Universidade de São Paulo
Instituto de Matemática e Estatística
MAC 5742 - Computação Paralela e Distribuída

Linguagem Chapel

Autor:

Walter Perez Urcia

São Paulo

Junio 2015

Resumo

Neste artigo o objetivo é explorar a linguagem Chapel e fazer comparações com outras linguagens desenvolvidos para computação paralela. As comparações serão baseados em clássicos programas de produto escalar e soma de matrizes.

Sumário

1 Introdução **5**

2 História **6**

2.1 Orígem 6
2.2 Versões 6
2.3 Motivações 7

3 Especificações da linguagem **8**

3.1 Instalação e configuração 8
3.2 Considerações previas 8
3.3 Paralelismo de dados (Data Paralellism) 9
3.3.1 Forall 9
3.3.2 Domínios (Domains) 9
3.3.3 Mapas de dominios (Domain maps) 10
3.4 Paralelismo de tarefas (Task Paralellism) 10
3.4.1 Begin 11
3.4.2 Cobegin 11
3.4.3 Coforall 11
3.4.4 Sync 12
3.4.5 Atom 12
3.5 Paralelismo multi máquina (Multi Locale Paralellism) 12

4 Experimentos e resultados **13**

4.1 Configuração para os experimentos 13
4.2 Experimento 1: Soma de matrizes 13
4.2.1 Códigos 13
4.2.2 Resultados 14
4.3 Experimento 2: Produto escalar 14
4.3.1 Códigos 14
4.3.2 Resultados 15

5 Conclusões **16**

Lista de Figuras

1 Mapeamento em blocos 10
2 Mapeamento em ciclos 10

1 Introdução

Computação paralela tem resultado em muitos avanços significativos em ciência e tecnologia ao longo das últimas décadas. Mas não muitos desenvolvedores são capazes de usar da melhor forma linguagens feitos para programação paralela. Pior ainda é o número de desenvolvedores que podem construir modelos de programação paralela para super computadores o sistemas computacionais de alto nível de produtividade (HPCS, em ingles).
Existem varios linguagens para poder fazer programas que sejam executados em paralelo como OpenMPI, CUDA, OpenMP e muitos outros, mas o que muitos de estos tem em comum é que são precisadas sempre muitas linhas de código para poder fazer até simples instruções em paralelo. Além disso, podem ser consideradas as seguintes características:

- Paralelismo geral: A capacidade de utilizar vários tipos de paralelismo num mesmo programa
- Control de localidade: Onde são guardadas as variáveis e as tarefas e como se comunicam entre elas
- Diferença entre as linguagens para HPCS: Em termos de sintaxe da linguagem, tarefas possíveis e tempo de implementação
- Portável: Pode ser executado sem importar o sistema operativo da máquina

Por isso é que a companhia Cray criou uma nova linguagem chamada Chapel como alternativa às linguagens existentes no mercado e que seja simple de entender. Nas próximas seções será explicado a linguagem Chapel, seu história, sintaxe e comparação com outras linguagens.

2 História

2.1 Orígem

A Agência de Projetos de Pesquisa Avançada de Defesa, encargada das tecnologias novas para uso militar no Estados Unidos, lançou um programa no ano 2002 que incluía as companhias Cray Inc, Hewlett-Packard, IBM, SGI e Sun. Eles tinham que criar novas tecnologias para melhorar o desempenho dos sistemas de alto desempenho (HPC, High Performance Computing) nas áreas de portabilidade e robustez. Para cumplir com o objetivo tiveram que considerar muitas características, incluindo entre elas: memoria do computador, sua arquitetura, e por último propuseram criar novas linguagens.

No seguinte ano (2003), foram aceitos projetos para criação de novas linguagens, tendo Cascade como o projeto da companhia Cray Inc a cargo de Burton Smith. Mas logo o nome mudaria a Chapel como uma aproximação do acrónimo Cascade High Productivity Language. Ao fin do ano 2003, também mudaria de chefe de projeto a David Callahan porque o anterior chefe não achava que criar uma nova linguagem fosse a solução para os sistemas de alto desempenho.

