Revisão

Introdução

Modelos de programação

Maquinas/arquiteturas paralelas

Sincronização

Passagem de mensagens

- Conexão entre processos
- Envio e recepção de msgs
- RPC e Rendezvous
- Projeto de progs que passam msgs

Hoje

Construções e técnicas de prog //

- Spin locks
- Barreiras
- Reduções
- Broadcasts
- Task queues

Instruções atômicas disponíveis em hardware

```
int TestAndSet(int x)
{
    int temp = x;
    x = 1;
    return temp;
}

Utilizacao:

shared boolean lock = false;
repeat
  boolean key = TestAndSet(lock);
until !key
critical region
lock = false;
```

Instruções atômicas disponíveis em hardware

```
procedure Swap (a : integer; b : integer);
temp : integer;
  begin
    temp := a;
    a := b;
    b := temp;
end;
```

Instruções atômicas disponíveis em hardware

```
function Fetch_and_Store (a : integer;
                          b : integer)
temp : integer;
  begin
    temp := a;
    a := b;
    return temp;
end;
```

Instruções atômicas disponíveis em hardware

```
function Compare_and_Swap(addr: address;
                          old: integer;
                          new: integer)
  compare the value at addr to old
  if they are equal then
    set value at addr to new
    return success
  else
    return failure
```

Artigo ijpp98, by Bianchini *et al.*

- Spin locks
- Barreiras
- Reduções

Broadcasts

- Linear
- Logarítmico dirigido por consumidor
- Logarítmico dirigido pelo produtor

Broadcast Linear

```
// Proc 0 is the fixed producer of the data.
// Ind stores the produced item's order number.

procedure bcast_linear
  if pid != 0
      buffer := ind mod NUM_BUFFS
      repeat until broad[0][buffer].index = ind
      // Consume data fetching it from the producer
      fetch_and_decrement(&broad[0][buffer].counter)
      ind := ind + 1
  else
      ind := ind + 1
      buffer := ind mod NUM_BUFFS
      repeat until broad[0][buffer].counter = 0
      // Produce new data item and
      //   store it in broad[0][buffer].data
      broad[0][buffer].counter := P-1
      // Fence
      broad[0][buffer].index := ind
```

Broadcast Logarítmico – Consumidor

```
// Tree is set up during init. Proc 0 is at the root.
// Parent stores the id of the parent in the tree.
// Children stores the num of children each proc has.
// Ind stores the produced item's order number.

procedure move_data
  repeat until broad[parent][buffer].index = ind
  repeat until broad[pid][buffer].counter = 0
  // Copy data from broad[parent][buffer].data
  //   to broad[pid][buffer].data
  broad[pid][buffer].counter := children
  // Fence
  broad[pid][buffer].index := ind
  fetch_and_decrement(&broad[parent][buffer].counter)

procedure bcast_consumer
  if pid != 0
      buffer := ind mod NUM_BUFFS
      move_data
      // Consume the new data
      ind := ind + 1
  else
      ind := ind + 1
      buffer := ind mod NUM_BUFFS
      repeat until broad[0][buffer].counter = 0
      // Produce new data item and
      //   store it in broad[0][buffer].data
      broad[0][buffer].counter := children
      // Fence
      broad[0][buffer].index := ind
```

Broadcast Logarítmico – Produtor

```
// Tree is set up during init. Proc 0 is at the root.
// Ind stores the produced item's order number.
// Children stores the pid of each child processor.

procedure move_data (child : integer)
  repeat until broad[pid][buffer].produced = 1
  repeat until broad[child][buffer].consumed = 1
  // Copy data from broad[pid][buffer].data to
  //   broad[child][buffer].data
  broad[child][buffer].consumed := 0
  // Fence
  broad[child][buffer].produced := 1

procedure bcast_producer
  for i := 0 until i = DEGREE or children[i] = -1
      if children[i] != -1  move_data (children[i])
  if pid != 0
      buffer := ind mod NUM_BUFFS
      repeat until broad[pid][buffer].produced = 1
      // Consume the new data
      broad[pid][buffer].produced := 0
      broad[pid][buffer].consumed := 1
      ind := ind + 1
  else
      ind := ind + 1
      buffer := ind mod NUM_BUFFS
      // Produce new data item and
      //   store it in broad[0][buffer].data
      // Fence
      broad[0][buffer].produced := 1
```

Task Queues

- Centralizada
- Distribuída

Task Queue Centralizada

```
shared queue : integer := 0
shared Barrier : barrier

procedure cent_task_queue
  my_task : integer
  round : integer := 0
  group_tasks : integer := TOT_TASKS / P / 2
  while round != TOT_ROUNDS
      my_task := fetch_and_increment (&queue)
      my_task := my_task * group_tasks
      if my_task > round * TOT_TASKS
          round := round + 1
          BARRIER(Barrier)
      my_task := my_task mod TOT_TASKS
      // Code that uses my_task work descriptor
```

Task Queue Distribuída

```
procedure dist_task_queue
  local_tasks : integer := TOT_TASKS / P
  for round := 1 to TOT_ROUNDS-1
    // Perform local tasks
    do
      LOCK(Lock[pid])
        if queue[pid] < round * local_tasks
          my_task := queue[pid]
          queue[pid] := queue[pid] + 1
        else
          my_task := -1
      UNLOCK(Lock[pid])
      if my_task != -1
        // Code that uses my_task work descriptor
    until my_task == -1
    // Search for remote tasks in 1/4 of the procs
    for i := 1 to P-1 step 4
      remote := (pid + i) % P  // Find next
      while queue[remote] < round*local_tasks-BEHIND
        LOCK(Lock[remote])
          if queue[remote] < round*local_tasks-BEHIND+1
          // