Parallel programming models

Prof. Rich Vuduc
Georgia Tech Summer CRUISE Program
June 6, 2008
Patterson’s Law of Attention Span

"In conclusion…"
- Problem $\rightarrow$ parallel algorithm
- Programming models:
  Algorithms $\rightarrow$ implementation
- Looking forward: DARPA HPCS Languages
Problem → parallel algorithm

Programming models:
Algorithms → implementation

Looking forward: DARPA HPCS languages
Computationally intensive problems

Source: Dubey, et al., Intel (2005)
Problem: Seamless image cloning

Problem:
Seamless image cloning

Idea:
Clone the gradient...

... then reconstruct.

One possible mathematical formulation:

\[ \min_f \iint_{\Omega} |\nabla f - v|^2 \partial\Omega \quad \text{with} \quad f|_{\partial\Omega} = f^*|_{\partial\Omega} \]

“Guided interpolation”

Find \( f(x,y) \) given gradient \( v(x,y) \)
Necessary condition:
Poisson’s equation.

One possible mathematical formulation:
Find \( f(x,y) \) given gradient \( v(x,y) \)

\[
\nabla^2 f = \nabla \cdot v \quad \text{over} \quad \Omega \quad \text{with} \quad f|_{\partial \Omega} = f^*|_{\partial \Omega}
\]
I-D analogue with $v=0$. Apply "calculus of variations."
Solution: \( f(x) = \text{line} \). Gradient is zero \( \Rightarrow \) linearly interpolates boundary.
\[ \mathbf{v} = \nabla g \quad \Longrightarrow \quad \nabla^2 f = \nabla \cdot \nabla g = \nabla^2 g \]
\[ \iff \nabla^2 (f - g) = 0 \]

Let \( f = g + \hat{f} \quad \Longrightarrow \quad \nabla^2 \hat{f} = 0, \text{ with } \hat{f}|_{\partial \Omega} = (f^* - g)|_{\partial \Omega} \]
I-D analogue.
$g = \text{\textbullet\textbullet\textbullet}$

I-D analogue.
\[ g = \text{[graph of a signal]} \]

\[ f^* = \text{[graph of a signal]} f(x) = ? \]

I-D analogue.
$g = \quad \quad \quad f^* = \quad \quad \quad \quad f(x) = \ ?$

$I$-D analogue.
$g = \overbrace{\text{waveform}}^{1}$

$f^* = \overbrace{\text{waveform}}^{2}$

$f(x) = ?$

$x_1$ $x_2$

$\hat{f}$

$= g + \hat{f}$

I-D analogue.
Seamless image cloning:

Summary

- Given: Image with a hole, object to “paste in”
- Find: “Seamless” pasting
- A mathematical formulation:
  - Take gradient of image
  - Solve Poisson’s equation
“Traditional” examples of Poisson’s equation

Electrostatics & gravity
Heat flow
Diffusion
Fluid flow
Elasticity

(2-D) Find $f(x, y)$:

$$\frac{\partial^2 f}{\partial x^2} + \frac{\partial^2 f}{\partial y^2} = v(x, y)$$
One algorithm for solving Poisson: Jacobi’s method

\[ \frac{\partial^2 f}{\partial x^2} + \frac{\partial^2 f}{\partial y^2} = 0 \]

For \( t \leftarrow 1, 2, 3, \ldots \)

\[ f_{i,j}^{t+1} \leftarrow \frac{1}{4} \left( f_{i+1,j}^t + f_{i-1,j}^t + f_{i,j+1}^t + f_{i,j-1}^t \right) \]
Parallelizing Jacobi

- Update points independently
- Partition?
- Communication?
Summary:
Problem → Parallelization
Problem → parallel algorithm

Programming models:
Algorithms → implementation

1. Data parallel
2. Shared memory
3. Message passing
Hybrids, e.g., partitioned global address space

Looking forward: DARPA HPCS languages
Parallel architecture
“design space”

Interconnection Network

Memory

Proc

Proc

Proc

Memory

Memory

Memory
Programming models

- Langs + libs composing abstract view of machine
- Major constructs
  - **Control:** Create parallelism? Execution model?
  - **Data:** Private vs. shared?
  - **Synchronization:** Coordinating? Atomicity?
- Variations in models
  - Reflect diversity of machine architectures
  - Imply variations in cost
Programming model 1: Data parallel