Nos três primeiros anos, o time foi criado o compilador para a linguagem e também foi estabelecido a visão do projeto e da linguagem. A partir do ano 2006, foram desenvolvidos dois diferentes tipos de paralelismo: de tarefas (2007) e de dados (2008). Além disso, eles tentaram otimizar todas as instruções que podiam para melhorar seu desempenho. Por último no ano 2009, o time fiz o projeto de código aberto (Open-Source) para que mais gente pudesse dar recomendações e observações sobre a linguagem. Também começaram a receber aportes à linguagem e foi lançada a primeira versão estável de Chapel (versão 0.9) [?].

2.2 Versões

Desde seu lançamento até agora, Chapel tem tido as seguintes versões estáveis:

- Versão 0.9: 16 de abril de 2009
- Versão 1.0: 13 de novembro de 2009
- Versão 1.1: 15 de julho de 2010
- Versão 1.2.0: 29 de outubro de 2010
- Versão 1.3.0: 22 de abril de 2011
- Versão 1.4.0: 22 de fevereiro de 2012
- Versão 1.5.0: 19 de abril de 2012
- Versão 1.6.0: 18 de outubro de 2012
- Versão 1.7.0: 18 de abril de 2013

- Versão 1.8.0: 17 de outubro de 2013
- Versão 1.9.0: 17 de abril de 2014
- Versão 1.10.0: 2 de outubro de 2014
- Versão 1.11.0: 2 de abril de 2015

Todas as versões anteriores podem ser baixados no site Sourceforge [?]. Além disso, a última versão estável da linguagem pode ser sempre baixado de seu site no Github [?].

2.3 Motivações

Os principais objetivos que tiveram para criar Chapel foram:

- Paralelismo geral: Eles queriam que todos os tipos de paralelismo existam e podam ser usados num mesmo programa sem ter que mudar para outra linguagem em que seja mais fácil de usar ou misturar muitos deles porque cada um é bom na alguma característica.
- Global-view programming: Poder fazer cálculos em estruturas de dados (como arrays) de tal forma que cada processo tenha uma parte (como em OpenMPI ao enviar mensagens).
- Abstração de alto nível: Que a linguagem tenha abstrações de alto nível, mas também podem ser usadas instruções de baixo nível quando seja necessário.
- Control de localidade: Definir onde são salvadas as variáveis e as tarefas e como se comunicam entre elas. Além disso, poder obter o valor de qualquer variável desde qualquer processo do programa (seja remota ou local).
- Diferença entre linguagems para HPC: Não queriam que seja difícil entender o código nem tenha linhas adicionais para especificar o paralelismo
- Portável: Que os programas desenvolvidos na linguagem podem ser executados em qualquer sistema operativo

3 Especificações da linguagem

3.1 Instalação e configuração

Para poder usar a linguagem Chapel só ter que baixar o código fonte de seu site no Github ([?]) e executar as seguintes instruções em terminal:

- . util/setchplenv.sh
- make
- make check

Uma vez instalado Chapel, para compilar e executar um programa em Chapel deve colocar:

- chpl -o <executável> <fonte>
- ./<executável>

Por exemplo para um programa chamado *hello.chpl* temos:

- chpl -o hello.o hello.chpl
- ./hello.o

3.2 Considerações previas

A linguagem Chapel foi feito baseado nas linguagens Java, C/C++, Python e Matlab. Por isso, tem algumas semelhanças com cada um como as seguintes:

- Com C/C++:
  - Records: Equivalente a Struct
  - Clases: Orientado a objetos
  - Sobrecarrega de 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

Mas ainda tem semelhanças com varios linguagens, seu sintaxe é mais próxima a linguagem C/C++.

Por outro lado, a principal diferença em termos de execução é que os programas podem ter variáveis de configuração que podem ser mudadas de valor ao executar o programa e são definidas com a palavra $config$. Por exemplo, para o programa 1 podemos executá-lo da seguinte forma:

./hello.o –numIters 1000

Programa 1: Hello World iterativo

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

Ao executá-lo da forma anterior, o valor da variável *numIters* será 1000 e não 10. Em 3.3, 3.4 e 3.5 serão explicados os diferentes tipos de paralelismo que foram desenvolvidos em Chapel.

