Caches



Caches

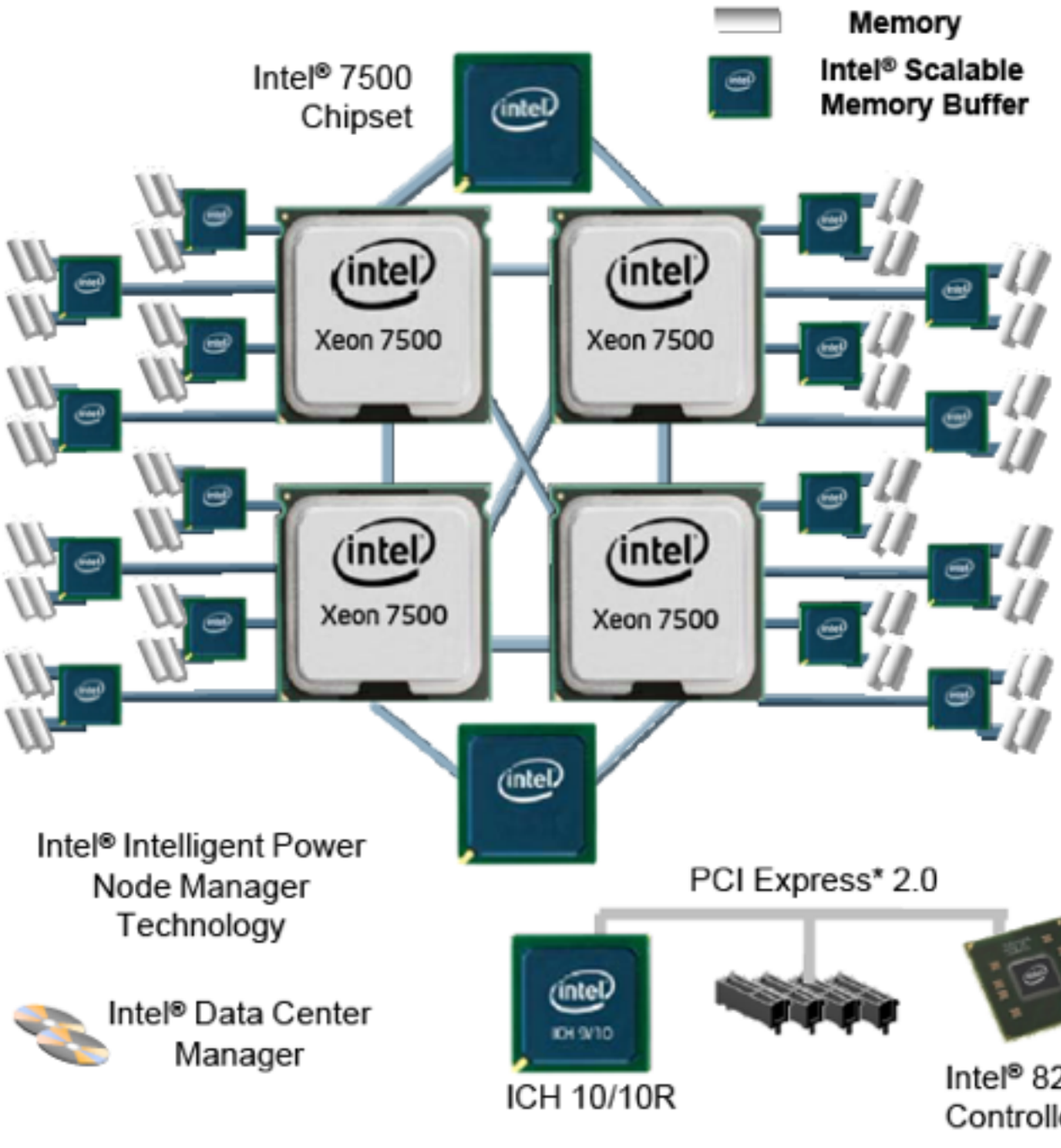
- Caches hide memory latency, but are much smaller
- Cache coherency: when multiple caches contain copies of same memory locations
  - Has additional costs