- Program = 1 thread, parallel ops on data
- Communication is implicit
- Drawback: Does not always apply
- Examples: HPF, Matlab, ZPL

% Example: Jacobi in MATLAB

\[ I = 2:(n-1); \]
\[ J = 2:(n-1); \]
\[ F(I,J) = 0.25*(F(I-1,J) + F(I+1,J) + F(I,J+1) + F(I,J-1)); \]
Machine model 1

- Vector processors

- Single-instruction, multiple-data (SIMD)
Programming model 2: Shared memory

- Program = **multiple threads** of control
- Communicate and synchronize via **shared variables**
Race conditions and locks

- **Race condition / data race:**
  Two threads access a variable, with at least one writing and concurrent accesses

```c
shared int s = 0;
```

Thread 1

```c
for i = 0, n/2-1
    s = s + f(A[i])
```

Thread 2

```c
for i = n/2, n-1
    s = s + f(A[i])
```
Race conditions and locks

- **Lock** for atomicity, to avoid races

```c
shared int s = 0;
shared lock lk;

Thread 1
local_s1 = 0
for i = 0, n/2-1
    local_s1 = local_s1 + f(A[i])
lock(lk);
unlock(lk);
s = s + local_s1

Thread 2
local_s2 = 0
for i = n/2, n-1
    local_s2 = local_s2 + f(A[i])
lock(lk);
unlock(lk);
s = s + local_s2
```
Machine model 2a: SMPs

- “Symmetric multiprocessor”
- All processors connect to large shared memory
Machine model 2b: SMTs

- Symmetric multithreaded processors
  - HW threads share memory and functional units
  - Switch threads during long-latency operations

![Diagram of SMTs]

- Memory
  - shared $, shared floating point units, etc.
Cray Eldorado processor

Source: John Feo (Cray)
Machine model 2c: Distributed shared mem.

- Memory logically shared, physical distributed
- Challenge to scale cache-coherency

Cache lines (pages) must be large to amortize overhead

$\Rightarrow$ locality is critical to performance
Programming model 3: Message passing

- Program = **named** tasks (processes)
- Tasks communicate via explicit **send/receive** operations

```
\[
y = \ldots s \ldots
\]
```

![Diagram showing message passing between named tasks](image-url)
Message passing example

Example: All-reduce

Processor 1:
\[ x = A[1] \]
\[
\text{SEND } x \rightarrow \text{Proc. 2} \\
\text{RECEIVE } y \leftarrow \text{Proc. 2}
\]
\[ s = x + y \]

Processor 2:
\[ x = A[2] \]
\[
\text{SEND } x \rightarrow \text{Proc. 1} \\
\text{RECEIVE } y \leftarrow \text{Proc. 1}
\]
\[ s = x + y \]

What could go wrong?
Machine model 3a: Distributed memory

- Separate nodes, memory
- Communicate through network

![Diagram of distributed memory architecture with nodes P0, PI, Pn connected through a network]
Machine model 3b: Global address space

- **Tweak:** NI can directly access the processor
- **One-sided** communication; remote direct-memory access (RDMA)
Programming model 3b: PGAS

- “Partitioned global address space”
- Data shared **but** partitioned
“Concrete” models

- Data parallel: Matlab, HPF
- Shared memory: PThreads / OpenMP / Intel TBB / Cilk
- Message passing: Message Passing Interface (MPI)
- PGAS: UPC, Co-Array Fortran
- Hybrids: CUDA; DARPA HPCS: X10, Chapel, Fortress
Problem → parallel algorithm

Programming models:
Algorithms → implementation

Looking forward (?) : DARPA HPCS languages
DARPA HPCS Program

- High-Productivity Computing Systems
- Goal: Create a new generation of economically viable computing systems by 2010
- Funded industrial/academic alliances
  - Architectures
  - Languages / programming models
  - User “productivity”
HPCS language effort

- New languages
  - X10 (IBM)
  - Chapel (Cray)
  - Fortress (Sun)
- Explicit constructs express parallelism & locality
- PGAS + dynamic parallelism (dynamic threads)
Base languages

- X10 (IBM) extends Java
  - Multi-dimensional arrays
  - Value types
  - Parallelism

- Chapel (Cray), Fortress (Sun): New
  - Fortress – Mathematical syntax (unicode)
  - Challenge: Adoption?
Creating parallelism