3.3 Paralelismo de dados (Data Paralellism)

Este é um estilo de programação paralela em que o paralelismo tem cálculos sobre coleções de dados ou seus índices. Para poder usar o paralelismo de dados em Chapel, podemos usar:

- Forall (3.3.1)
- Domains (3.3.2)
- Domain Maps (3.3.3)

3.3.1 Forall

A instrução *forall* é muito similar ao *for* convencional que tem muitas linguagens, mas a diferença é que o primeiro muda a execução de uma loop de forma sequencial a uma versão paralela. O número de processos que são criados para fazer a loop de forma paralela dependem do número de cores que tem o computador.
Por exemplo, podemos mudar a linha 2 do Programa 1 e obter uma versão paralela de Hello World. Da mesma forma, pode ser usada para move-se em um array ou uma matriz sem ter que adicionar ou mudar muitas linhas de código.

3.3.2 Domínios (Domains)

Os domínios são úteis para declarar arrays dinámicos e fazer operações com eles. Devido a que os arrays em Chapel são estáticos, ou seja, seus dimensões não podem ser mudadas uma vez declarados, mas os domínios se podem ser mudados para dimensões mais grandes ou mais pequenas.
No Programa 2 podemos ver as operações que podem ser feitas com domínios. Na linha 1 mostra como se declara um domínio de dois dimensões e a linha 2 mostra como é que se pode obter a interseção de dois domínios.

Programa 2: Uso de domínios

```
1   config const m = 4 , n = 8 ;
2   const D = { 1..m , 1..n } ;
3
4   var A , B , C : [D] real ;
5   forall ( i , j ) in D {
6     A[ i , j ] = −1.0 ;
```

```
7         B[ i , j ] = 4.0 ;
8       }
9       var sum = + reduce abs( A[ D ] + B[ D ] )
```

Para declarar um array com as dimensões de um domínio só tem que ser declarado como na linha 4. Além disso, as linhas 5-8 mostram que um domínio pode ser recorrido como um iterador. Por último, os domínios podem ser usados para fazer operações de redução como na linha 9.

3.3.3 Mapas de dominios (Domain maps)

Os mapas de domínios dizem ao compilador como tem que distribuir a memória de uma coleção de dados. Existem varios tipos de mapeamento de domínios, mas os mais importantes são Block e Cyclic. Por exemplo, supondo que temos um computador com 8 cores as figuras 1 e 2 mostram a distribuição dos elementos aos diferentes processos em paralelo executados em uma instrução *forall* e sua forma de declarar os mapeamentos de domínios.

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

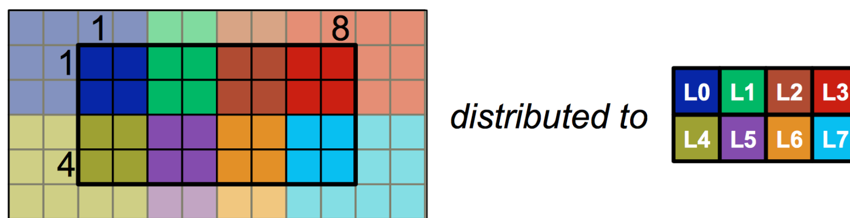

Figura 1: Mapeamento em blocos

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

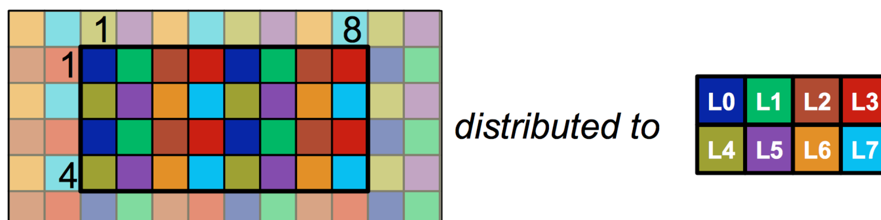

Figura 2: Mapeamento em ciclos

3.4 Paralelismo de tarefas (Task Paralellism)

É um estilo de programação em que o paralelismo tem tarefas especificadas pelo programador que podem ser executadas em paralelo. Em Chapel, uma tarefa é diferente a um thread, o segundo é um recurso do sistema que pode executar uma ou mais tarefas. Para paralelismo de tarefas temos as seguintes intruções:

- Begin (em 3.4.1)
- Cobegin (em 3.4.2)
- Coforall (em **??**)

Além disso, tem modificadores para variáveis:

- Sync (em 3.4.4)
- Atom (em 3.4.5)

3.4.1 Begin

Executa uma tarefa de forma asíncrona. O programa 3 mostra um exemplo do uso de Begin.