Non-Uniform Memory

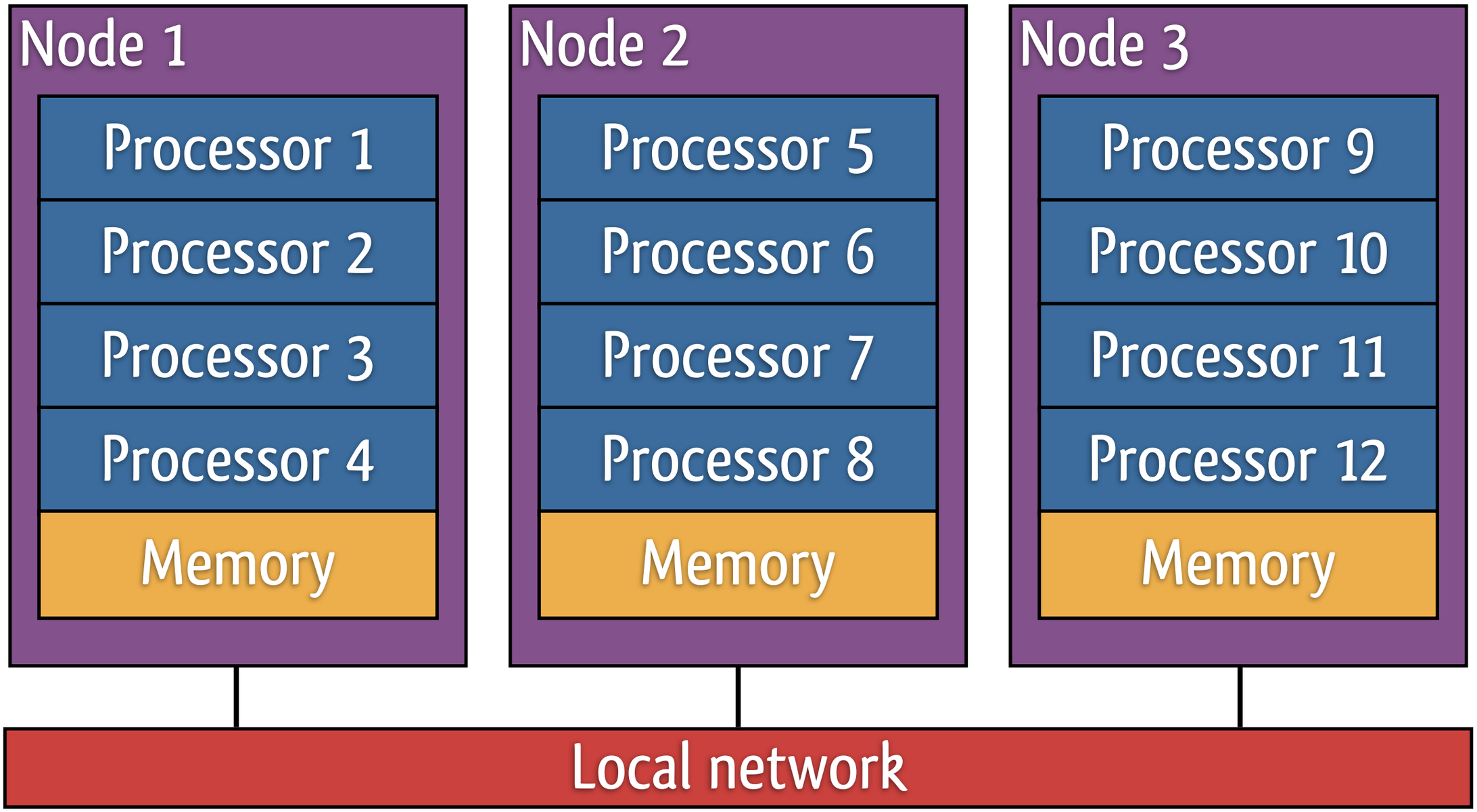


Global address space, different access times

Non-Uniform Memory



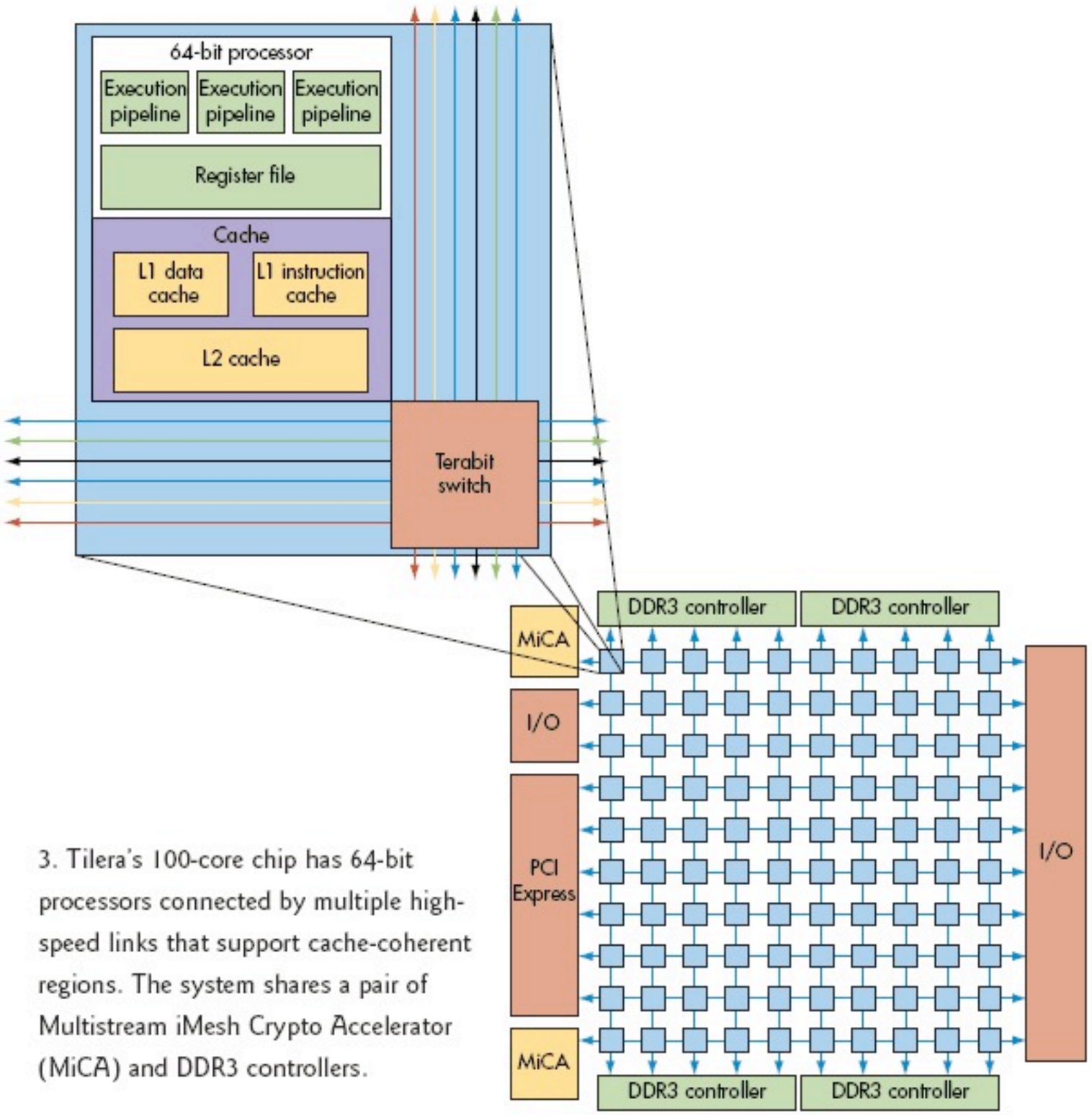
Cluster



Multiple local address spaces

Cluster

Integrated Many-Core



3. Tilera's 100-core chip has 64-bit processors connected by multiple high-speed links that support cache-coherent regions. The system shares a pair of Multistream iMesh Crypto Accelerator (MiCA) and DDR3 controllers.

Conclusion

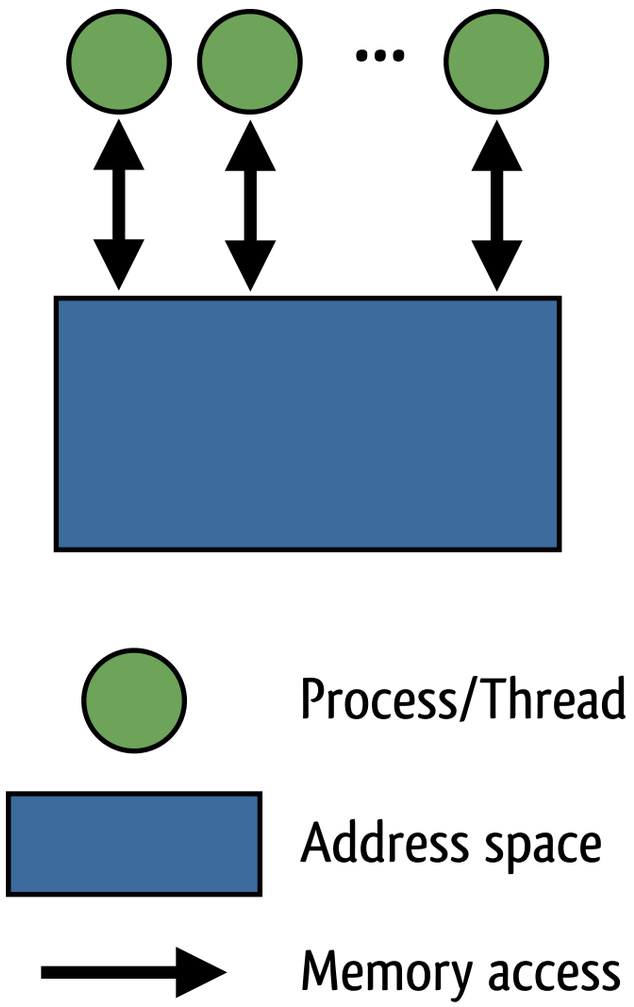
Irrespective of the architecture,
data locality is extremely important!

Parallel Programming Models

Programming Models

- Why programming models?
  - Decouple applications and architectures
  - Write applications that run effectively across architectures
- Examples
  - Shared memory (Global Address Space)
  - Message Passing
  - Partitioned Global Address Space (PGAS)

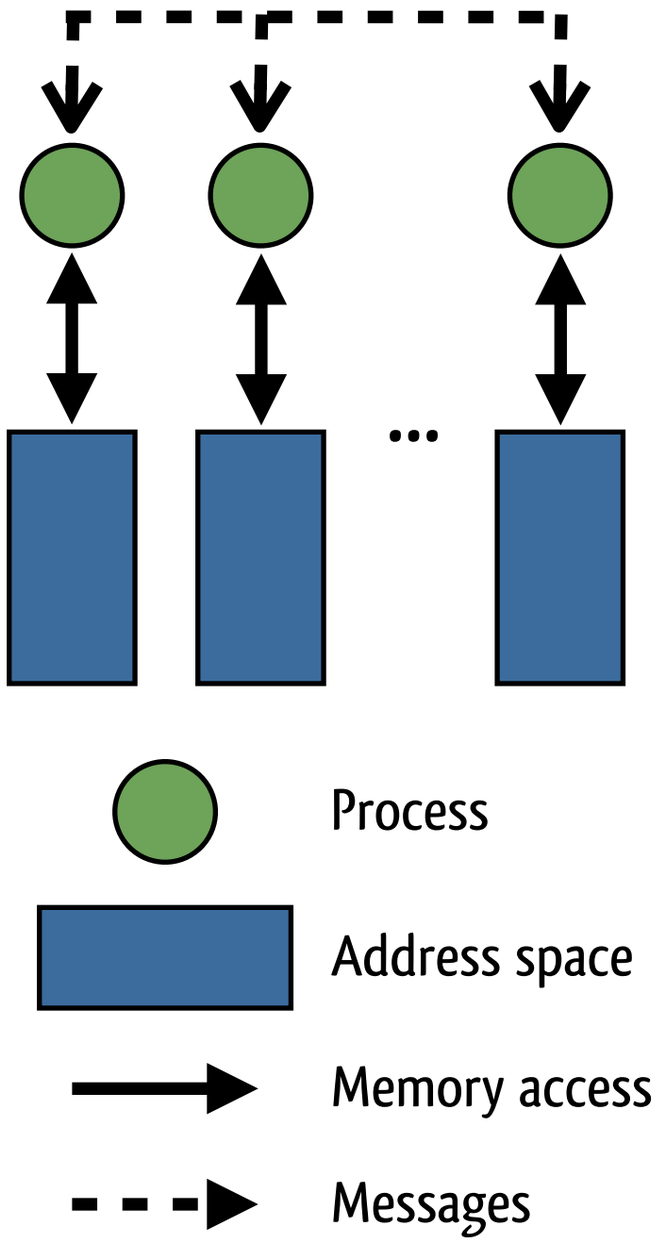
Shared Memory



- Concurrent threads with shared space
- Advantages:
  - Simple programming statements
- Disadvantages:
  - Requires synchronization (locks)
  - No notion of locality
  - Doesn't work on clusters

