CHAPEL

Version 1.11.0

Author: Yağmur Şentürk
Supervisor: Stéphane Gimenez

June 23, 2015

OVERVIEW

OVERVIEW

- What is Chapel?
- History
- Motivating Themes
- Language Features
- 99 bottles of beer

WHAT IS CHAPEL?

CHAPEL

- Cascade High Productivity Language
- Productive parallel language developed at Cray Inc.
- Chapel designed for high-end supercomputers
- runs on multicore desktops/laptops, commodity clusters, and the cloud
- supports a multithreaded execution model for data parallelism, task parallelism, concurrency, and nested parallelism
- supports code reuse and rapid prototyping
- easy to learn for programmers of C, C++, Java, Python, ..

HISTORY OF CHAPEL

EMERGENCE OF CHAPEL

- Design and development at Cray Inc. under the DARPA High Productivity Computing Systems (HPCS) program
- 2002 – HPCS program started with five teams
  - Cray Inc., Hewlett-Packard, IBM, SGI and Sun
- 2003 initial designs of new programming languages began
  - Cray - Chapel
  - IBM - X10
  - Sun(Oracle) - Fortress
- four developers: David Callahan, Hanz Zima, Brad Chamberlain, John Plevjak

DEVELOPMENT PHASES

- **2003-2006** – draw up the vision of Chapel, first publication about Chapel
- **2006-2008** – stabilization of design and compiler architecture
- **2008-2012** – Chapel at an open source control repository (sourceforge), started with a public release mechanism

MOTIVATING THEMES

MOTIVATING THEMES

- General Parallelism
  - Chapel supports general parallel programming
  - supports data parallelism, task parallelism, concurrency, nested parallelism
  - users have not to change language for trying out other algorithms
- Locality Control
  - control over data value storage and task execution
  - users can execute parallel computations near the accessed variables
- Multiresolution Design
  - supports programming at higher or lower levels
- Data-Centric Synchronization
- Portable, Open Source
- Object-Oriented Programming

LANGUAGE FEATURES

BASIC LANGUAGE FEATURES

- syntax similar to C and Java
- left-to-right declaration

```
var declare: int = 1;
```

- variable types: bool, int and uint, real, complex, string
- declaration modifiers: var, const, config
- supports type inference
  - ```
    var sometype = 1.234;

    const anothertype = 2*sometype;
    ```
- record and class types
- range and array types
- tuple types
    ```

range types

- unique expression
- [low]..[high]
- represents sequence of integers
- example: 1..3 $\rightarrow$ 1, 2, 3
- if low and high is undefined than the range is unbounded in that direction $\rightarrow$ example: 1..

array types

- creating arrays are different to C and Java
- declaration:
  var a: [0..3]int;
  var b: [1..3, 0..5]real;

TUPLE TYPES

- tuple variables are groups of numbers or ordered pairs

example

```
var tuple1: (int, int) = (50,100);
var tuple2: (int, string, real)=(5,"tuple test", 3.9);

writeln((tuple1));
writeln("1 = "+tuple2(1)+", 2 = "+tuple2(2)+", 3 = "+tuple2(3));
```

output

(50, 100)
1 = 5, 2 = tuple test, 3 = 3.9

IF CONDITION

- very similar to C syntax
- first version with braces

```
if(condition) {
  //do something
}else{
  //do something else
}
```

- second version without braces

```
if(condition) then
  //do something
else
  //do something else
```

FOR LOOPS

- syntax

```
for [index—expression] in [iterable—expression] {
  //statements
}
```

- iterable-expression can be an array, tuple or a range

FOR LOOP EXAMPLE

```
var length: int = 4;
var array: [1..length] string = ("this","is","an","example!");

writeln("first loop:");

for i in {1..5}{
    write(i," ");
}

writeln("\n second loop:");

for i in array{
    write(i," ");
}
```

output

first loop:
1 2 3 4 5

second loop:
this is an example!

WHILE LOOPS, DO WHILE LOOPS

while loop

- same syntax as in C and Java

```
while (condition) {
    //do something
}
```

do while loop

- same syntax as in C and Java

```
do{
    //do something
} while (condition);
```

PROCEDURE

- procedures in Chapel are like functions in C or methods in Java
- arguments and return type can be omitted
- main procedure without any arguments
- example

```
proc main(){
    writeln("Hello World!");
}
```

- main procedure is not necessary

PROCEDURE

- passing arguments to procedure

```
helloName("Yagmur");
writeln(sum());
writeln(sum(2,2));

proc helloName(name:string){
    writeln("Hello "+name);
}

proc sum(a: int = 3, b: int = 3){
    return a+b;
}
```

- output:

  Hello Yagmur

  6

  4

RECORD AND CLASS TYPES

class

- reference based
- data structures with associated state and functions
- creating objects by calling a class constructor
- variables of class types either refer to an instance of that class or to one of its derived classes

record

- value based
- data structure with value semantics, like primitive types
- variables of record types include only the fields defined by that record

RECORD EXAMPLE