Programa 3: Uso de Begin

```
1   writeln( "Original task prints this" ) ;
2   begin{
3     writeln( "Other task will be created to print this" ) ;
4     computeSomething() ;
5     writeln( "The other task will terminate after printing this" ) ;
6   }
7   writeln( "Original task may print before the other task completes" ) ;
```

3.4.2 Cobegin

Executa muitas tarefas e espera que todas terminem para continuar o programa. Por exemplo, o algoritmo Quicksort pode ser executado de forma mais rápida executando a divisão do array em paralelo usando cobegin como se mostra no programa 4.

Programa 4: Quicksort com Cobegin

```
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  }
```

3.4.3 Coforall

Cria um thread por cada iteração da loop. É útil para ter um melhor control do número de processos que são executados em paralelo, mas a desvantagem é que pode fazer muito lento o programa se o número de threads é muito grande porque terá que criar espaços de memoria para cada um. Por exemplo, mudando a linha 2

do programa 1 a $coforall$ este criaria $numIters$ threads e os executaria todos em paralelo.

3.4.4 Sync

Funciona como um promise, espera a que a variável tenha um valor quando vai ser usada. É usado para fazer cálculos custosos em paralelo con outras tarefas que não precisam muito tempo e não dependem da primeira. Um possível programa usando *Sync* se mostra a continuação:

Programa 5: Uso de Sync

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

No programa anterior, a linha 4 não será executada até que a instrução na linha 2 não tinha terminado porque a variável fut não tem um valor ainda.

3.4.5 Atom

Um problema que pode aparecer ao usar $forall$ é que se se tem uma variável que acumula cálculos, uma tarefa acesa antes que outra à variável e muda seu valor. Então ao final da execução da loop, essa variável não terá o valor correto. A solução para esse problema é usar o modificador *atom* que faz que cada mudança de valor da variável seja atómico. O programa 6 mostra a solução da situação explicada antes.

Programa 6: Uso de Atom

```
1  var sum : atom real = 0.0 ;
2  forall i in 1..100 do
3    sum += i ;
4  writeln( "SUM = " , sum ) ;
```

3.5 Paralelismo multi máquina (Multi Locale Paralellism)

É um estilo de programação paralelo em que o paralelismo é feito em várias máquinas para um mesmo programa. Para esto, todos os programas em Chapel tem uma variável de configuração chamada *numLocales* que pode ser modificada ao executar como qualquer outra variável de configuração. Além disso, tem a constante *Locales* que é um array das máquinas configuradas em Chapel.
Cada locale tem os seguintes atributos:

- 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 memória disponível na máquina

Por último, a comunicação entre as diferentes máquinas, as variáveis e as tarefas que contém cada uma pode ser especificada usando o bloco de instruções *on Locales*(< *ID* >). No seguinte programa pode ser visto a interação entre as máquinas.

Programa 7: Interação entre máquinas

```
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 ; // Reads values of x and y from Locale 0
5    on Locales( 0 ) do
6      z = x + y ; // Remote modification of z on Locale 0
7  }
```

Chapel permite que todos os tipos de paralelismo mencionados anteriormente sejam usados em um mesmo programa sem nenhum problema. Desta forma cumple con seu objetivo de ter um paralelismo geral como se diz em 2.3.

4 Experimentos e resultados

4.1 Configuração para os experimentos

Para cada experimento foram feitos programas usando OpenMP, Chapel e OpenMPI que serão executados em um processador Core i5. Em todos os casos serão comparados os tempos de execução só dos cálculos e não da geração de dados. Os parâmetros para cada linguagem são os seguintes:

- OpenMP: 8 threads
- Chapel: Threads por default da máquina
- OpenMPI: 8 threads

Para o experimento 1, as matrizes são de 32 x 32 e para o segundo experimento os vetores tem tamanho 32768.

4.2 Experimento 1: Soma de matrizes

4.2.1 Códigos

Nos programas 8, 9 e 10 mostram as soluções para a linguagem Chapel, OpenMP e OpenMPI respetivamente.

