Partitioned Global Address Space Languages (for High-Level Message-Passing Programming)

Jingke Li

Portland State University

Overview

Partitioned Global Address Space (PGAS) is an emerging programming model for message-passing systems, aimed at reducing the programming complexity for these systems.

- PGAS model provides the illusion of a shared address space to a parallel program.
- At the same time, it distinguishes between local and remote memory accesses.

In other words, PGAS model is trying to balance two competing goals:

ease of programming and *high performance*.

How Does It Work?

It requires an underlying communication supporting layer that can

- *perform one-sided communication* —

  Allow an origin node to read or write the memory of a target node, with no explicit interaction required by the target node or any other node.
- *have low latency for remote accesses* —

  Traditional high-latency interfaces such as TCP/IP are generally unacceptable.

PGAS Languages

A group of PGAS languages are emerging:

- Co-Array Fortran (CAF) — Fortran-based, originated at Rice U
- Unified Parallel C (UPC) — C-based, originated at UC Berkeley
- X10 — Java-based, originated at IBM
- Chapel — new language, originated at Cray

Their approaches towards supporting PGAS are different. We'll look at two of them, UPC and Chapel, in more details.

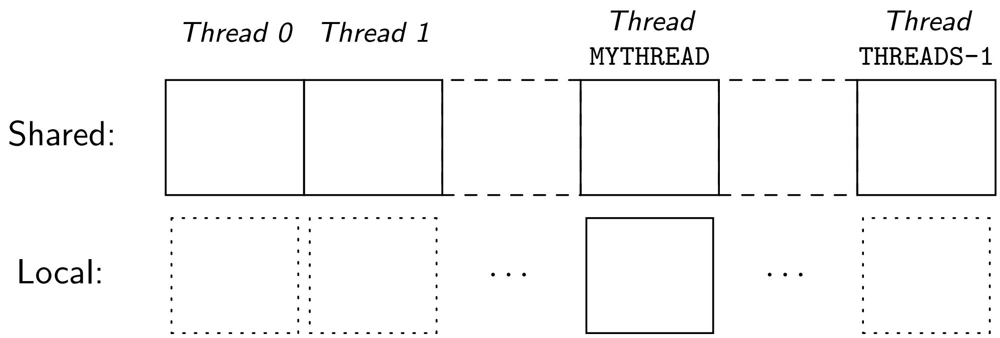
Unified Parallel C (UPC)

Originated from UC Berkeley's earlier projects (*i.e.* Active messages, Split-C, and NOW).

- *Global space* — Arrays
- *Partitioning* — Special declaration (limited forms)
- *Locality* — Local variables and pointers
- *Remote memory access* — Shared variables and pointers

UPC Memory Model

- A linear array of virtual global memory, with affinity to threads.
- A collection of independent local memory modules; one for each thread.
- `THREADS` is an external parameter representing the number of threads the program will use. It can be set either at compile or run time.



UPC's Shared Memory

- Use the keyword `shared` to declare variables for shared memory.
- All scalar variables in shared memory have an affinity to thread 0.
- Array elements have an affinity to the thread in whose memory they are stored.
- High-dimensional arrays are treated as 1D arrays in shared memory mapping.

```
shared         type var;          // affinity to thread 0
shared [bsize] type var[bound];   // cyclic (round robin)
shared         type var[bound];   // same as above, bsize==1
shared [*]     type var[bound];   // block (bsize==bound/THREADS)
shared []      type var[bound];   // affinity to thread 0
```


Shared Memory Examples

Assume `THREADS = 4` in this and the following examples.

```
shared     int a1[8], a2[4][2];
shared [3] int b1[8], b2[4][2];
shared [*] int c1[8], c2[4][2];
shared []  int d1[8], d2[4][2];
```

```
Thread 0:            Thread 1:            Thread 2:            Thread 3:

a1[0]  a2[0][0]      a1[1]  a2[0][1]      a1[2]  a2[1][0]      a1[3]  a2[1][1]
a1[4]  a2[2][0]      a1[5]  a2[2][1]      a1[6]  a2[3][0]      a1[7]  a2[3][1]

b1[0]  b2[0][0]      b1[3]  b2[1][1]      b1[6]  b2[3][0]
b1[1]  b2[0][1]      b1[4]  b2[2][0]      b1[7]  b2[3][1]
b1[2]  b2[1][0]      b1[5]  b2[2][1]

c1[0]  c2[0][0]      c1[2]  c2[1][0]      c1[4]  c2[2][0]      c1[6]  c2[3][0]
c1[1]  c2[0][1]      c1[3]  c2[1][1]      c1[5]  c2[2][1]      c1[7]  c2[3][1]

d1[1]  d2[0][0]
...    ...
d1[7]  d2[3][1]
```

UPC's Pointers

Pointer Types:

```
int* p1;                // local to local
shared int* p2;         // local to shared
int* shared p3;         // shared to local (not recommended)
shared int* shared p4;  // shared to shared
```

Notes:

- Shared and local pointers have different storage formats, and follow different arithmetic rules.
- Shared pointers all reside in Thread 0's shared memory block.
- Local pointers are replicated, reside in their thread's local memory.
- Notice the subtlety in notation:

```
shared int  i;  // a shared int variable
shared int* p;  // a local pointer (to a shared int object)
```


UPC's Pointers (cont.)

Pointers can have their own blocks(!), and they follow their own blocking, instead of the array's:

```
shared int a[10];        // shared array
shared [2] int* p, p2;   // lcl-to-shd ptr with bsz==2
shared [3] int* q, q2;   // lcl-to-shd ptr with bsz==3
p  = &a[1];
q  = &a[1];
p2 = p + 3;
q2 = q + 3;
```

Thread 0:	Thread 1:	Thread 2:	Thread 3:
a[0]	a[1] <- p q	a[2] <- p+2 q+3(q2)	a[3]
a[4]	a[5] <- p+1 q+1	a[6] <- p+3(p2)	a[7]
a[8]	a[9] q+2		

UPC's Forall Construct

Similar to other languages' forall, but needs to specify *thread affinity*.

Example 1:

```
int i, a[10];  // local var and array
upc_forall (i = 0; i < 10; i++; i) {
  a[i] = MYTHREAD;
}
```

Thread 0: (i=0,4,8)	Thread 1: (i=1,5,9)	Thread 2: (i=2,6)	Thread 3: (i=3,7)
a[0] = 0	a[0] (undef'd)	a[0] (undef'd)	a[0] (undef'd)
a[1] (undef'd)	a[1] = 1	a[1] (undef'd)	a[1] (undef'd)
a[2] (undef'd)	a[2] (undef'd)	a[2] = 2	a[2] (undef'd)
a[3] (undef'd)	a[3] (undef'd)	a[3] (undef'd)	a[3] = 3
a[4] = 0	a[4] (undef'd)	a[4] (undef'd)	a[4] (undef'd)
a[5] (undef'd)	a[5] = 1	a[5] (undef'd)	a[5] (undef'd)
a[6] (undef'd)	a[6] (undef'd)	a[6] = 2	a[6] (undef'd)
a[7] (undef'd)	a[7] (undef'd)	a[7] (undef'd)	a[7] = 3
a[8] = 0	a[8] (undef'd)	a[8] (undef'd)	a[8] (undef'd)
a[9] (undef'd)	a[9] = 1	a[9] (undef'd)	a[9] (undef'd)

UPC's Forall Construct (cont.)

Example 2:

```
int i;                   // local var
shared [3] int a[10];    // share array
upc_forall (i = 0; i < 10; i++; i) {
  a[i] = MYTHREAD;
}
```

Thread 0: (i=0,4,8)	Thread 1: (i=1,5,9)	Thread 2: (i=2,6)	Thread 3: (i=3,7)
a[0] = 0	a[3] = 3	a[6] = 2	a[9] = 1
a[1] = 1	a[4] = 0	a[7] = 3	
a[2] = 2	a[5] = 1	a[8] = 0	

UPC's Forall Construct (cont.)

Example 3:

```
int i;                  // local var
shared [3] int a[10];   // share array
upc_forall (i = 0; i < 10; i++; &a[i]) {
  a[i] = MYTHREAD;
}
```

Thread 0: (i=0,1,2)	Thread 1: (i=3,4,5)	Thread 2: (i=6,7,8)	Thread 3: (i=9)
a[0] = 0	a[3] = 1	a[6] = 2	a[9] = 3
a[1] = 0	a[4] = 1	a[7] = 2	
a[2] = 0	a[5] = 1	a[8] = 2	

Forall Example: Matrix Multiplication

```
#include <upc_relaxed.h>
#define N 4
#define P 4
#define M 4

shared [N*P/THREADS] int a[N][P];  // a and c are blocked shared
shared [N*M/THREADS] int c[N][M];  // matrices, initialization is
shared [M/THREADS]   int b[P][M];  // not currently implemented

void main (void) {
  int i, j, l; // private variables
  upc_forall (i = 0; i < N; i++; &c[i][0]) {
    for (j = 0; j < M; j++) {
      c[i][j] = 0;
      for (l = 0; l < P; l++)
        c[i][j] += a[i][l] * b[l][j];
    }
  }
}
```

(Code credit: UPC Tutorial.)

Other UPC Features

- Thread Synchronization
  - Locks
  - Barriers
- Collection Functions
  - Memory allocation
  - Gather and scatter
  - I/O

Chapel PGAS Features

- *Global space* — Domains
- *Partitioning* — Domain maps (very general)
- *Locality* — Locales
- *Remote memory access* — any variables (very general)

Chapel Domains

Domains are first-class index sets

- Specify the size and shape of arrays
- Provide bases for distribution

- *Rectangular Domains:*

```
config var n = 10;
var D1: domain(1) = {1..n};             // 1D domain
var D2: domain(2) = {1..n, 1..n};       // 2D domain
var D3: domain(3) = {1..n, 1..n, 1..n}; // 3D domain
```

- *Sub Domains:*

```
var Inner1: subdomain(D1) = D1[2..n-1];
var Inner2: subdomain(D2) = D2[2..n-1, 2..n-1];
var Inner3: subdomain(D3) = D3[2..n-1, 2..n-1, 2..n-1];
```


Rectangular Domain Methods

Program:

```
config var n = 10;
var D: domain(2) = {1..n, 1..n};
writeln("Bigger: ",  D.expand((1,1));
writeln("Smaller: ", D.expand((-1,-1));
writeln("Exter_p: ", D.exterior((1,1));
writeln("Exter_n: ", D.exterior((-1,-1));
writeln("Inter_p: ", D.interior((1,1));
writeln("Inter_n: ", D.interior((-1,-1));
writeln("Trans_p: ", D.translate((1,1));
writeln("Trans_n: ", D.translate((-1,-1));
```

Output:

```
Bigger:  {0..11, 0..11}
Smaller: {2..9, 2..9}
Exter_p: {11..11, 11..11}
Exter_n: {0..0, 0..0}
Inter_p: {10..10, 10..10}
Inter_n: {1..1, 1..1}
Trans_p: {2..11, 2..11}
Trans_n: {0..9, 0..9}
```

Chapel Domains (cont.)

- *Associative Domains* — Behave like sets.

```
var AD: domain(string);

// add indices
AD += "John";
AD += "Paul";
AD += "Stuart";
AD += "George";
writeln(AD);    // {John, Paul, Stuart, George}

// remove indices
AD -= "Stuart";
AD += "Ringo";
writeln(AD);    // {John, Ringo, Paul, George}
```


Chapel Domain Maps

Domain maps specify mapping of domains to locales.

- Locales = processors
- Pre-defined mappings:
  - block, cyclic, block-cyclic, and replicated
- User may specify the target set of locales for a mapping explicitly.
- To use any of the mappings, a corresponding library needs to be explicitly included in the program.

Domain Map Example 1

```
const D: domain(1) = {1..8};
const BD1 = D dmapped Block(D);
const BD2 = D dmapped Block({2..6});
const BD3 = D dmapped Block({1..12});
var a: [D] int;
var b1: [BD1] int;
var b2: [BD2] int;
var b3: [BD3] int;
forall e in a do e = here.id;
forall e in b1 do e = here.id;
forall e in b2 do e = here.id;
forall e in b3 do e = here.id;
writeln(a);
writeln(b1);
writeln(b2);
writeln(b3);
```

```
linux> ./dmap -nl 4
0 0 0 0 0 0 0 0
0 0 1 1 2 2 3 3
0 0 0 1 2 3 3 3
0 0 0 1 1 1 2 2
```


Domain Map Example 2

```
use BlockDist, CyclicDist, BlockCycDist;
const D = {1..8, 1..8};
const BD = D dmapped Block(D);
const CD1 = D dmapped Cyclic(startIdx=D.low);
const CD2 = D dmapped Cyclic(startIdx=D.high);
const BCD = D dmapped BlockCyclic(startIdx=D.low, blocksize=(2,3));
```

```
var b: [BD] int;
var c1: [CD1] int;
var c2: [CD2] int;
var bc: [BCD] int;

forall e in b do e = here.id;
forall e in c1 do e = here.id;
forall e in c2 do e = here.id;
forall e in bc do e = here.id;

writeln(b);
writeln(c1);
writeln(c2);
writeln(bc);
```

Results:

```
// block              // cyclic (low)
0 0 0 0 1 1 1 1       0 1 0 1 0 1 0 1
0 0 0 0 1 1 1 1       2 3 2 3 2 3 2 3
0 0 0 0 1 1 1 1       0 1 0 1 0 1 0 1
0 0 0 0 1 1 1 1       2 3 2 3 2 3 2 3
2 2 2 2 3 3 3 3       0 1 0 1 0 1 0 1
2 2 2 2 3 3 3 3       2 3 2 3 2 3 2 3
2 2 2 2 3 3 3 3       0 1 0 1 0 1 0 1
2 2 2 2 3 3 3 3       2 3 2 3 2 3 2 3

// block-cyclic       // cyclic (high)
0 0 0 1 1 1 0 0       3 2 3 2 3 2 3 2
0 0 0 1 1 1 0 0       1 0 1 0 1 0 1 0
2 2 2 3 3 3 2 2       3 2 3 2 3 2 3 2
2 2 2 3 3 3 2 2       1 0 1 0 1 0 1 0
4 4 4 5 5 5 4 4       3 2 3 2 3 2 3 2
4 4 4 5 5 5 4 4       1 0 1 0 1 0 1 0
0 0 0 1 1 1 0 0       3 2 3 2 3 2 3 2
0 0 0 1 1 1 0 0       1 0 1 0 1 0 1 0
```

Chapel Locales

Locale is a central concept for supporting explicit locality in programs.

- A locale is an abstract unit of target architecture. It corresponds to a (multicore) computing node.
  - accessing a locale's local variables have uniform cost
- Can't have more locales than computing nodes.
  - a program running on a single processor can only use one locale

Note: On the CS Linux system, the Chapel compiler is compiled to run programs with multiple locales.

- You need to specify the available hosts through the environment variable `GASNET_SSH_SERVERS` first.
- Even if you are running a program without domain map, you need to explicitly specify the number of locales to use.

Example: Multi-Locale "Hello World"

- `Locales` is a built-in array variable holding the set of available locales.
- `numLocales` is a variable representing the number of available locales.

```
coforall loc in Locales do
  on loc do
    writeln("Hello, world! ",
            "from node ", loc.id, " of ", numLocales);
```

```
linux> ./hello-ml -nl 6
Hello, world! from node 0 of 6
Hello, world! from node 5 of 6
Hello, world! from node 3 of 6
Hello, world! from node 4 of 6
Hello, world! from node 2 of 6
Hello, world! from node 1 of 6

linux> ./hello-ml -nl 20
Not enough machines in environment variable SSH_SERVERS to satisfy
request for (20). Only (16) machines available: ...
```

Locale Properties

- Locale has these attributes: `name`, `id`, and `numCores`.

```
writeln("Locales[0].id = " + Locales[0].id);
writeln("Locales[0].name = " + Locales[0].name);
writeln("Locales[0].numCores = " + Locales[0].numCores);
```

```
linux> ./locale-ex1 -nl 1
Locales[0].id = 0
Locales[0].name = african
Locales[0].numCores = 4
```


The On Statement

- `here` is a built-in variable representing the current locale.

```
writeln("start executing on " + here.id +
            " (" + here.name + " with " + here.numCores + " cores)");

on Locales[1] do
  writeln("now we are on locale " + here.id +
            " (" + here.name + " with " + here.numCores + " cores)");

writeln("back on locale " + here.id + " again");
```

```
linux> ./locale-ex2 -nl 2
start executing on 0 (chatham with 4 cores)
now we are on locale 1 (african with 4 cores)
back on locale 0 again
```

Locality and Parallelism are Orthogonal

On-clauses do not introduce any parallelism, but can be be combined with constructs that do:

```
writeln("start executing on locale 0 - " + Locales[0].name);
cobegin {
  on Locales[1] do
    writeln("this task runs on locale 1 - " + Locales[1].name);
  on Locales[2] do
    writeln("while this one runs on locale 2 - " + Locales[2].name);
}
eln("back on locale 0 again");
```

```
linux> ./locale3 -nl 3
start executing on locale 0 - african
while this one runs on locale 2 - catron
this task runs on locale 1 - adelie
back on locale 0 again
```


Variable's Locale Attribute

Every variable is associated with a locale.

