Linguagens para Programação Paralela

Programação Concorrente

novembro de 2013

Linguagens de Programação Paralelas

- paralelismo de dados (data-parallel)
- GAS e PGAS
- orientação a processos
  - memória compartilhada
  - troca de mensagens

GAS e PGAS

- compartilhamento de dados com conhecimento de localização
- estruturas de controle induzem um estilo SPMD
  - Titanium, UPC, co-array Fortran
  - biblioteca GASNet: biblioteca para construção de sistemas GAS
- modelos de programação fortemente síncronos

- ver slides em separado

PGAS – Projetos + Recentes

- DARPA – projeto PERCS (Productive Easy-to-use Reliable Computer Systems)
- arquiteturas *NUCC*
  - arquiteturas híbridas: non-uniform cluster computing
- X10 (IBM) e Chapel (Cray)

produtividade

- desempenho
- *programmability*
- portabilidade
- robustez

Princípios

programador define explicitamente:

- paralelismo
- posicionamento de dados
- sincronização

- ênfase em abstrações que facilitariam essa tarefa

Chapel – Desiderata

- visão global (... PGAS)
- suporte a paralelismo de forma geral (dados e tarefas)
- separação de algoritmo e implementação
- facilidades "de mercado": genéricos, inferência de tipos, módulos, ...
- abstrações de dados
- desempenho
- transparência do modelo de execução
- portabilidade
- ...

Visões Global x Fragmentada

```
1 var n: int = 1000;
2 var A, B: [1..n] float;
3 forall i in 2..n-1
4   B(i) = (A(i-1) + A(i+1)) / 2;
```

```
1  var n: int = 1000;
2  var locN: int = n/numTasks;
3  var A, B: [0..locN+1] float;
4  var myItLo: int = 1;
5  var myItHi: int = locN;
6  if (iHaveLeftNeighbor) then
7    send(left, A(1));
8  else
9    myItLo = 2;
10 if (iHaveRightNeighbor) {
11   send(right, A(locN));
12   recv(right, A(locN+1));
13 } else
14   myItHi = locN-1;
15 if (iHaveLeftNeighbor) then
16   recv(left, A(0));
17 forall i in myItLo..myItHi do
18   B(i) = (A(i-1) + A(i+1)) / 2;
```

controle de espaços de endereçamento

- *locale* – representa unidade de arquitetura paralela

Chapel – Controle (1)

- básicos: loops, condicionais, select, break, continue, ...

spmd

- *forall* e *coforall* para iteração paralela
- promoção de operações escalares a arrays
- operações de redução – pré-definidas e definidas por usuário
- distribuição de domínios (espaços de índices) por *locales*
  - mapeamento de índices em locales
  - alinhamento de arrays com domínios iguais

```
forall ij in D {
  A(ij) = ...;
  ...
}
```

```
forall i in People {
  Age(i) = ...;
  ...
}
```

Chapel – Controle (2)

paralelismo de tarefas

- *cobegin* e *begin*
- variáveis de sincronização
  - semântica produtor-consumidor
- seções atômicas
  - alinhamento de arrays com domínios iguais
- controle de localidade:

```
forall Loc in Locales do
  on loc do fragmentedFunctions();
```

```
var lock: sync bool; //the sync variable
var sum: int = 0;
forall i in 1..1000000 {
   lock = true; //the sync variable is set to full
   sum += i;
   var unlock = lock; //empty the variable allowing the next process in
}
writeln(sum);
```

X10 – Modelo da Linguagem

- derivação de Java
- sublinguagem para arrays
- PGAS – reificação de localidade: *places*
- *places* (espaços de endereçamento) são fixados no início da execução do programa
- programador define distribuição de dados entre os *places*
- computação realizada por *atividades* (threads desacoplados de objetos)
  - variáveis usadas por mais que uma atividade devem ser declaradas como *final*
    - imutabilidade!

X10 – Atividades

- threads de peso leve
- assincronismo

```
final place Other = here.next();
final T t = new T();
finish async (Other){
  final T t1 = new T();
  async (t) t.val = t1;
}
```

X10 – Arrays

- *pontos* são usados para definir *regiões*
- *distribuições* mapeiam os pontos de uma região a um lugar
- regiões e distribuições não se alteram durante a vida de um array
- distribuições podem ser definidas na declaração e consultadas durante execução

X10 – Paralelismo

- *for*: iteração sequencial
- *foreach*: iteração paralela sobre pontos em uma região

```
foreach (point[i]: A.region.rank(0))
  for (point[j]: [(A.region.rank(1).low()+1):
                  A.region.rank(1).high()])
    A[i,j] = A[i,j-1] + 1;
```

- *ateach*: iteração paralela sobre pontos em uma distribuição

```
ateach ( point [i,j] : A ) A[i,j] = f(B[i,j]) ;
```

X10 – Sincronização

- `async A` disparo de atividade `A`
- `finish A` suspende criador de `A` até que `A` termine
- blocos atômicos
- blocos atômicos condicionais: `when (c) S`
- `future(P) E` dispara atividade para avaliar `E` e retorna futuro