- Explicitly parallel (no automatic parallelization)
- Designed to encourage expression of as much parallelism as possible
  - Data, loop, task
  - Rely on compiler + run-time to schedule
- Fortress: Loops and argument evaluation parallel by default
- Dynamic parallelism
Locality

- Explicit notions of locality
  - “Places” in X10
  - “Locales” in Chapel
  - “Regions” in Fortress
- Static and dynamic
- Extensive support for data distribution
Synchronization

Generally believed that locks and barriers are error-prone

All 3 languages provide “atomic blocks”
  - Syntax forces matching begin/end
  - X10: Place-local atomic blocks
  - Speculation + rollback

X10 provides “clocks,” which are barriers attached to set of tasks

Support for “futures” (producer-consumer)
Summary

- Ultimate goal is to solve some problem
  - Formalize
  - Develop algorithm (serial or parallel)

- Choice of programming models
  - Abstract model of machine execution
  - Differ in specification of control, data sharing
  - Basic: Data parallel, shared mem., message passing

- Active area of research and debate
A few concrete programming models

- PThreads
- OpenMP
- MPI
- Cilk
- Unified Parallel C (UPC)
- Co-Array Fortran
POSIX Threads (PThreads)

- Portable system call interface for creating and synchronizing threads
- Threads share all global variables
- Fork/join style

```c
errcode = pthread_create (&thread_id, 
&thread_attribute, 
&thread_fun, 
&fun_arg)

...
errcode = pthread_join (thread_id, NULL);
```

- Reference: [https://computing.llnl.gov/tutorials/pthreads/](https://computing.llnl.gov/tutorials/pthreads/)
Loop-level parallelism

- May fork threads at any time, e.g., within a loop

  ```c
  ... A[n];
  for (i = 0; i < n; ++i)
  
    pthread_create (..., &task, &i);
  ...
  ```

- Must have sufficient granularity to mask thread-creation overhead
Low-level policy control

- Detached state: Avoid pthread_join calls
- Scheduling parameters: priority, policy (FIFO vs. round-robin)
- Contention scope: With what thread does this thread compete for CPU
Barriers for global synchronization (Optional extension)

Usage outline

```c
pthread_barrier_t b;
pthread_barrier_init (&b, NULL, 3);  // 3 threads
...  
pthread_barrier_wait (&b);  // All threads wait
...  
pthread_barrier_destroy (&b);
```
Mutual exclusion locks (mutexes)

- Basic usage

```c
pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;
pthread_mutex_init (&lock, NULL);
...
pthread_mutex_lock (&lock);
   // ... do critical work ...
pthread_mutex_unlock (&lock);
```

- Beware of **deadlock**

<table>
<thead>
<tr>
<th>Thread 1</th>
<th>Thread 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>lock (a);</td>
<td>lock (b);</td>
</tr>
<tr>
<td>lock (b);</td>
<td>lock (a);</td>
</tr>
</tbody>
</table>
OpenMP: An API for multithreaded shared-memory programming

- Programmer identifies **serial** and **parallel regions**, not threads

- Library + directives (requires compiler support)
- Official website: [http://www.openmp.org](http://www.openmp.org)
- Also: [https://computing.llnl.gov/tutorials/openMP/](https://computing.llnl.gov/tutorials/openMP/)
Simple example

```c
int main()
{
    printf ("hello, world!\n"); // Execute in parallel
    return 0;
}
```
int main()
{
    
    omp_set_num_threads (16);

    #pragma omp parallel
    {
        printf (“hello, world!\n”); // Execute in parallel
    } // Implicit barrier/join
    return 0;
}
Concurrent loops

- May parallelize a loop, but **you must check dependencies**

```c
s = 0;
for (i = 0; i < n; ++i)
    s += x[i];
```

```c
#pragma omp parallel for \
    shared (s)
for (i = 0; i < n; ++i)
    #pragma omp critical
    s += x[i];
```

```c
#pragma omp parallel for \
    reduction(+: s)
for (i = 0; i < n; ++i)
    s += x[i];
```
Loop scheduling

- Use “schedule” clause to partition loop iterations
  - **Static**: $k$ iterations per thread, assigned statically
    
    ```
    #pragma omp parallel for schedule static($k$) ...
    ```
  - **Dynamic**: $k$ iterations per thread, using logical work queue
    
    ```
    #pragma omp parallel for schedule dynamic($k$) ...
    ```
  - **Guided**: $k$ iterations per thread initially, reduced with each allocation
    
