The PGAS model & Introduction to UPC

Dr Michèle Weiland
Applications Consultant, EPCC
The University of Edinburgh
Outline of talk

• HPC, parallel architectures & the motivation behind PGAS

• The PGAS programming model

• Introduction to UPC
  ‣ basic concept
  ‣ data distribution & blocking factors
  ‣ synchronisation & work sharing
  ‣ pointers, dynamic memory allocation & collectives
What is HPC?

- High performance computing = parallel computing
  - distributing computation over many CPUs
- Performance is the key
  - aim is to make codes run faster!
  - not to possible to simply use faster CPUs (heat, power, physical limitations)
What is HPC?

- Maximise parallel speed-up $S(P)$ on $P$ processors

$$S(P) = \frac{T(1)}{T(P)}$$

- parallel algorithms to solve science
- parallel codes that implement algorithms
- parallel machines to run codes
Parallel architectures

• Shared memory

  ‣ each processor has access to a global memory store

  ‣ communications via memory reads/writes
Parallel architectures

- Distributed memory
  - each processor has its own memory and runs a copy of the OS
  - communication via the interconnect

in recent years, these single processors have become multi-core chips
Parallel programming paradigms

- Data parallelism
  - divide data into subsets, process all subsets in the same way

- Task parallelism
  - divide problem into independent tasks and process tasks in parallel

⇒ divide a large problem up into smaller problems!
Challenges facing HPC

• Systems have ten thousands of multi-core CPUs today
  ‣ will go up to hundreds of thousands before end of decade

• We need
  ‣ better algorithms
  ‣ software designed to take advantage of architecture
  ‣ improved parallel programming models
New programming model - why?

• Parallel programming is hard because mainstream languages were designed for serial programming

• No support for parallelism in the languages - specialist libraries are required

• High level of complexity does not encourage well written and properly designed software...

➡ MPI (Message Passing Interface) library and OpenMP API are currently the most widely used approaches in parallel applications
• **Partitioned Global Address Space**

  ‣ logically partitioned

  ‣ local portions for each process
PGAS vs MPI

- multi-threaded control
- global name space
- single-sided communication
- explicit parallel syntax

- multi-threaded control
- private name space
- mostly two-sided communication
- explicit communication

PGAS

MPI
PGAS languages

- UPC
- Coarray Fortran
- Chapel
- Fortress
- X10

New developments

Language extensions
UPC

- Unified Parallel C

- Parallel extension to ISC C99
  - with global shared address space
  - and explicit parallelism & synchronisation

- Both commercial and open-source compilers available
  - LNBL & UC Berkley: http://upc.lbl.gov
UPC and the world of PGAS

• PGAS is a programming model

• UPC is one implementation of the model

  › there are many other implementations

  › all implementations are different, but fundamental concept remains the same!
Private vs shared data

- concept of two memory spaces: **private** and **shared**

- **private** variables are declared as normal C variables
  - multiple instances will exist

```c
int x; // private variable
```

- **shared** variables are declared with shared qualifier
  - only allocated once, accessible by all threads

```c
shared int y; // shared variable
```
The UPC model

thread 0  thread 1  thread 2  thread 3

shared

private

cpu
cpu
cpu
cpu

global partitioned address space
UPC basics

• UPC threads operate independently in SPMD fashion

• Two variables for querying environment:

  ▸ THREADS: holds total number of threads

  ▸ MYTHREAD: stores thread index (runs from 0 to THREADS-1)

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

void main() {
  printf("Thread %d of %d says: Hello!", MYTHREAD, THREADS);
}
```
Data distribution

- if a shared variable is scalar, space is allocated on thread 0 \textit{only}

```c
int x;
shared int y;
```
Data distribution

- if a shared variable is an array, space is by default allocated across shared memory space in cyclic fashion

```c
int x;
shared int y[8];
```

<table>
<thead>
<tr>
<th></th>
<th>thread 0</th>
<th>thread 1</th>
<th>thread 2</th>
<th>thread 3</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>y[0]</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>y[4]</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>y[1]</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>y[5]</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>y[2]</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>y[6]</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>y[3]</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>y[7]</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

private memory space

shared memory space
Data distribution

- if the number of elements in the shared array does not divide by the number of threads, the distribution will be uneven

```c
int x;
shared int y[9];
```

<table>
<thead>
<tr>
<th>thread 0</th>
<th>thread 1</th>
<th>thread 2</th>
<th>thread 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>y[9]</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

private memory space

shared memory space
Data distribution

- change the default data layout by adding a “blocking factor” to shared arrays

```c
int x;
shared [2] int y[8];
```

<table>
<thead>
<tr>
<th>thread 0</th>
<th>thread 1</th>
<th>thread 2</th>
<th>thread 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
</tbody>
</table>

private memory space

shared memory space
2D array decomposition

```
shared [6] int a[8][8];
```

important to think about how blocking factor can impact data layout!

example on 4 threads
Blocking factor

• should be used if default distribution is not suitable

• four different cases:
  
  ‣ shared \([n]\): defines a block size of \(n\) elements
  
  ‣ shared \([0]\): all elements are given affinity to thread 0
  
  ‣ shared \([*]\): when possible, data is stored in contiguous blocks
  
  ‣ shared \([^\hspace{0.1cm}]\): equivalent to shared \([0]\)
Static vs dynamic compilation

- number of UPC threads can be specified at compile time (static) or at runtime (dynamic)

- Advantages
  - dynamic: program can be executed using any number of threads
  - static: easier to distribute data based on THREADS

- Disadvantages
  - dynamic: not always possible to achieve best possible distribution
  - static: program needs to be executed with number of threads specified at compile time
Static vs dynamic compilation

“An array declaration is illegal if THREADS is specified at runtime and the number of elements to allocate at each thread depends on THREADS.”

- `shared int x[4*THREADS];`
  - legal for static and dynamic environment
- `shared[] int x[8];`
- `shared int x[8];`
- `shared[] int x[THREADS];`
  - illegal for dynamic environment
- `shared int x[10+THREADS];`
Side-effects of shared data

Holding data in shared memory space has implications

1. the lifetime of the shared data needs to extend beyond the scope in which it was defined
   - storage duration

2. the shared data needs to be kept up-to-date
   - synchronisation
Storage duration of shared objects

Shared objects cannot have **automatic storage duration**

› any variable inside a function!

Why?

› SPMD model means a shared variable may be accessed outside the lifetime of the function!

Shared variables must either have **file scope** or be declared with **static** keyword.
Synchronisation

• SPMD model means threads operate independently

• Synchronisation vital to ensure all threads reach same point in execution
  
  ‣ necessary for memory and data consistency

  ‣ only read data that is up-to-date, only overwrite data that is no longer needed

• UPC uses barriers for synchronisation

  ‣ most commonly used: upc_barrier
Work sharing with upc_forall

- 4th parameter defines affinity to thread

\textbf{Condition}: iterations of upc_forall must be independent!

\begin{verbatim}
upc_forall (expression; expression; expression; affinity)
\end{verbatim}
Example: maximum of an array

```c
#define max(a,b) (((a)>(b)) ? (a) : (b))

shared int maximum[THREADS];
shared int globalMax = 0;
shared int a[THREADS*10];

void main(int argc, char **argv) {
  ... // initialise array a

  upc_barrier;
  upc_forall(int i=0; i<THREADS*10; i++; i){
    maximum[MYTHREAD] = max(maximum[MYTHREAD], a[i]);
  }
  upc_barrier;
  if (MYTHREAD == 0){
    for (int thread=0; thread<THREADS; thread++){
      globalMax = max(globalMax, maximum[thread]);
    }
  }
  upc_barrier;
  ...}
```

Here: shared variables have file scope!

Ensure all threads found local maximum

Make sure globalMax is found before being used!
Example: vector addition (1/3)

- three vectors with default distribution - module operation identifies local elements