Outro exemplo híbrido é Cilk

- C com palavras-chave extra: `cilk`, `spawn`, e `sync`
- visão da computação como um DAG
- memória compartilhada "clássica"

  > *The easiest way to deal with the anomalies of shared access is simply to avoid writing code in which one thread modifies a value that might be read or written in parallel by another thread. (do Manual, versão 5.4.6)*

- ênfase em work-stealing
- `http://supertech.csail.mit.edu/cilk/`

Cilk – Exemplos

```
#include <stdlib.h>
#include <stdio.h>
cilk int fib (int n) {
  if (n<2) return n;
  else {
    int x, y;
    x = spawn fib (n-1);
    y = spawn fib (n-2);
    sync;
    return (x+y);
  }
}
cilk int main (int argc, char *argv[]) {
  int n, result;
  n = atoi(argv[1]);
  result = spawn fib(n);
  sync;
  printf ("Result: %d\n", result);
  return 0;
}
```

Outro exemplo: Charm++

- sistema baseado em C++
- criado no final dos anos 80 (em C)
- *chares* (objetos concorrentes) formam unidades de execução
- modelo de execução orientado a eventos
  - chamadas assíncronas de métodos
  - preocupação com interoperabilidade: Converse

modelo: chamadas assíncronas entre objetos distribuídos

- objeto principal inicia a execução do programa
- chamadas a construtores lançam objetos remotos
- geração de proxies
- suporte a threads (*user-level*) e futuros
  - chamadas síncronas podem ser feitas em threads auxiliares
- sistema decide onde criar novos *chares*
- chamada a `CkExit` termina execução do programa

Entidades Charm++

- objetos sequenciais
- mensagens
  - controle de marshalling e unmarshalling, ...
- *chares*
- *chare arrays*
  - número de elementos definidos pelo programa
- *chare groups*
  - um elemento por processador
- *chare nodegroups*
  - um elemento por máquina

Arquivo de Interface

```
mainmodule Hello {
  readonly CProxy_HelloMain mainProxy;
  mainchare HelloMain {
    entry HelloMain(); // implicit CkArgMsg * as argument
    entry void PrintDone(void);
  };
  group HelloGroup {
    entry HelloGroup(void);
  };
};
```

Arquivo .h

```
#include "Hello.decl.h" // Note: not pgm.decl.h
class HelloMain: public Chare {
  public:
    HelloMain(CkArgMsg *);
    void PrintDone(void);
  private:
    int count;
};
class HelloGroup: public Group {
  public:
    HelloGroup(void);
};
```

Arquivo .C

```
#include "pgm.h"
CProxy_HelloMain mainProxy;
HelloMain::HelloMain(CkArgMsg *msg) {
  delete msg;
  count = 0;
  mainProxy=thishandle;
  CProxy_HelloGroup::ckNew(); // Create a new "HelloGroup"
}
void HelloMain::PrintDone(void) {
  count++;
  if (count == CkNumPes()) { // Wait for all group members
    CkExit();
  }
}
HelloGroup::HelloGroup(void) {
  ckout << "Hello World from processor " << CkMyPe() << endl;
  mainProxy.PrintDone();
}
#include "Hello.def.h" // Include the Charm++ object implementat
```

Passagem de Parâmetros

- métodos de marshalling (Pup) podem ser redefinidos pelo programador
- passagem de valores sequenciais é sempre por valor

Arrays

- um array de chares é uma coleção

```
array [1D] A { entry A(parameters1);
  entry void someEntry(parameters2);
};
```

- identificador CkArrayID descreve array inteiro
- identificador CkArrayIndex descreve elementos individuais
- novos elementos podem ser inseridos dinamicamente

Comunicação com Arrays

- chamada individual de método: `A[i].someEntry()`
- chamada coletiva de método: `A.someEntry()`
- chamada a método com atributo `createhere` ou `createhome` causa criação de novo elemento se não existir

Grupos de Chares

- criação de um chare por processador
- chamada individual de método:
  `g1[processador].someEntry()`
- chamada coletiva de método: `g1.someEntry()`

Migração

- todos os chares, exceto grupos, podem ser migrados dinamicamente
  - migração ocorre no momento em que chare está passivo

Estratégias

- um conjunto de estratégias de balanceamento está disponível no sistema
- outras estratégias podem ser programadas
- grupo de objetos `LBDatabase` armazena informação sobre carga em cada processador

Erlang

- características funcionais
  - listas e map
  - variáveis com atribuição única
    - imutabilidade

```
grauna:code noemi$ erl
Eshell V5.8.5  (abort with ^G)
1> L = [1, 2, 3, 4].
[1,2,3,4]
2> Myf = fun(X) -> 2*X end.
#Fun<erl_eval.6.80247286>
3> L1 = lists1:map (Myf, L).
[2,4,6,8]
4> L2 = lists1:map (fun(X) -> 3*X end, L).
[3,6,9,12]
```