    ```
    #pragma omp parallel for schedule guided($k$) ...
    ```
  - **Run-time**: Use value of environment variable, **OMP_SCHEDULE**
## Synchronization primitives

<table>
<thead>
<tr>
<th>Synchronization primitive</th>
<th>Description</th>
<th>Code Example</th>
</tr>
</thead>
</table>
| **Critical sections**          | No explicit locks                        | `#pragma omp critical 
{ ... }`                                              |
| **Barriers**                   |                                          | `#pragma omp barrier`                                                      |
| **Explicit locks**             | May require flushing                     | `omp_set_lock(1); 
... 
omp_unset_lock(1);`                                                  |
| **Single-thread regions**      | Inside parallel regions                  | `#pragma omp single 
{ /* executed once */ }`                               |
Cilk: C extensions to support dynamic multithreading
Cilk (Leiserson, et al., 1996+)

- **Fork/join-style C extensions** for dynamic multithreaded apps on SMPs

```c
int fib (int n) {
    if (n < 2) return 1;
    else {
        int x, y;
        x = fib (n-1);
        y = fib (n-2);
        return x + y;
    }
}
```

```cilk
int fib (int n) {
    if (n < 2) return 1;
    else {
        int x, y;
        x = spawn fib (n-1);
        y = spawn fib (n-2);
        sync;
        return x + y;
    }
}
```

- **“Faithful” extension**: Omitting parallel keywords = valid serial C program
```cilk
int fib (int n) {
    if (n < 2) return 1;
    else {
        int x, y;
        x = spawn fib (n-1);
        y = spawn fib (n-2);
        sync;
        return x + y;
    }
}
```

Dynamic computation DAG:

fib(4)
Critical path:

\[ T_\infty = \text{“span”} \]

Degree of parallelism:

\[ \frac{T_1}{T_\infty} \]
\( T_1 = 17 \)
\( T_\infty = 8 \)

\[ \frac{T_1}{T_\infty} \approx 2 \]
Cilk’s work-stealing scheduler

- Processors maintain work queues
- When out of work, processor selects another processor uniformly at random and takes work
Performance of Cilk’s work-stealing scheduler

**Theorem:** Expected running time is

\[ T_p \leq \frac{T_1}{p} + O(T_\infty) \]

**Proof sketch:** See Blumofe & Leiserson (FOCS ’94)

- A processor is either **working** or **stealing**.
- Total time working is \( T_1 \).
- Each steal has \( \frac{1}{p} \) chance of reducing span by 1, so cost of all steals is \( O(p*T_\infty) \).
- Expected time: \( (T_1 + O(p*T_\infty)) / p = T_1/p + O(T_\infty) \)
Cilk vs. PThreads

- What happens if one instance of `foo` waits on another?
- Liveness property
  - Cilk: Lazy ("may") parallelism
  - PThreads: Eager ("must") parallelism

```c
for (i = 0; i < N; ++i)
    spawn-or-fork foo (i);
sync-or-join;
```
Additional features and caveats

- Provides
  - Fence (Cilk_fence)
  - Locking (Cilk_lock)
  - Automatic clean-up for local dynamic allocation (Cilkalloca)
- Aborts
  - “Inlets” support use of spawned results in arbitrary expressions
  - Run-time scheduler uses work-stealing
  - Beware sharing through pointer-passing, deadlocks
Message Passing Interface (MPI)
Recall the message passing model

- Program = **named** processes
- **No shared** address space
- Processes communicate via **explicit send/receive** operations

```
<table>
<thead>
<tr>
<th></th>
<th>P0</th>
<th></th>
<th>P1</th>
<th></th>
<th>Pn</th>
</tr>
</thead>
<tbody>
<tr>
<td>i</td>
<td>2</td>
<td>i</td>
<td>3</td>
<td>i</td>
<td>1</td>
</tr>
<tr>
<td>s</td>
<td>12</td>
<td>s</td>
<td>14</td>
<td>s</td>
<td>11</td>
</tr>
</tbody>
</table>
```

send P1,s

receive Pn,s

Network

y = ..s ..
Message Passing Interface (MPI)

- Logical processes/tasks with distinct address spaces
- Communication primitives
  - Pairwise, or "point-to-point," send & receive
  - **Collectives** on subsets of processes: broadcast, scatter/gather, reduce
- **Barrier** synchronization
- Advanced interface: **topology, one-sided, I/O**
- **Profiling** interface
Hello, world in MPI

