Linguagem Chapel

Walter Perez Urcia

Universidade de São Paulo
Instituto de Matemática e Estadística
Departamento de Ciências da Computação

17 de junho de 2015

Timeline

1. **Introdução**
2. **Chapel**
   - Que é Chapel
   - Instalação e configuração
   - Tipos de dados
   - Comparação com outras linguagens
   - Tipos de paralelismo

Soma de matrizes

Quantas linhas são necessários para a sua implementação?

Soma de matrizes

OpenMP

```
1 #pragma omp shared( A , B , C , chunk ){
2    #pragma omp for schedule( static , chunk )
3    for( i = 0 ; i < N ; i++)
4        for( j = 0 ; j < N ; j++)
5            C[ i ][ j ] = A[ i ][ j ] + B[ i ][ j ] ;
6 }
```

Soma de matrizes

CUDA

```
1 const int BlockSizeX = 32 ;
2 const int Factor = 4 ;
3 const int BlockSizeY = BlockSizeX / Factor ;
4 __global__ void add( int* C , int* A , int* B , const int N ){
5     int col = blockIdx.x * BlockSizeX + threadIdx.x ;
6     int row = blockIdx.y * BlockSizeX + threadIdx.y * Factor ;
7     for( i = 0 ; i < Factor ; i++)
8         C[ ( row + i ) * N + col ] = A[ ( row + i ) * N + col ] + B[
              ( row + i ) * N + col ] ;
9 }
```

Soma de matrizes

OpenMPI

```
1 MPI_Comm_size( MPI_COMM_WORLD , &npes ) ;
2 MPI_Comm_rank( MPI_COMM_WORLD , &myrank ) ;
3 if( myrank == ROOT )
4     for( target = 0 ; target < npes ; i++){
5         MPI_Send( A+N*target , N , MPI_INT , target , target ,
              MPI_COMM_WORLD ) ;
6         MPI_Send( B+N*target , N , MPI_INT , target , target ,
              MPI_COMM_WORLD ) ;
7     }
8 MPI_Recv( myA , N , MPI_INT , ROOT , myrank , MPI_COMM_WORLD , &
      status ) ;
9 MPI_Recv( myB , N , MPI_INT , ROOT , myrank , MPI_COMM_WORLD , &
      status ) ;
10 for( i = 0 ; i < N ; i++) myC[ i ] = myA[ i ] + myB[ i ] ;
11 MPI_Send( myC , N , MPI_INT , ROOT , 0 , MPI_COMM_WORLD ) ;
12 if( myrank == ROOT )
13     for( sender = 0 ; sender < npes ; sender++)
14         MPI_Recv( C+N*sender , N , MPI_INT , sender , 0 ,
              MPI_COMM_WORLD , &status ) ;
```

Soma de matrizes

Linhas

- OpenMP: 6
- CUDA: 9
- OpenMPI: 14

Então, quantas linhas são necessários com Chapel?

Que é Chapel

- Desenvolvido por Cray Inc no ano 2007[1]
- Otimizado para programação paralela
- Tem uma abstração de alto nível para muitas operações
- Usado em HPCS (High Productivity Computing Systems)
- Baseado em C/C++, Java e Python

Configuração

```
. util/setchplenv.sh
```

Instalação

```
make
make check
```

Hello World

Para compilar: $chpl - o < executvel >< fonte >$

Exemplo de Compilação e Execução:

chpl -o hello.o hello.chpl

./hello.o

```
1 writeln( "Hello World!" ) ;
```

Tipos de dados[2]

Tipo	Bits	Default Init
int	64	0
uint	64	0
real	64	0.0
imag	64	$0.0i$
complex	128	$0.0 + 0.0i$
string	n/a	

Semelhanças

Com C/C++

- Records: Equivalente a Struct
- Class: Igual que em C++
- Pode sobrecarregar operadores

Com Python

- Cada archivo é um módulo e pode ser importado em outros
- Ranges: $1..N$, $1..N$ # 5, $1..N$ *by* 2
- Tuplas: Igual que em Python

Diferenças

Variáveis de configuração

Por exemplo, para o programa *Hello2.chpl*, sua variável de configuração *numIters* tem valor 10, mas pode ser alterado ao executâ-lo da seguinte forma:

./hello2.o −−numIters 1000

Listing 1: Hello2.chpl

```
1 config const numIters = 10 ;
2 for i in 1..numIters do
3    writeln( "Hello World! from iteration " , i , " of " , numIters
         ) ;
```