Examples: pthreads, Java threads, OpenMP

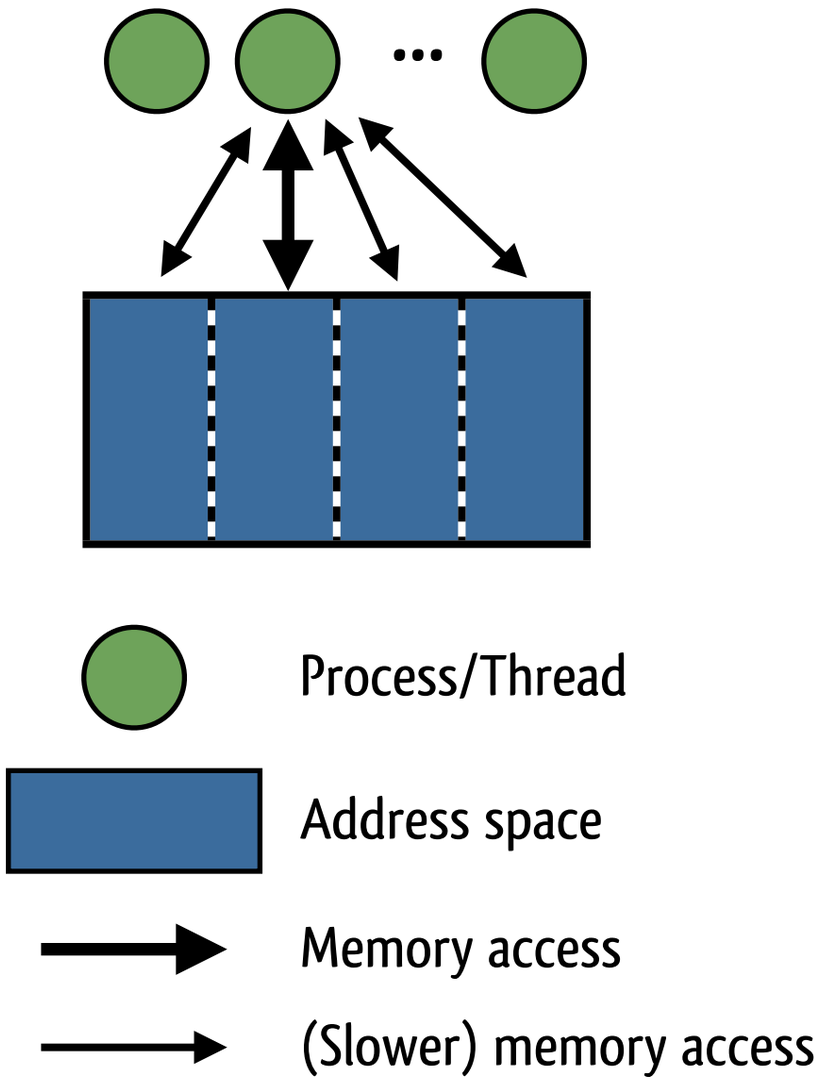
Message Passing



- Concurrent sequential processes, explicit communication
- Advantages:
  - Full control over data and work distribution
  - Scales to clusters
  - Implicit synchronization
- Disadvantages:
  - Hard to program

Examples: MPI, (actors), (CSP)

Partitioned Shared Memory



- Concurrent threads with partitioned shared space
- Advantages:
  - Simple programming statements
  - Allows to exploit locality
  - Scales to clusters
- Disadvantages:
  - Requires synchronization

Examples: UPC, Co-array Fortran, Chapel, X10, Titanium

Message Passing Interface (MPI)

MPI

- Message passing library specification
- Concrete API for C, C++, Fortran
- MPI-1 (1994), MPI-2 (1996), MPI-3 (future)
- Several implementations: OpenMPI, mpich, Intel MPI,...
- De facto standard for cluster programming

MPI: Hello World

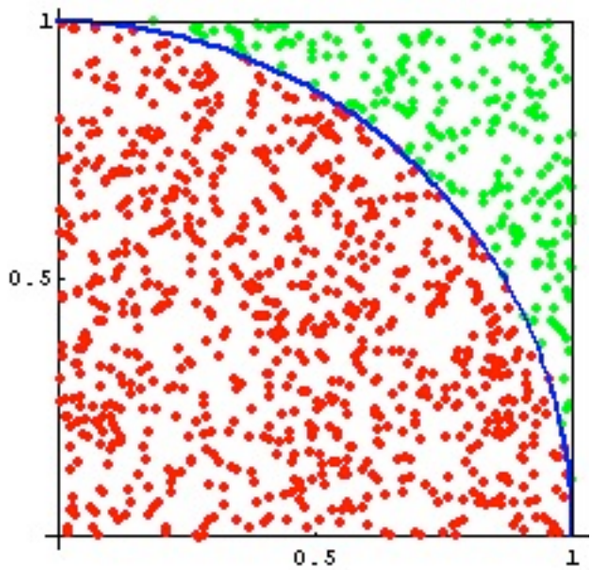
```c
#include <mpi.h>
#include <stdio.h>

int main(int argc, char **argv) {
    MPI_Init(&argc, &argv);

    int rank, size;
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    printf("Hello world from %d of %d\n",
           rank, size);

    MPI_Finalize();
    return 0;
}
```

- Single program, multiple data (SPMD)
- MPI_Init, MPI_Finalize
- MPI_COMM_WORLD has every process
- Size = # processes
- Rank in [0..size)

Output:

```
$ mpicc -O2 -Wall -std=c99 -o mpi-hello mpi-hello.c
$ mpirun -np 4 ./mpi-hello
Hello world from 0 of 4
Hello world from 1 of 4
Hello world from 3 of 4
Hello world from 2 of 4
```

Monte Carlo Method for π



$$\frac{\#(\text{quadrant})}{\#(\text{square})} \approx \frac{A_{\text{quadrant}}}{A_{\text{square}}} = \frac{\frac{1}{4}\pi\, 1^2}{1^2} = \frac{\pi}{4}$$

- Can be calculated in extremely parallel manner
- Generate random square points (uniformly distributed)
- Count points inside quadrant $\sqrt{x^2 + y^2} < 1$
- 4x hit ratio approximates π

Monte Carlo: auxiliaries

```
unsigned seed;

// Seed random number generator with current time
seed = (unsigned)time(NULL);

// Generate pseudo-random number in [0,1)
double urand() {
    return rand_r(&seed) * (1.0 / RAND_MAX);
}

// Check wether (x,y) is inside circle
int is_hit(double x, double y) {
    return sqrt(x * x + y * y) < 1.0;
}
```

MPI Monte Carlo

```
MPI_Send(sendbuf, count, type, dest, tag, comm)
MPI_Recv(recvbuf, count, type, src, tag, comm, status)
```

```
#define DOTS (500*1000*1000)

// Calculate local contribution
long hits = 0;
for(long k = 0;
    if(is_hit(u
        hits++;


if(rank != 0) {
    // Send cor
    MPI_Send(&h
} else {
    // Add up e
    long totalh
    for(long k
        MPI_Rec
        totalhi
    }
    double ratio = (double)totalhits / (double)DOTS;
    printf("pi = %.12f\n", ratio*4);
}
```

Output:

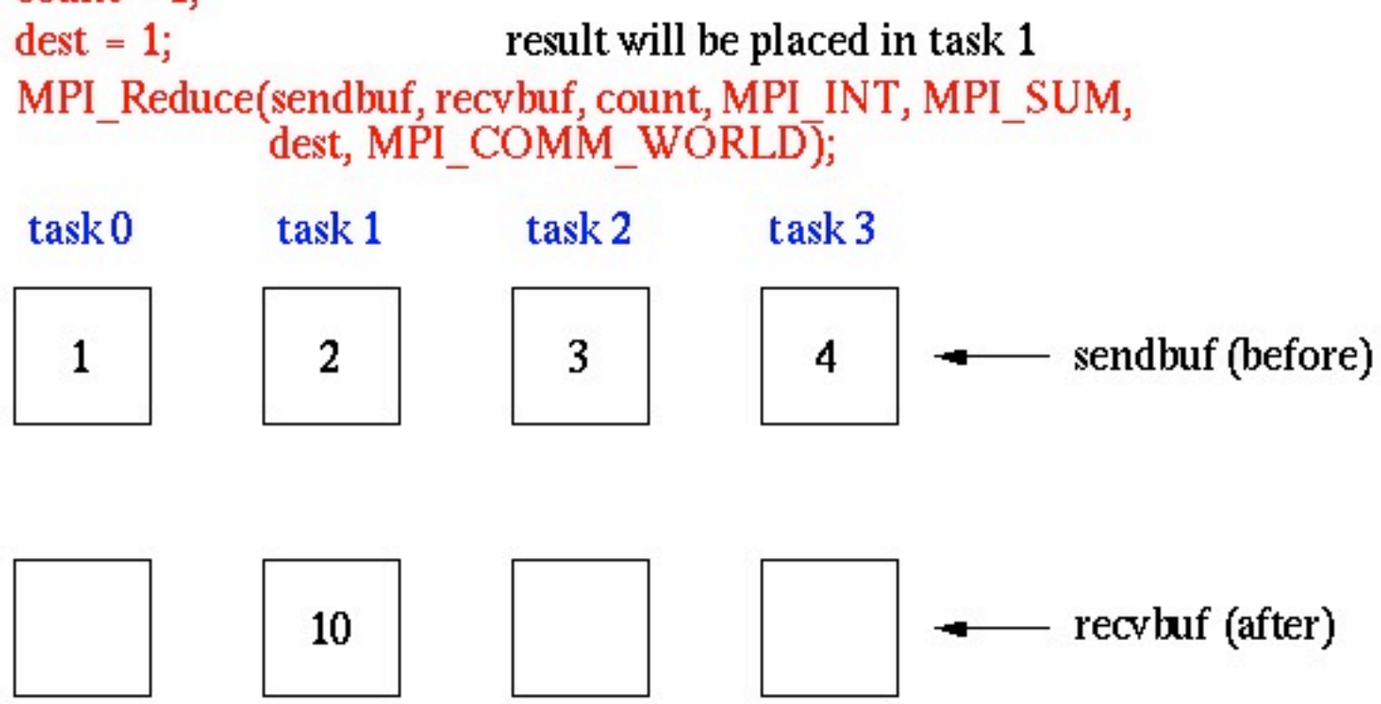
```
$ mpicc -O2 -Wall -std=c99 -o mpi-montecarlo mpi-montecarlo.c
$ mpirun -np 8 ./mpi-montecarlo
pi = 3.141274944000

$ mpirun -np 4 ./mpi-montecarlo
pi = 3.141461216000

$ mpirun -np 32 ./mpi-montecarlo
pi = 3.141771264000

$ mpirun -np 2 ./mpi-montecarlo
pi = 3.141602848000
```