```
record recordtype {
    var a: int;
    proc double() {
        return a*2;
    }
}

r1 = new recordtype(4);
r2 = r1;
r2.a = 10;

writeln("The double of r1 is ",r1.double()," and the double of
    r2 is ",r2.double());
```

output:

The double of r1 is 8 and the double of r2 is 20

CLASS EXAMPLE

```
class classtype {
    var a: int;
    proc double() {
        return a*2;
    }
}

c1 = new classtype(4);
c2 = c1;
c2.a = 10;

writeln("The double of c1 is ",c1.double()," and the double of
    c2 is ",c2.double());
```

output:

The double of c1 is 20 and the double of c2 is 20

- every parallelism is in connection with tasks

Unstructured Task Parallelism

- begin keyword for creating a task
- example:

```
writeln("This is the FIRST print from the original task");
begin{
    writeln("Here the second task is created")
    writeln("The second task terminates after this");
}
writeln("This is the SECOND print from the original task");
```

- This is the FIRST print from the original task
  This is the SECOND print from the original task
  Here the second task is created
  The second task terminates after this

Sync Statement

- sync keyword for waiting to all created tasks within to complete
- example:

```
writeln("This is the FIRST print from the original task");
sync{
    begin{
        writeln("Here the second task is created");
        writeln("The second task terminates after this");
    }
}
writeln("This is the SECOND print from the original task");
```

- This is the FIRST print from the original task
  Here the second task is created
  The second task terminates after this
  This is the SECOND print from the original task

Synchronization Variable

- value of the variable stores a full/empty state
- read blocks until the variable is full
- write blocks until the variable is empty

- example:

```
var written$ : sync bool;
var b: int = 0;

begin write();
read();

proc read(){
    writeln("waiting for write...");
    const wait = written$;
    writeln("read value of b = "+b);
}

proc write(){
    b = 3;
    writed$ = true;
    writeln("written$...");
}
```

- output:
  waiting for write...
  writed...
  read value of b = 3

Structured Parallelism

- cobegin keyword guarantees that the original task waits until its child task terminates
- example:

```
writeln("This is the FIRST print from the original task");

cobegin{
    writeln ("Here the second task is created");
    writeln("The second task terminates after this");
}
writeln("This is the SECOND print from the original task");
```

- This is the FIRST print from the original task
  The second task terminates after this
  Here the second task is created
  This is the SECOND print from the original task

- parallel programming style easier to use

Forall-Loops

- forall is the data parallel loop construct of Chapel
- similar to for loops but forall uses a random number of tasks
- example:

```
forall i in 1..5{
    write(i+" ");
}
```

- or:

```
[i in 1..5] write(i+" ");
```

- output: 4 5 1 2 3

Domains and Arrays

- domain describes an index set
- domains for specify iteration spaces, define the shape and size of arrays
- domains can be resized dynamically

- example (1-dimension):

```
var d1: domain(1) = 1..3;
var array: [d1] int;

for i in d1 {
    array[i] = i;
}
writeln(array);

d1 = 1..3*2;

for i in d1 {
    array[i] = i;
}
writeln(array);
```

- output:

1 2 3

1 2 3 4 5 6

- example (2-dimension):

```
var d2: domain(2) = {1..3 , 1..6};
var arr: [d2] int;

for i in d2.dim(1){
    for j in d2.dim(2){
        arr[i,j]=i*2;
    }
}
writeln(arr);
```

- output:

```
2 2 2 2 2 2
4 4 4 4 4 4
6 6 6 6 6 6
```

Reduction

- reduction is an operation that joins a set of values and builds a single value
- reduction reduces code and is also faster
- syntax

```
var varName = [reduce_operator] reduce [iterator_expression];
```

- reduce operator can be +, *, & , |, ^, && , ||, min, max, minloc, and maxloc
- iterator expression can be any iterable expression

Reduction

- example

```
var d1: domain(1) = 1..10;
var array: [d1] int;

for i in d1 {
    array[i] = i;
}

var sum = + reduce array;
writeln(sum);
```

- output: 55
- the same as:

```
for i in d1{
    sum += array[i];
}
```

Scan

- scan performs like a sequantial operation
- scan carries down the intermediate results
- same syntax, operator and expression as in reduction

```
var varName = scan_operator scan iterator_expression;
```

- example

```
var d1: domain(1) = 1..10;
var array: [d1] int;
for i in d1 {
    array[i] = i;
}
var sum = + scan array;
writeln (sum);
```

- output: 1 3 6 10 15 21 28 36 45 55

99 BOTTLES OF BEER

CONCLUSION

CONCLUSION

- Chapel – Cascade High Productivity Language
- developed from Cray Inc. under the HPCS program in 2002
- is a productive parallel programming language
- supports data, task, nested parallelism and concurrency
- used for education at some univerisities

BIBLIOGRAPHY

- http://chapel.cray.com/papers/BriefOverviewChapel.pdf
- http://chapel.cray.com/overview.html
- http://www.cray.com/blog/chapel-productive-parallel-programming/
- http://faculty.knox.edu/dbunde/teaching/chapel/#Procedures
- http://www.99-bottles-of-beer.net/lyrics.html
- https://github.com/chapel-lang/chapel/blob/master/doc/release/chapelLanguageSpec.pdf

Questions?