Chapel tem três tipos de paralelismo desenvolvido

1. Data Parallelism
2. Task Parallelism
3. Multi-Locale Parallelism

Data Parallelism

Definição

Estilo de programação paralela em que o paralelismo tem cálculos sobre coleções de dados ou seus índices

Hello World em paralelo

Por exemplo, o seguinte programa é a versão em paralelo do programa anterior:

Listing 2: Hello3.chpl

```
1 config const numIters = 10 ;
2 forall i in 1..numIters do
3     writeln( "Hello World! from iteration " , i , " of " , numIters
        ) ;
```

A segunda linha foi mudada de *for* a *forall*, então a saída do programa pode ser:

```
1 Hello World! from iteration 4
2 Hello World! from iteration 2
3 Hello World! from iteration 1
4 ...
```

Domínios

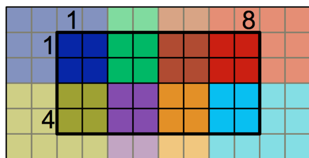
Os domnínios são úteis para declarar arrays e fazer operações com eles. Por exemplo:

```
1 config const m = 4 , n = 8 ;
2 const D = { 1..m , 1..n } ;
3 const Inner = D[ 2..m-1 , 2..n-1 ] ;
4
5 var A , B , C: [D] real ;
6
7 forall ( i , j ) in D {
8     A[ i , j ] = -1.0 ;
9     B[ i , j ] = 4.0 ;
10 }
11 var sum = + reduce abs( A[ D ] + B[ D ] ) ;
```

Mapas de domínios

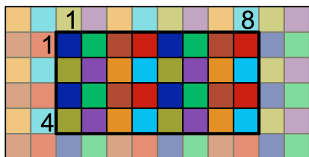
Os mapas de domínios dizem ao compilador como tem que distribuir a memória de uma coleção de dados. Por exemplo:

```
1 var Dom = { 1..4 , 1..8 } dmapped Block( { 1..4 , 1..8 } ) ;
```



distributed to

```
1 var Dom = { 1..4 , 1..8 } dmapped Cyclic( startIdx = ( 1 , 1 ) ) ;
```



distributed to

Task Parallelism

Definição

- Tarefa: Unidade de computação que pode executar em paralelo com outras tarefas
- Thread: Recurso do sistema que executa tarefas

Então, task parallelism é um estilo de programação paralela em que o paralelismo tem tarefas especificadas pelo programador.

Task Parallelism

- Begin: Executa uma tarefa de forma asíncrona
- Cobegin: Executa muitas tarefas em paralelo e espera que todas terminam
- Coforall: Cria uma tarefa por cada iteração

```
1 proc quickSort( arr , low , high ){
2     if( high - low < 4 ){
3         bubbleSort( arr , low , high ) ;
4     }else{
5         const pivot = partition( arr , low , high ) ;
6         cobegin{
7             quicksort( arr , low , p - 1 ) ;
8             quicksort( arr , p + 1 , high ) :
9         }
10    }
11 }
```

Task Parallelism

- Sync: Para esperar que a variável tenha um valor quando vai ser usada
- Atomic: Cada mudança de valor da variável é atómico

```
1 var fut : sync real ;
2 begin fut = compute() ;
3 res = computeSomethingElse() ;
4 useComputedResults( fut , res ) ;
```

Multi-locale Parallelism

Definição

Estilo de programação paralela em que o paralelismo é feito em várias máquinas para um mesmo programa

Multi-locale Parallelism

Todos os programas em Chapel tem a variável de configuração *numLocales* e também pode ser modificada ao executar. Além disso, tem a constante *Locales* que é um array das máquinas configuradas em Chapel. Também tem os seguintes métodos para cada locale:

- locale.id: Retorna o índice na constante *Locales*
- locale.name: Retorna o nome da máquina
- locale.numCores: Retorna o número de cores da máquina
- locale.physicalMemory: Retorna a memoria disponível na máquina

Exemplo

A comunicação de dados entre as diferentes máquinas é implícita. Por exemplo, para duas máquinas temos:

```
1 var x , y : real ; // Locale 0
2 on Locales( 1 ){ // Send task to Locale 1
3     var z: real ; // Locale 1
4     z = x + y ; // Locale 1 reads from Locale 0
5
6     On Locales( 0 ) do
7         z = x + y ; // Remote modification of z in Locale 0
8 }
```

Então, quantas linhas são necessários para soma de matrizes em Chapel?

Então, quantas linhas são necessários para soma de matrizes em Chapel?

```
1 C = A + B ;
```

Obrigado!

Referências

Cray Inc.
Chapel Language Official Site, 2015.

Cray Inc.
Quick Reference, 2015.