MPI Collective: Reduce

MPI_Reduce

Perform and associate reduction operation across all tasks in the group and place the result in one task

```
count = 1;
dest = 1;                          result will be placed in task 1
MPI_Reduce(sendbuf, recvbuf, count, MPI_INT, MPI_SUM,
                    dest, MPI_COMM_WORLD);
```



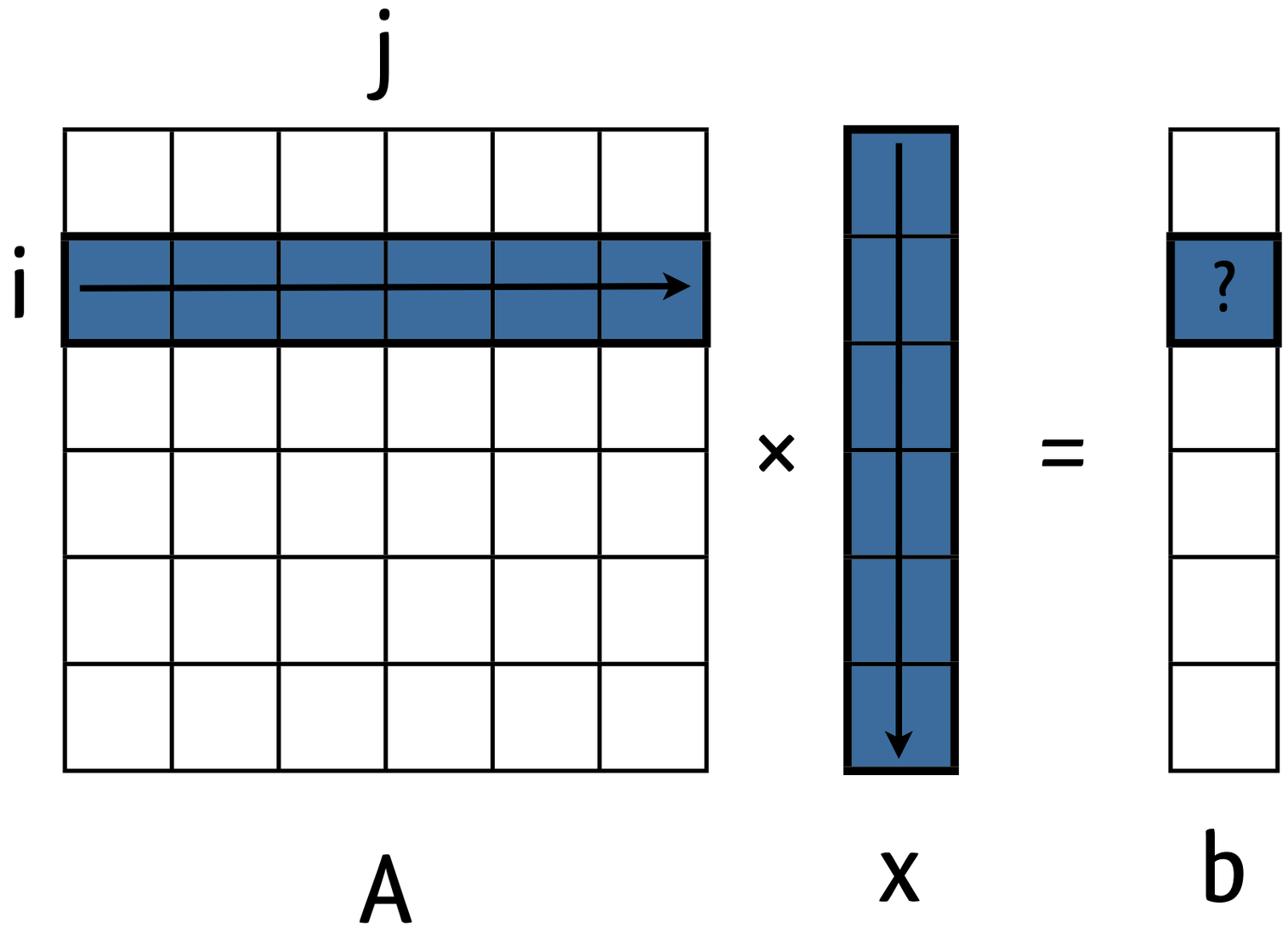
Reduction

```
if(rank != 0) {
    // Send contribution to root process
    MPI_Send(&hits, 1, MPI_LONG, 0, rank, MPI_COMM_WORLD);
} else {
    // Add up everyone's contribution
    long totalhits = hits;
    for(long k = 1; k < size; k++) {
        MPI_Recv(&hits, 1, MPI_LONG, k, k, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
        totalhits += hits;
    }
    double ratio = (double)totalhits / (double)DOTS;
    printf("pi = %.12f\n", ratio*4);
}
```

```
long totalhits;
MPI_Reduce(&hits, &totalhits, 1, MPI_LONG, MPI_SUM, 0, MPI_COMM_WORLD);
if(rank == 0) {
    double ratio = (double)totalhits / (double)DOTS;
    printf("pi = %.12f\n", ratio*4);
}
```

```
MPI_Reduce(sendbuf, recvbuf, count, type, op, root, comm)
```

Matrix-vector Multiplication



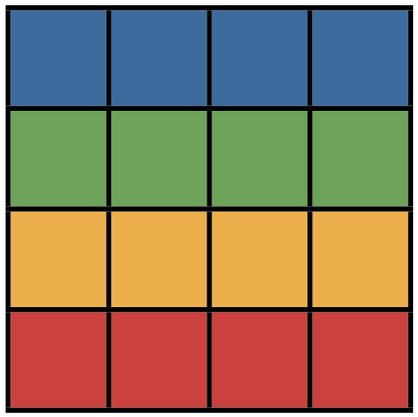
$$b_i = \sum_j A_{i,j}\, x_j$$

Often iterative: Ax, A^2x, A^3x, A^4x...

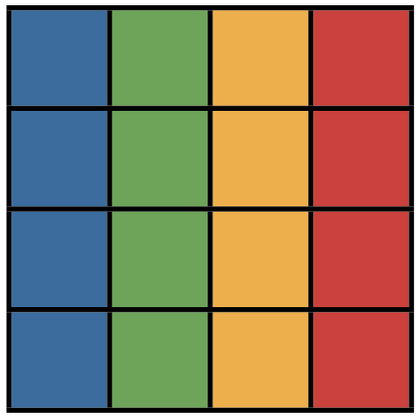
Serial MV multiply

```
#define N       1000

double a[N][N];
double x[N];
double b[N];

// Input a and x...
for(int k = 0; k < nr; k++)
    multiply();
// Output x...

void multiply() {
    // Calculate b values
    for(int i = 0; i < N; i++) {
        b[i] = 0.0;
        for(int j = 0; j < N; j++)
            b[i] += a[i][j] * x[j];
    }

    // Copy b to x
    for(int i = 0; i < N; i++)
        x[i] = b[i];
}
```

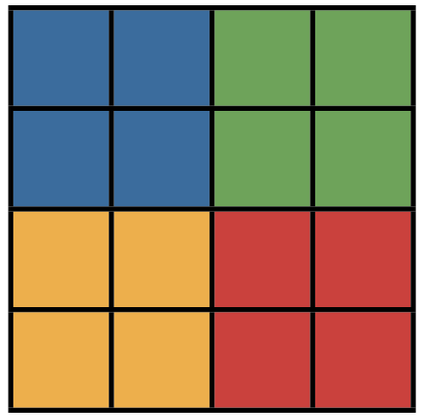
Matrix Parallelizations



Row-wise block striped

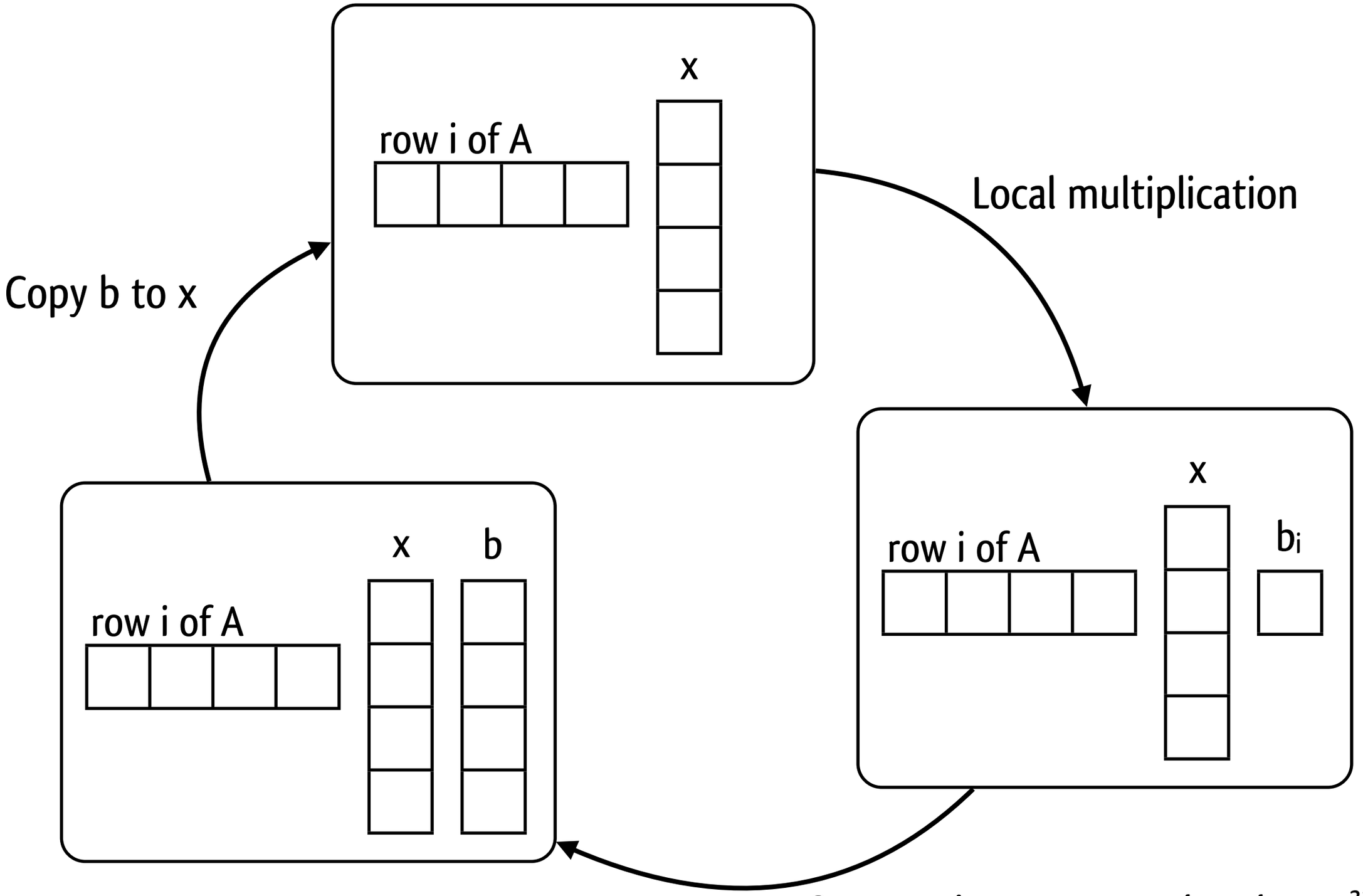


Column-wise block striped

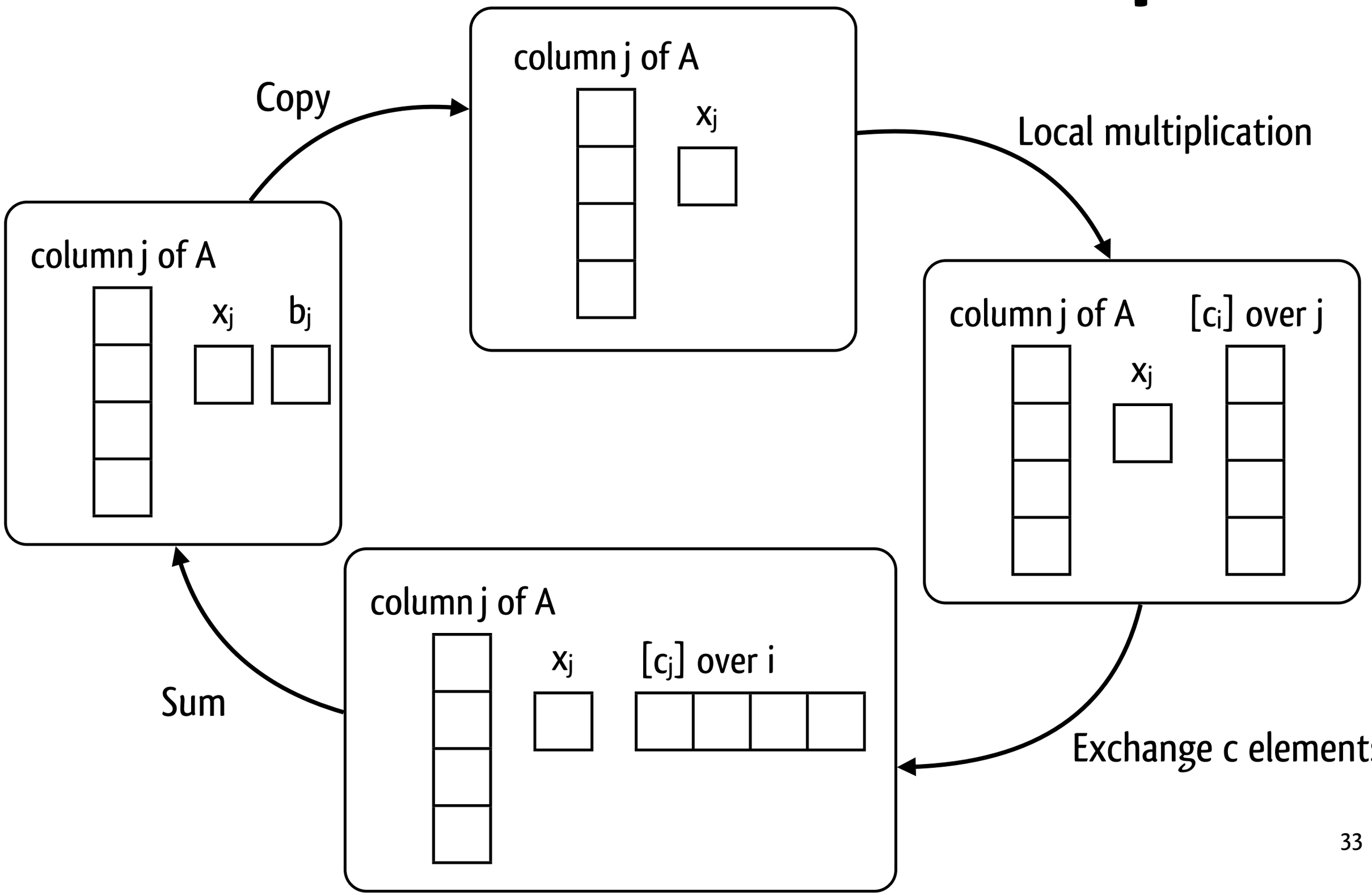


Checkerboard block

Row-wise Block Striped



Column-wise Block Striped



MPI: Row-wise MV multiply 1

```
#define N       1000
#define NPROCS  10
#define NP      (N/NPROCS)

double a[NP][N];
double x[N];
double b[NP];

// Input a and x...
for(int k = 0; k < nr; k++)
    multiply();
// Output x...

void multiply() {
    // Assumption: x values are up to date

    // Calculate local part of b
    for(int i = 0; i < NP; i++) {
        b[i] = 0.0;
        for(int j = 0; j < N; j++)
            b[i] += a[i][j] * x[j];
    }

    // Update x with all parts of b
    ...
}
```

MPI: Row-wise MV multiply 2

```
#define N         1000
#define NPROCS    10
#define NP        (N/NPROCS)

double a[NP][N];
double x[N];
double b[NP];

...
```

```
MPI_Sendrecv(sendbuf, count, type, dest, tag,
        recvbuf, count, type, src, tag,
        comm, status)
```