- Variable's default locale is 0, but it can explicitly changed.
- Variable's locale can be queried through its `locale` attribute.

```
var x: int;
on Locales(1) {
  var y: int;
  on Locales(2) {
    var z = x;
    writeln("x's locale: " + x.locale.id);
    writeln("y's locale: " + y.locale.id);
    writeln("z's locale: " + z.locale.id);
  }
}
```

```
linux> ./locale1 -nl 3
x's locale: 0
y's locale: 1
z's locale: 2
```

SPMD Programming in Chapel

Since there is no explicit program replication, SPMD programming in Chapel takes the form of master-slave:

```
// Main thread -- global view
proc main() {
  coforall loc in Locales do
    on loc do
      MySPMDProgram(loc.id, Locales.numElements);
}

// Worker thread -- local view
proc MySPMDProgram(me, p) {
  ...
}
```


Chapel Example: Producer-Consumer

```
var buff$: [0..buffersize-1] sync int;

proc main() {
  cobegin {
    producer();
    consumer();
  }
}

proc producer() {
  for i in 1..numItems {
    const buffInd = (i-1) % buffersize;
    buff$(buffInd) = i;
    if (verbose) then writeln("producer wrote value #", i);
  }
  buff$(numItems % buffersize) = -1;
}
```

(Code credit: Chapel distribution.)

Chapel Example: Producer-Consumer (cont.)

```
proc consumer() {
  for buffVal in readFromBuff() {
    writeln("Consumer got: ", buffVal);
  }
}

iter readFromBuff() {
  var ind = 0,
      nextVal = buff$(0);

  while (nextVal != -1) {
    yield nextVal;

    ind = (ind + 1)%buffersize;
    nextVal = buff$(ind);
  }
}
```


Chapel Example: Quicksort

```
config var n: int = 2**15;   // the size of the array to be sorted
config var thresh: int = 1;  // the recursive depth to serialize
var A: [1..n] real;          // array of real numbers

fillRandom(A);       // initialize array with random numbers
pqsort(A, thresh);   // call parallel quick sort routine
verify(A);           // verify that array is sorted

proc pqsort(arr: [],                      // arr: 1D array of values
            thresh: int,                  // thresh: recursive depth
            low: int = arr.domain.low,    // low: index to start sort at
            high: int = arr.domain.high   // high: index to stop sort at
           ) where arr.rank == 1 {        // defined only for 1D array
  if high - low < 8 {
    bubbleSort(arr, low, high);
    return;
  }
  const pivotVal = findPivot();
  const pivotLoc = partition(pivotVal);
  serial thresh <= 0 do cobegin {
    pqsort(arr, thresh-1, low, pivotLoc-1);
    pqsort(arr, thresh-1, pivotLoc+1, high);
  }
```

Chapel Example: Quicksort (cont.)

```
  proc findPivot() {
    const mid = low + (high-low+1) / 2;
    if arr(mid) < arr(low) then arr(mid) <=> arr(low);
    if arr(high) < arr(low) then arr(high) <=> arr(low);
    if arr(high) < arr(mid) then arr(high) <=> arr(mid);
    const pivotVal = arr(mid);
    arr(mid) = arr(high-1);
    arr(high-1) = pivotVal;
    return pivotVal;
  }
  proc partition(pivotVal) {
    var ilo = low, ihi = high-1;
    while (ilo < ihi) {
      do { ilo += 1; } while arr(ilo) < pivotVal;
      do { ihi -= 1; } while pivotVal < arr(ihi);
      if (ilo < ihi) then arr(ilo) <=> arr(ihi);
    }
    arr(high-1) = arr(ilo);
    arr(ilo) = pivotVal;
    return ilo;
  }
}
```


Chapel Example: Quicksort (cont.)

```
proc bubbleSort(arr: [], low: int, high: int) where arr.rank == 1 {
  for i in low..high do
    for j in low..high-1 do
      if arr(j) > arr(j+1) then
        arr(j) <=> arr(j+1);
}

proc verify(arr: []) {
  const n = arr.domain.high;
  for i in 2..n do
    if arr(i) < arr(i-1) then
      halt("arr(", i-1, ") == ", arr(i-1),
              " > arr(", i, ") == ", arr(i));
  writeln("verification success");
}
```

(Code credit: Adapted from Chapel distribution.)