- oportunidade de paralelismo

Erlang

- tipos de dados: números, átomos, strings, tuplas, listas
- listas úteis na definição de formas recursivas
  - `lista ou é lista vazia [] ou formada por cabeça e cauda [H|T]`

```
map(_, [])     -> [];
map(F, [H|T]) -> [F(H)|map(F, T)].
```

Erlang – concorrência

- processos peso *realmente* leve
- envio assíncrono
- recebimento síncrono com *patterns*

Erlang – exemplo processos

```
rpc(Pid, Request) ->
    Pid ! {self(), Request},
    receive {Pid, Response} -> Response
    end.
loop() ->
    receive
        {From, {rectangle, Width, Ht}} ->
            From ! {self(), Width * Ht}, loop();
        {From, {circle, R}} ->
            From !  {self(), 3.14159 * R * R}, loop();
        {From, Other} ->
            From ! {self(), {error,Other}}, loop()
    end.

1> Pid = spawn(fun area_server2:loop/0).
<0.33.0>
2> area_server2:rpc(Pid, {circle, 4}).
50.26544
3>
```

Erlang – outro exemplo processos

```
-module(clock).
-export([start/2, stop/0]).

start(Time, Fun) ->
    register(myclock, spawn(fun() -> tick(Time, Fun) end)).

stop() -> myclock ! stop.

tick(Time, Fun) ->
    receive stop -> void
    after Time -> Fun(), tick(Time, Fun)
    end.
```

```
1> clock:start (3000, fun() -> io:format ("tiquetaque ~p~n", [erlang:no
true
tiquetaque {1319,542626,402059}
tiquetaque {1319,542629,403038}
tiquetaque {1319,542632,404038}
tiquetaque {1319,542635,405039}
2> clock:stop().
stop
```

Erlang: pmap (1)

```
pmap(F, L) ->
    S = self(),
    Ref = erlang:make_ref(),  %%   we'll match on this later
    Pids = map(fun(I) ->
                       spawn(fun() -> do_f(S, Ref, F, I) end)
               end, L),
    %% gather the results
    gather(Pids, Ref).
do_f(Parent, Ref, F, I) ->
    Parent ! {self(), Ref, (catch F(I))}.
```

Erlang: pmap (2)

```
gather([Pid|T], Ref) ->
    receive
        {Pid, Ref, Ret} -> [Ret|gather(T, Ref)]
    end;
gather([], _) ->
    [].
```

Erlang: mapreduce

- casamento com características funcionais
  - versão básica na distribuição oficial

Erlang: mapreduce

```
%% F1(Pid, X) -> sends {Key,Val} messages to Pid
%% F2(Key, [Val], AccIn) -> AccOut

mapreduce(F1, F2, Acc0, L) ->
    S = self(),
    Pid = spawn(fun() -> reduce(S, F1, F2, Acc0, L) end),
    receive
        {Pid, Result} ->
            Result
    end.
```

Erlang: mapreduce (2)

```
reduce(Parent, F1, F2, Acc0, L) ->
    process_flag(trap_exit, true),
    ReducePid = self(),
    %% Create the Map processes
    %%   One for each element X in L
    foreach(fun(X) ->
                    spawn_link(fun() -> do_job(ReducePid, F1, X)
            end, L),
    N = length(L),
    %% make a dictionary to store the Keys
    Dict0 = dict:new(),
    %% Wait for N Map processes to terminate
    Dict1 = collect_replies(N, Dict0),
    Acc = dict:fold(F2, Acc0, Dict1),
    Parent ! {self(), Acc}.
```

Erlang: mapreduce (3)

```
collect_replies(0, Dict) ->
    Dict;
collect_replies(N, Dict) ->
    receive
        {Key, Val} ->
            case dict:is_key(Key, Dict) of
                true ->
                    Dict1 = dict:append(Key, Val, Dict),
                    collect_replies(N, Dict1);
                false ->
                    Dict1 = dict:store(Key,[Val], Dict),
                    collect_replies(N, Dict1)
            end;
        {'EXIT', _,  _Why} ->
            collect_replies(N-1, Dict)
    end.

do_job(ReducePid, F, X) ->
    F(ReducePid, X).
```

Referências

Chapel B. Chamberlain e outros. Parallel Programmability and the Chapel Language. International Journal of High Performance Computing Applications, August 2007, 21(3): 291-312.

X10 P. Charles e outros. X10: An Object-oriented approach to non-uniform Clustered Computing. OOPSLA 2005.

Cilk M. Frigo, C. Leiserson, and K. Randall. The implementation of the Cilk-5 multithreaded language. PLDI '98, 212-223.

Charm++ L. V. Kale and S. Krishnan, Charm++: Parallel Programming with Message-Driven Objects, Book Chapter in "Parallel Programming using C++", MIT Press, 1996. pp 175-213.

erlang J. Armstrong. Programming Erlang – Software for a Concurrent World. Pragmatic Bookshelf. 2007.