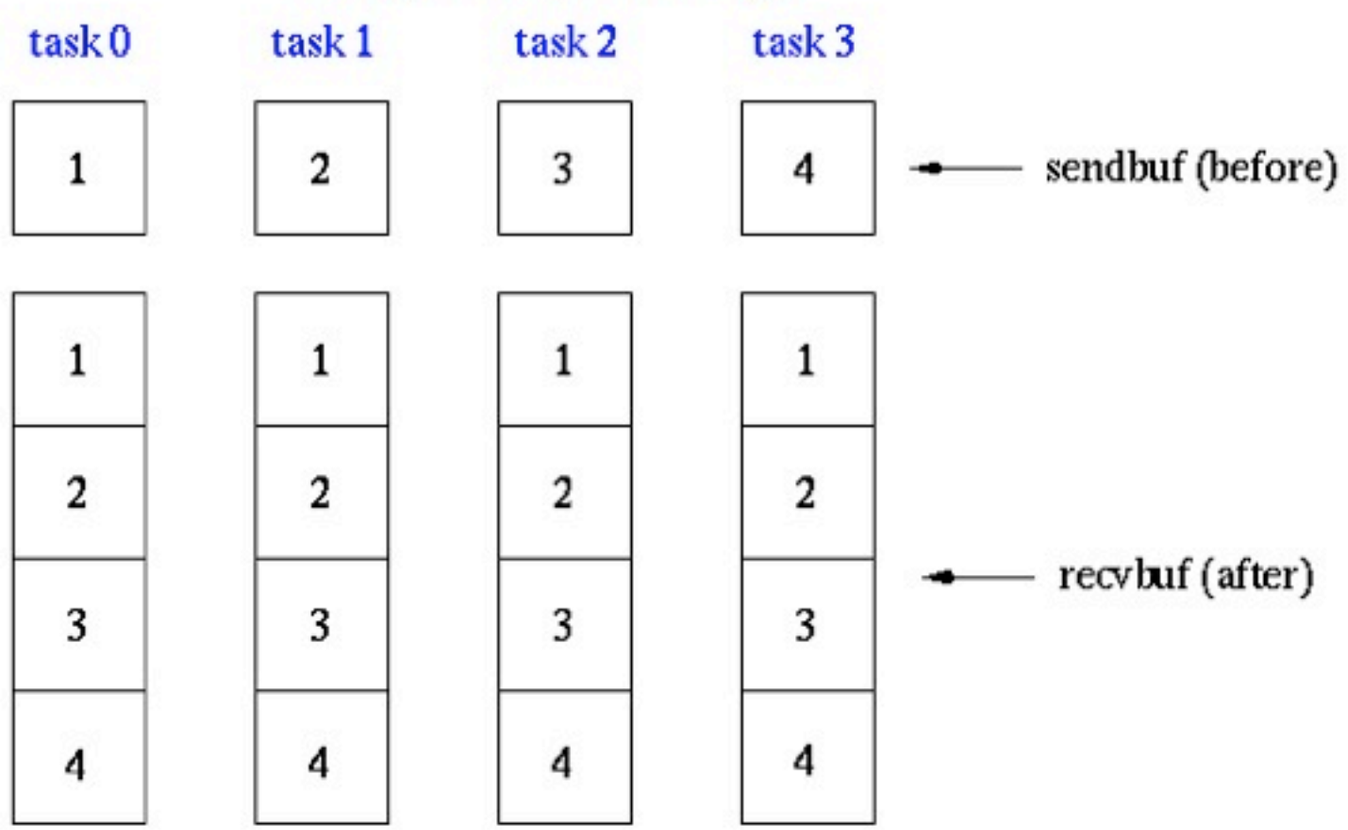
```
// Update x with all parts of b
...
// Copy local part of b to x
for(int i = 0; i < NP; i++)
    x[NP*rank + i] = b[i];

// Copy remote parts of b to local x
for(int k = 1; k < size; k++) {
    int to = (rank + k) % size;
    int from = (rank - k) % size;
    MPI_Sendrecv(b, NP, MPI_DOUBLE, to, rank,
        &x[NP*from], NP, MPI_DOUBLE, from, from,
        MPI_COMM_WORLD, MPI_STATUS_IGNORE);
}
```

MPI Collective: Gather

MPI_Allgather

Gathers together values from a group of processes and distributes to all

```
sendcnt = 1;
recvcnt = 1;
MPI_Allgather(sendbuf, sendcnt, MPI_INT,
              recvbuf, recvcnt, MPI_INT,
              MPI_COMM_WORLD);
```



MPI: Row-wise MV multiply 2

`MPI_Allgather(`*sendbuf*, *count*, *type*, *recvbuf*, *count*, *type*, *comm*`)`

```
#define N        1000
#define NPROCS   10
#define NP       (N/NPROCS)

double a[NP][N];
double x[N];
double b[NP];

...

    // Update x with all parts of b
    MPI_Allgather(b, NP, MPI_DOUBLE, x, NP, MPI_DOUBLE, MPI_COMM_WORLD);
    // Copy local part of b to x
    ...
    // Copy remote parts of b to local x
    ...
```

MPI Collective: Barrier

`MPI_Barrier(`*comm*`)`

- Not intended to communicate anything; synchronize
- Block until all processes have reached the same call

MPI: Summary

- SPMD
- Point-to-point communication (send/recv)
- Collective operations: reductions, gather/scatter, barrier

Unified Parallel C (UPC)

Uniform Parallel C

- PGAS extension of the C programming language
- Specification by UPC consortium
- UPC 1.0 (2001), UPC 1.1 (2003), UPC 1.2 (2005)
- Implementations: Berkeley UPC, gcc UPC, HP UPC, Cray UPC,...

UPC: Hello World

```
#include <upc.h>

int main(int argc, char **argv) {
    printf("Hello world from %d of %d\n",
           MYTHREAD, THREADS);
    return 0;
}
```

- Single program, multiple data (SPMD)
- THREADS = # threads
- MYTHREAD in [0..THREADS)

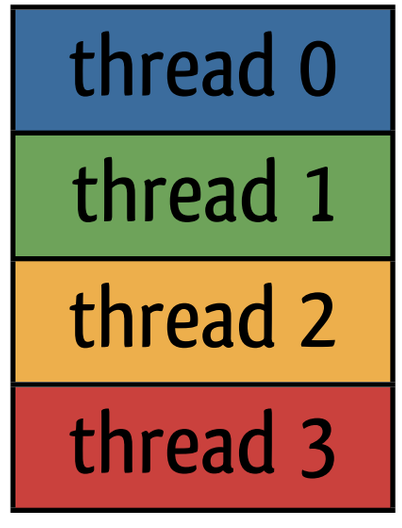
Output:

```
$ upcc [-network=...] [-pthreads] -o upc-hello upc-hello.c
$ upcrun -n 4 ./upc-hello
Hello world from 0 of 4
Hello world from 1 of 4
Hello world from 3 of 4
Hello world from 2 of 4
```

UPC: Shared Memory

- Variables are private (and local) by default
- Types may include `shared` qualifier
  - Shared memory (local or remote)
    - Transparent read/write
- Only global variables or dynamic memory can be shared
  - Not local variables
- Each memory location "belongs" to one thread
  - The "layout" controls how arrays are distributed

UPC: Shared Memory



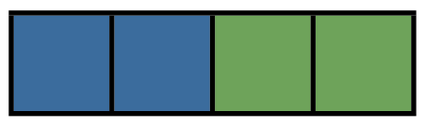

```
shared int a;
```

```
shared int b[NTHREADS];
```

```
shared int c[3*NTHREADS];
shared [1] int c[3*NTHREADS];
```

```
shared [] int d[4];
shared [0] int d[4];
```

```
shared [2] int e[4];
```



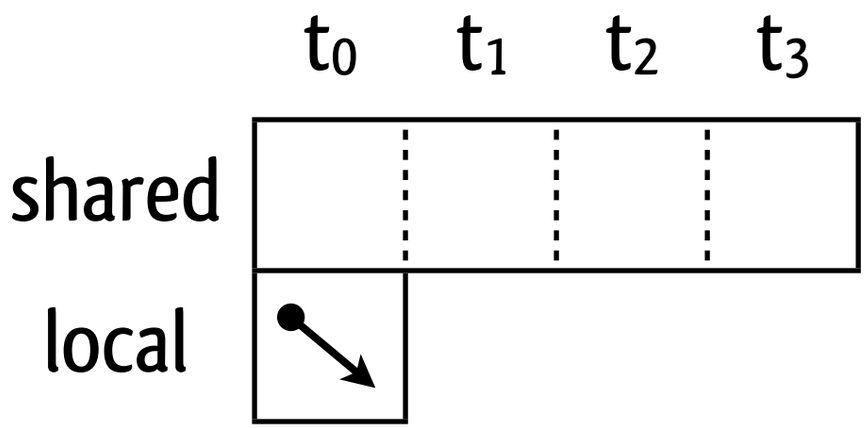
```
shared [3] int f[3*NTHREADS];
```

```
shared [*] int g[10];
```