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

int main (int argc, char *argv[]) {
    int rank, size;
    MPI_Init (&argc, &argv);
    MPI_Comm_rank (MPI_COMM_WORLD, &rank);
    MPI_Comm_size (MPI_COMM_WORLD, &size);
    printf ("I am %d of %d\n", rank, size);
    MPI_Finalize ();
    return 0;
}
```
Basic concepts: Send and receive

- How to describe “data?”
- How to identify processes?
- How will receiver recognize and screen messages?
- What does it mean for operations to complete?
MPI_COMM_WORLD

Groups

Communicator
Basic concepts: Communicators

- **Group** = subset of processes
- **Communicator** = Group + attributes (e.g., topology)
- **Rank** = process ID in its communicator
- **MPI_COMM_WORLD** = Group consisting of all processes
- **MPI_ANY_RANK** = Wildcard rank
Basic concepts: Data types

- In MPI call, “data” = (address, count, type)
- Data type = (recursively defined)
  - “Standard” scalar types: MPI_INT, MPI_DOUBLE, MPI_CHAR, …
  - An array of data types
  - A strided block of data types
  - An indexed array of blocks of data types
  - An arbitrary structure of data types
Basic concepts:
Message tags and status objects

- **Message tags**
  - Every message has a user-defined integer ID
  - Wildcard: `MPI_ANY_TAG`

- **Status objects**: Opaque structures for querying error and other conditions
MPI blocking send:

**MPI_Send** (buffer-start, count, **type**, dest-rank, tag, communicator)

**Process 1:**

MPI\_Send (data, n, MPI\_INT, 2, tag, comm);

**Process 2:**

MPI\_Recv (data, n, MPI\_INT, 1, tag, comm, &stat);

- **Buffer** = (buffer-start, count, type) ; **Target** = (dest-rank, tag, comm)
- On return:
  - Data delivered to “the system”
  - May **reuse** buffer
  - Semantic note: Target may **not yet** have received message
What happens to data on “send”??
MPI blocking receive:

`MPI_Recv` (buffer-start, count, type, source-rank, tag, comm, status)

- **Buffer** = (buffer-start, count, type) ; **Source** = (source-rank, tag, comm)
- Returns when *matching message* received
- Match on source triplet; wildcards OK
- Receiving fewer than \( n \) items is OK, but more is an error
- May query “status” for more information (e.g., size of message)

Process 1:

```
MPI_Send (data, n, MPI_INT, 2, tag, comm);
```

Process 2:

```
MPI_Recv (data, n, MPI_INT, 1, tag, comm, &stat);
```
(Potentially) Avoid copies with non-blocking communication

- Non-blocking operations **return immediately** with handles
- **Wait** on handles

```c
MPI_Request req;
MPI_Status stat;

MPI_Isend (buf, n, MPI_INT, dest, tag, comm, &req);

// ... do not use "buf" ...