Programa 8: Soma com Chapel

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

Programa 9: Soma com OpenMP

```
1#pragma omp parallel shared( a , b , c , chunk ) private ( i , j )
2{
3  #pragma omp for schedule( static , chunk )
4  for( i = 0 ; i < N ; i++){
5    for( j = 0 ; j < N ; j++){
6      c[ i ][ j ] = a[ i ][ j ] + b[ i ][ j ] ;
7    }
8  }
9}
```

Programa 10: Soma com OpenMPI

```
1  if( myrank == ROOT ){
2    // ROOT send part of matrix a to other process
3    for( target = 0 ; target < npes ; target++ )
4      MPI_Send( a + target * TAM * nRowsPerProcess , TAM * nRowsPerProcess
          , MPI_INT , target , target , MPI_COMM_WORLD ) ;
5  }
6  MPI_Bcast( b , TMAT , MPI_INT , 0 , MPI_COMM_WORLD ) ;
7  MPI_Status status ;
8  MPI_Recv( myA , TAM * nRowsPerProcess , MPI_INT , ROOT , myrank ,
      MPI_COMM_WORLD , &status ) ;
9  int frow = myrank * nRowsPerProcess ;
10 for( k = 0 ; k < nRowsPerProcess ; k++)
11   for( i = 0 ; i < TAM ; i++)
12     sum[ ( k + frow ) * TAM + i ] = myA[ ( k + frow ) * TAM + i ] + b[ (
          k + frow ) * TAM + i ] ;
13 MPI_Send( sum + TAM * frow , TAM * nRowsPerProcess , MPI_INT , ROOT ,
      myrank , MPI_COMM_WORLD ) ;
14 if( myrank == ROOT ){
15   // ROOT receives calculations of all process and combines in matrix sum
16   for( sender = 0 ; sender < npes ; sender++ )
17     MPI_Recv( sum + TAM * nRowsPerProcess * sender , TAM *
          nRowsPerProcess , MPI_INT , sender , sender , MPI_COMM_WORLD , &
          status ) ;
18 }
```

4.2.2 Resultados

A seguinte tabela mostra os tempos de execução para todos os programas.

Linguagem	Tempo
Chapel	0.505417
OpenMP	0.001074
OpenMPI	Maior a dois minutos

4.3 Experimento 2: Produto escalar

4.3.1 Códigos

Da mesma forma que o experimento anterior, os programas 11, 12 e 13 mostram as soluções para a linguagem Chapel, OpenMP e OpenMPI respetivamente.

Programa 11: Produto Escalar com Chapel

```
1 C = A * B ;
```

Programa 12: Produto Escalar com OpenMP

```
1 #pragma omp parallel shared( a , b , c , chunk ) private( i , j )
2 {
3   #pragma omp for schedule( static , chunk )
4   for( i = 0 ; i < N ; i++){
5     c[ i ] = a[ i ] * b[ i ] ;
6   }
7 }
```

Programa 13: Produto Escalar com OpenMPI

```
1 // Every process calculate their scalar product
2 for( i = 0 ; i < nElemPerProcess ; i++){
3   myV[ i ] = myA[ i ] * myB[ i ] ;
4 }
5 // Send scalar product to ROOT
6 MPI_Send( myV , nElemPerProcess , MPI_INT , ROOT , 0 , MPI_COMM_WORLD ) ;
7 if( myrank == ROOT ){
8   MPI_Status status ;
9   for( sender = 0 ; sender < npes ; sender++){
10    MPI_Recv( product + nElemPerProcess * sender , nElemPerProcess ,
        MPI_INT , sender , 0 , MPI_COMM_WORLD , &status ) ;
11  }
12 }
```

4.3.2 Resultados

A seguinte tabela mostra os tempos de execução para todos os programas.

Linguagem	Tempo
Chapel	0.504335
OpenMP	0.000318
OpenMPI	0.002931

5 Conclusões

Nossas conclusões finais são:

- Chapel é uma linguagem com abstrações de alto nível mais para coleções de dados não muito grandes, outras linguagens podem ter melhor desempenho
- A sintaxe de Chapel é muito mais fácil que outras linguagens e são precissadas menos linhas que outras linguagens
- Podem ser usados vários tipos de paralelismo em um mesmo programa sem nenhum problema
- Os programas em Chapel são portáveis e as variáveis de configuração dos programas também ajudam em isso ao executar um algum programa

Referências