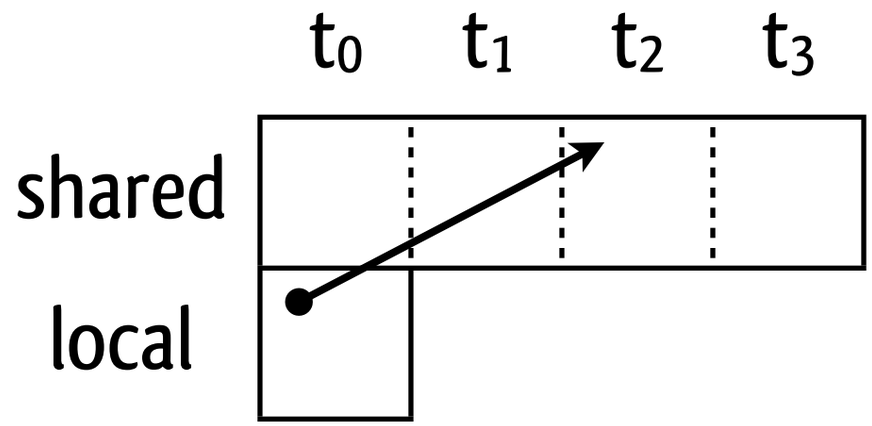
UPC: Monte Carlo

```
#define DOTS (500*1000*1000)

shared long hits[THREADS];

// Calculate local contribution
for(long k = 0; k < DOTS / THREADS; k++)
    if(is_hit(urand(), urand()))
        hits[MYTHREAD]++;

// Wait until everyone is ready
upc_barrier;

if(MYTHREAD == 0) {
    // Add up everyone's contribution
    long totalhits = 0;
    for(long k = 0; k < THREADS; k++)
        totalhits += hits[k];
    double ratio = (double)totalhits / (double)DOTS;
    printf("pi = %.12f\n", ratio*4);
}
```

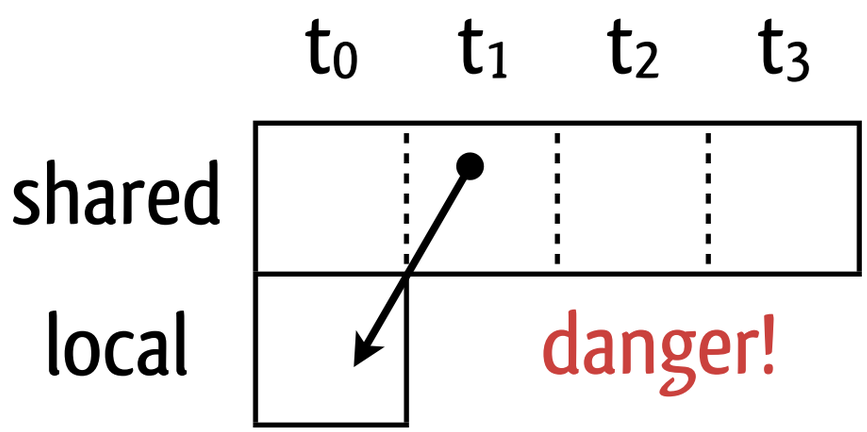
UPC: `shared` and pointers



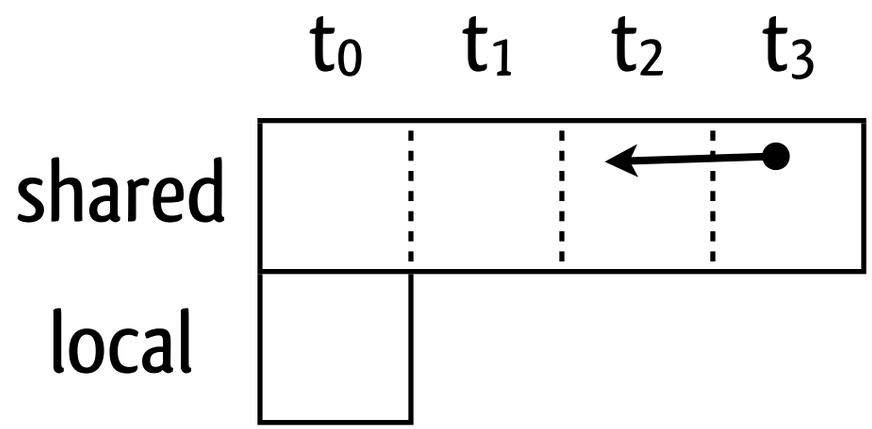

```
int *p;
```




```
shared int *q;
```




```
int* shared r;
```




```
shared int* shared s;
```

UPC: Dynamic Shared Memory

```
shared int *q;    // Global memory address

// Allocate dynamic memory
q = upc_all_alloc(10, sizeof(int));

// Query memory affinity
for(int i = 0; i < 10; i++)
    printf("Thread %d owns q[%d]\n", upc_threadof(&q[i]), i);

// Conversion
int *p;                 // Local memory address
q = (shared int*)p;     // Always illegal
p = (int*)q;            // Legal, but will only work when
                        // upc_threadof(q) == MYTHREAD

// Deallocate dynamic memory
if(MYTHREAD == 0)
    upc_free(p);
```

```
shared [b] T var[n*b]
```

```
upc_all_alloc(n, b*sizeof(T))
```

(collective)

```
int upc_threadof(shared void *)
```

```
upc_free(shared void *)
```

(local)

Other UPC features

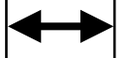
Bulk shared memory read/write

```
upc_memget(void *dst, shared void *src, size_t count)
upc_memput(shared void *dst, void *src, size_t count)
```

Global iteration construct

```
upc_forall(int i=0; i<N; i++; i)
    ... body ...
```

```
upc_forall(int i=0; i<N; i++; &x[i])
    ... body ...
```



```
for(int i=0; i<N; i++)
    if(i % THREADS == MYTHREAD)
        ... body ...
```

```
for(int i=0; i<N; i++)
    if(upc_threadof(&x[i]) == MYTHREAD)
        ... body ...
```

UPC Row-wise MV Multiply

```
#define N       1000
#define NT      N/THREADS

shared [NT*N] double a[N][N];
shared [NT]   double x[N];
shared [NT]   double b[N];

// Input a and x...
for(int k = 0; k < nr; k++)
    multiply();
// Output x...
```

```
void multiply() {
    // Calculate b values
    upc_forall(int i = 0; i < N; i++; &b[i]) {
        b[i] = 0.0;
        for(int j = 0; j < N; j++)
            b[i] += a[i][j] * x[j];
    }
    upc_barrier;

    // Copy b to x
    upc_forall(int i = 0; i < N; i++; &x[i])
        x[i] = b[i];
    upc_barrier;
}
```

This is almost the serial version!

UPC: Summary

- SPMD
- Shared memory with (transparent) read/write
- Distributed arrays with configurable layout
- Synchronization: barriers, locks,...
- Collective operations: gather/scatter, reduce,...

Global Arrays (GA)

Global Arrays

- PGAS library with API for Fortran, C, C++, Python (in that order)
- Developed at Pacific Northwest National Laboratory
  - Smaller than MPI or UPC
- Currently at version 5.2

GA: Hello World

```
#include <mpi.h>
#include <ga.h>
#include <stdio.h>

int main(int argc, char **argv) {
    MPI_Init(&argc, &argv);
    GA_Initialize();

    printf("Hello world from %d of %d\n",
                GA_Nodeid(), GA_Nnodes());

    GA_Terminate();
    MPI_Finalize();
    return 0;
}
```

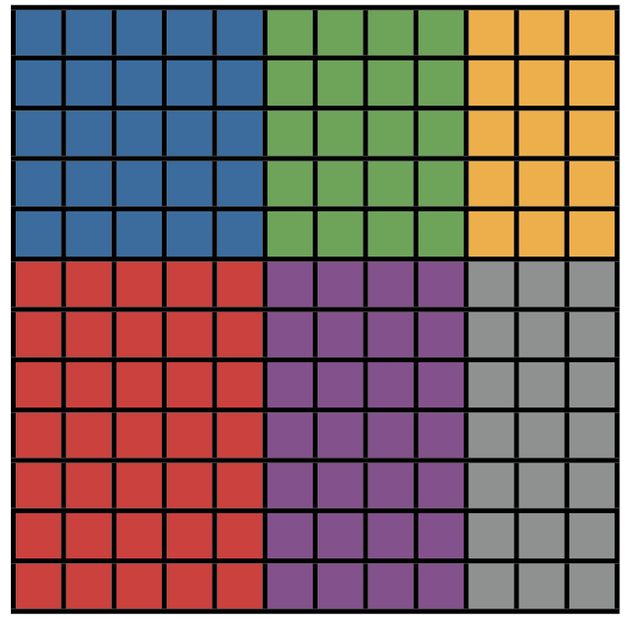
- Single program, multiple data (SPMD)
- Uses MPI for startup

GA: PGAS Library?