MPI_Wait (&req, &stat);
```

- May poll instead of wait ("test"), or poll or wait on multiple requests
Other communication modes

- **Synchronous** sends (*MPI_Ssend*): Send completes when receive begins
- **Buffered** mode (*MPI_Bsend*): Use user-supplied buffer
- **Ready** mode (*MPI_Rsend*): User guarantees matching receive has posted
- Non-blocking versions of above
- *MPI_Recv* accepts messages sent in any mode
- *MPI_Sendrecv* initiates simultaneous send and receive
Beware of deadlock

**“Unsafe” orderings** of send/receive

(a) Process 1:
- `Recv` (data → 2)
- `Send` (data ← 2)

(b) Process 1:
- `Send` (data → 2)
- `Recv` (data ← 2)

Process 2:
- `Recv` (data → 1)
- `Send` (data ← 1)

Process 2:
- `Send` (data → 1)
- `Recv` (data ← 1)

How to avoid?
Ways to avoid deadlock

- **Use safe orderings**
  
  **Process 1:**
  - Send (data → 2)
  - Recv (data ← 2)
  
  **Process 2:**
  - Recv (data → 1)
  - Send (data ← 1)

- **Use simultaneous send/receive**
  
  **Process 1:**
  - Sendrecv (data → 2)
  
  **Process 2:**
  - Sendrecv (data → 1)
More ways to avoid deadlock

- **Supply** buffer space
  - **Process 1:**
    - Bsend (data → 2)
    -Recv (data ← 2)
  - **Process 2:**
    - Bsend (data → 1)
    -Recv (data ← 1)

- **Use non-blocking** send/receive
  - **Process 1:**
    - Isend (data1 → 2)
    - Irecv (data2 ← 2)
    -Waitall
  - **Process 2:**
    - Isend (data1 → 1)
    - Irecv (data2 ← 1)
    -Waitall
Collective communication

- Higher-level communication primitives
  - **MPI_Bcast**: Broadcast data to all processes
  - **MPI_Reduce**: Combine data from all processes to one process
  - **MPI_Barrier**

- Each process executes same operation
- Presumably optimized/tuned for hardware, but …
Broadcast

Scatter

Gather
All gather

P0
P1
P2
P3
A
B
C
D
A
B
C
D
A
B
C
D
A
B
C
D

All-to-all

P0
P1
P2
P3
A0
A1
A2
A3
B0
B1
B2
B3
C0
C1
C2
C3
D0
D1
D2
D3
Summary: MPI

- Most commonly used MPI primitives
  - Init, Comm_size, Comm_rank, Send, Recv, Finalize
  - Non-blocking primitives for correctness and performance
- “Advanced” MPI features
  - Custom communicators
  - I/O
  - one-sided communication
Unified Parallel C (UPC)
Recall the partitioned global address space (PGAS) model

- Program = **named** threads
- Shared data, but **partitioned** over local processes
- Implied cost model: **remote accesses cost more**

![Diagram of PGAS model]

\[ y = .. s[i] ... \]

\[ i: 2 \]

\[ i: 5 \]

\[ i: 8 \]

\[ s[0]: 27 \]

\[ s[1]: 27 \]

\[ s[n]: 27 \]

\[ s[myThread] = ... \]
Unified Parallel C (UPC)

- Implements PGAS model using concise, explicit parallel extensions to C
- Aimed at “low-level” performance programmers
- Precursors: Split-C, AC, PCP
- Other PGAS languages: Co-Array Fortran, Titanium (Java)
UPC execution model: Threads running in SPMD fashion

- **THREADS** = no. of threads, specified at compile- or run-time
- **MYTHREAD** = current thread's index (0 … THREADS-1)
- **upc_barrier**: All wait (global sync)
“Hello, world” in UPC

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

int main ()
{
    printf ("[Thread %d of %d] Hello, world!\n", MYTHREAD, THREADS);
    return 0;
}
```
Private vs. shared variables in UPC

```c
int mine;  /* thread-private */
shared int ours; /* thread 0 */
```
Shared arrays distributed cyclically by default

```c
shared int x[THREADS];    /* 1 elt per thread */
shared int y[3][THREADS]; /* 3 elt per thread */
shared int z[3][3];       /* 2 or 3 per thread */
```

Example:

```
THREADS = 4
```

黄色 = “lives” on thread 0

Distribution rule:
1. Linearize
2. Distribute round-robin
Synchronization

- “Traditional” all-threads block barrier: `upc_barrier` [LABEL];
- Split-phase barrier
  ```c
  upc_notify;  /* Ready */
  ... do computation ...
  upc_wait;    /* Wait */
  ```
- Locks
  ```c
  upc_lock_t* l = upc_all_lock_alloc ();
  ...
  upc_lock (l);
  ... critical code ...
  upc_unlock (l);
  ```
UPC collectives

- Usual suspects, **untyped**: broadcast, scatter, gather, reduce, prefix, ...
- Interface has synchronization modes
  - Avoid over-synchronizing (barrier before/after is simplest, but may be unnecessary)
  - Data collected may be read/written by any thread
- Simple interface for collecting scalar values (*i.e.*, **typed**)
- Berkeley UPC value-based collectives
- Reference: [http://upc.lbl.gov/docs/user/README-collectivev.txt](http://upc.lbl.gov/docs/user/README-collectivev.txt)
Example: Compute sum of an array

```c
shared double data[N][THREADS];

...

{
    double s_local = 0; /* local sum */
    double s;  /* global sum */

    for (i = 0; i < N; ++i)
        s_local += data[i][MYTHREAD];

    /* Reduce to sum on thread 0 */
    s = bupc_allv_reduce (double, s_local, 0, UPC_ADD);
    /* Implicit barrier */

    if (MYTHREAD == 0)
        printf ("Sum = %g\n", s);
}
```
Common idiom: Owner computes

**Example:** Vector addition

```c
shared double A[N], B[N], Sum[N]; /* laid out cyclically */
...
{
    int i;
    for (i = 0; i < N; ++i)
        if (i % THREADS == MYTHREAD) /* owner computes */
            Sum[i] = A[i] + B[i];
...
}
```
Work sharing with \texttt{upc\_forall}

- Special type of loop for preceding idiom

  \texttt{upc\_forall (init; test; inc; affinity) statement;}

- Programmer asserts iterations are independent

- “Affinity” field
  
  - Integer: \texttt{affinity \% THREADS == MYTHREAD}
  
  - Pointer: \texttt{upc\_threadof (affinity) == MYTHREAD}

- Compiler may do better than iterate N times
Common idiom: Owner computes

**Example:** Vector addition using `upc_forall`

```c
int i;
upc_forall (i = 0; i < N; ++i; i) /* Note affinity */
   Sum[i] = A[i] + B[i];
```
Recall: Shared arrays in UPC

```
shared int x[THREADS];     /* 1 elt per thread */
shared int y[3][THREADS];  /* 3 elt per thread */
shared int z[3][3];        /* 2 or 3 per thread */
```

Example:
- **THREADS** = 4
- Yellow = “lives” on thread 0

Distribution rule:
1. Linearize
2. Distribute cyclically
Recall: Shared arrays in UPC

Example: Vector addition using `upc_forall`

```c
shared int A[N], B[N], C[N]; /* distributed cyclically */
...
  int i;
  upc_forall (i = 0; i < N; ++i; i) /* Note affinity */
    C[i] = A[i] + B[i];
...
```

![Array diagram]
Example: Vector addition using `upc_forall`

```c
shared int [*] A[N], B[N], C[N]; /* distributed by blocks */
... {
    int i;
    upc_forall (i = 0; i < N; ++i; &C[i]) /* Note affinity */
        C[i] = A[i] + B[i];
... }
```

A

B

S
Data layout in UPC

- All non-arrays bound to thread 0
- Variety of layout specifiers exist
  - No specifier (default): **Cyclic**
  - [*]: **Blocked**
  - [0] or [ ]: **Indefinite**, all on 1 thread
  - [b] or [b1][b2]…[bn] = [b1*b2*…*bn]: **Fixed** block size
- **Affinity** of element i = floor(i / block-size) % **THREADS**
- Dynamic allocation also possible (**upc_alloc**)
2-D array layouts in UPC

Example: n x m array

```
shared int [m] a1[n][m];
```

```
shared int [k][m] a2[n][m];
```
Co-Array Fortran (CAF)
Co-Array Fortran (CAF)

- Extends Fortran 95 to support PGAS programming model
  - Program == collection of images (i.e., threads)
  - Array “co-dimension” type extension to specify data distribution

References:
- http://www.co-array.org
- http://www.hipersoft.rice.edu/caf/index.html
Co-array data type

- Declare real array, locally of length $n$, globally distributed

  Example: $n = 3$, \texttt{num\_images()} = 4

  \begin{verbatim}
  real :: A(n)[*]
  \end{verbatim}

- Compare to UPC

  \begin{verbatim}
  shared float [*] A_upc[n*THREADS];
  \end{verbatim}

  \begin{verbatim}
  shared float [3] A_upc[THREADS][3];
  \end{verbatim}
Communication in CAF

- Example: Every image copies from an image, \( p \)

\[
\begin{align*}
\text{real} & :: \ A(\text{n})[\ast]\text{] } \\
\ldots & \\
A(:) &= A(:)[\text{p}] \\
\end{align*}
\]

- Syntax "[p]" is a visual flag to user
More CAF examples

```plaintext
real :: s ! Scalar
real :: z[*] ! “co-scalar”
real, dimension(n)[*] :: X, Y ! Co-arrays
integer :: p, list(m) ! Image IDs

...  
X     = Y[p] ! 1. get
Y[p]   = X  ! 2. put
Y[*]   = X  ! 3. broadcast
Y[list] = X ! 4. broadcast over subset
X(list) = z[list] ! 5. gather
s = minval(Y[*]) ! 6. min (reduce) all Y
X(:,list) = s ! 7. initialize whole co-array
```
Multiple co-dimensions

real :: x(n)[p,q,*]

- Organizes images in logical 3-D grid
- Grid size: \( p \times q \times k \), where \( p^*q^*k == \text{num\_images}() \)