```c
#include <upc.h>
#define N 100*THREADS

shared int v1[N], v2[N], v1plusv2[N];

void main() {
    int i;
    for(i=0; i<N; i++)
        if (MYTHREAD == i%THREADS)
            v1plusv2[i] = v1[i] + v2[i];
}
```

if distribution changes, this code will fail to identify local elements - however it will still produce the correct result!
Example: vector addition (2/3)

- alternative implementation would iterate in steps of THREADS and eliminate the need for the modulo operation

```c
#include <upc.h>
#define N 100*THREADS

shared int v1[N], v2[N], v1plusv2[N];

void main() {
    int i;
    for(i=0; i<N; i+=THREADS) {
        v1plusv2[i] = v1[i] + v2[i];
    }
}
```

if distribution changes, this code will fail to identify local elements - however it will still produce the correct result!
Example: vector addition (3/3)

• implementation using upc_forall, taking advantage of the affinity parameter

```c
#include <upc.h>
#define N 100*THREADS

shared int v1[N], v2[N], v1plusv2[N];

void main() {
    int i;

    upc_forall(i=0; i<N; i++; i) /* even if the distribution changes, upc_forall will behave correctly */
        v1plusv2[i] = v1[i] + v2[i];
}
```
Working sharing & performance

- perform as much computation as possible on local data

- remote memory operations are expensive!

```
#include <upc.h>

shared [THREADS] int a[THREADS][THREADS];
shared int b[THREADS], c[THREADS];

void main (void)
{
  int i, j;

  upc_forall( i = 0 ; i < THREADS ; i++ ; &c[i]) {
    c[i] = 0;
    for ( j= 0 ; j < THREADS ; j++)
      c[i] += a[i][j]*b[j];
  }
}
```

Ensure matrix A is distributed by row.

![Matrix Distribution Diagram]
UPC pointers

1. private to private
   \[
   \text{int} \; *p1;
   \]
   standard C pointer

2. private to shared
   \[
   \text{shared int} \; *p2;
   \]
   private pointer into the shared memory space

3. shared to private
   \[
   \text{int} \; \text{*shared} \; p3;
   \]
   shared pointer into the private memory space - not recommended!

4. shared to shared
   \[
   \text{shared int} \; \text{*shared} \; p4;
   \]
   shared pointer into the shared memory space
UPC pointers

- pointers in UPC have 3 fields
  - thread: the thread affinity of the pointer
  - address: the local address of the block
  - phase: the location of the element with a block

- it is allowed to cast a shared pointer to private (although there will be some loss of thread and phase information), but a cast the other way round would produce unknown results and is therefore not allowed
Dynamic memory allocation

• in private memory space, usual C functions apply

• in shared space, UPC offers three different functions

  ‣ upc_alloc: allocate local shared spaces

  ‣ upc_global_alloc: allocate multiple global spaces

  ‣ upc_all_alloc: allocate a global shared memory space collectively

• upc_free used to deallocate shared memory
UPC collectives

• requires upc_collective.h header file

• implemented by most compilers, but performance not necessarily optimised

• two types of collectives

  ▶ **relocalisation**: upc_all_broadcast, upc_all_scatter, upc_all_gather, ...

  ▶ **computational**: upc_all_reduceT, upc_all_sort, ...

• calls to these functions must be performed by all threads
#include <upc_collective.h>

shared [] int A[2];
shared [2] int B[N][2];

upc_all_broadcast(B, A, 2*sizeof(int), UPC_IN_ALLSYNC | UPC_OUT_ALLSYNC);
References & further reading

  
  http://upc.gwu.edu/docs/upc_specs_1.2.pdf

- Berkley Unified Parallel C project homepage:
  
  http://upc.gwu.edu/

- Tarek El-Ghazawi et al. “UPC: Distributed Shared Memory Programming”. Available through the Wiley Online Library.