- Global arrays: partitioned global data structures
  - Each process stores part of the array
  - Opaque global array handle; manipulations through GA library API
- More high-level, domain-specific approach:
  - “N-dimensional arrays” instead of “memory”
  - “Indices” instead of “pointers”
  - Distribution of the array is hidden → generic code

GA: Create and Destroy

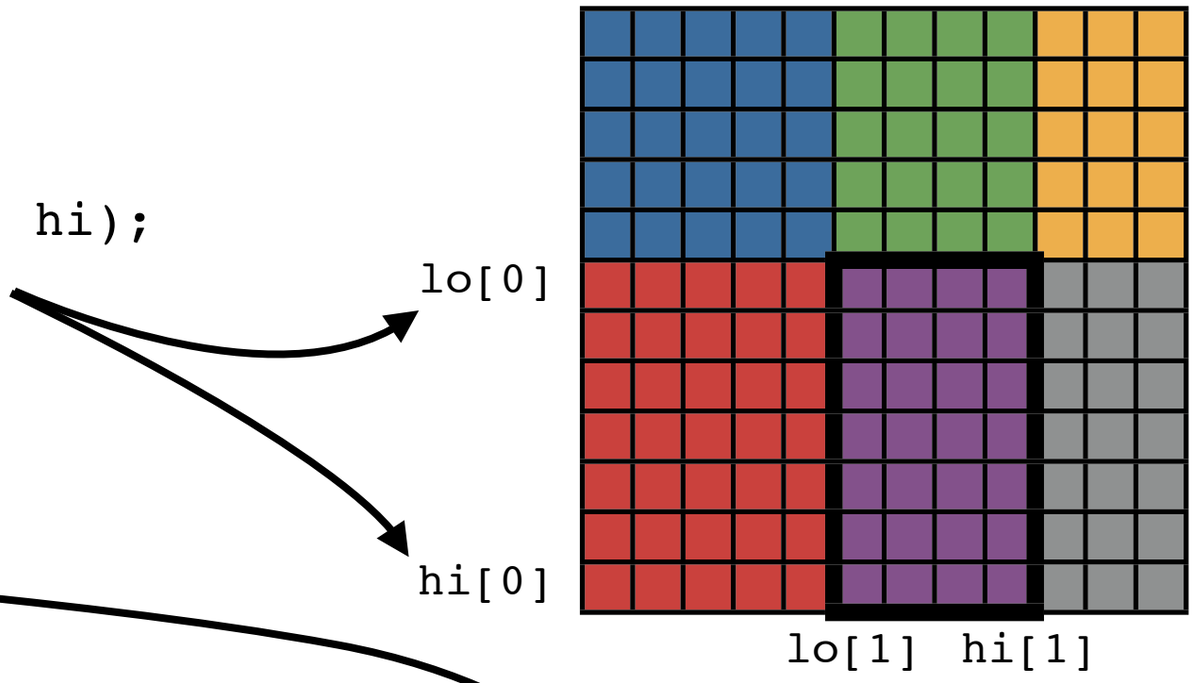
```
int ndims = 2;
int dims[] = { n, n };
int g_a = NGA_Create(MT_C_DBL, ndims, dims, ...);

GA_Print_distribution(g_a);

GA_Destroy(g_a);
```

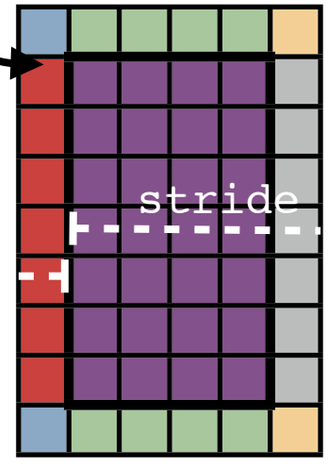


Collective operations!

GA: Query & Local Access

```
// What is my local region?
int lo[ndims], hi[ndims];
NGA_Distribution(g_a, GA_Nodeid(), lo, hi);

// Access the local part
double *m;
int stride;
NGA_Access(g_a, lo, hi, &m, &stride);

// Do some local work
for(int i = 0; i <= hi[0] - lo[0]; i++)
    for(int j = 0; j <= hi[1] - lo[1]; j++)
        m[i * stride + j] = ... ;

// Release the access to the local part
NGA_Release_update(g_a, lo, hi);
```





Local operations!

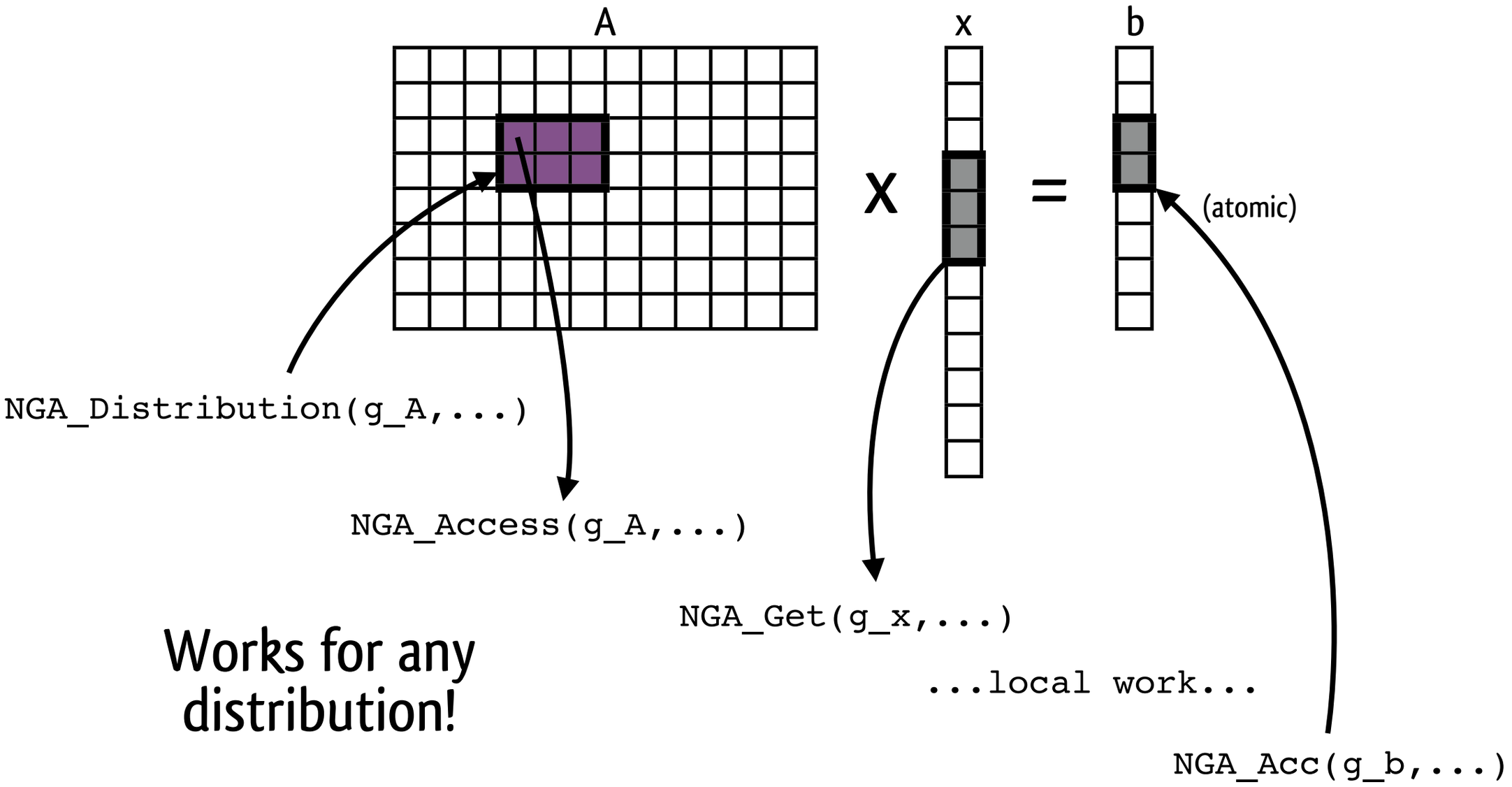
GA: Remote Access

```
NGA_Get(g_a, lo, hi, recvbuf, stride)
NGA_Put(g_a, lo, hi, sendbuf, stride)
NGA_Acc(g_a, lo, hi, sendbuf, stride)
```

One-sided operations!

- Non-transparent: copy data from/to local buffer
- Combination of lower and upper indices specifies “patch”
- Accumulate adds data values atomically

MV multiply



Works for any distribution!

Defined as a new collective operation: `GA_Matvec_mul(g_A, g_x, g_b)`

GA: Toolkit Organization

Primitives	NGA_Create, NGA_Destroy, NGA_Distribution, NGA_Access, NGA_Release, NGA_Get, NGA_Put, NGA_Acc,...
Basic matrix operations	GA_Zero, GA_Fill, GA_Scale, GA_Add, GA_Dgemm, GA_Ddot, GA_norm1,...
Matrix algorithms	GA_Solve, GA_Diag,...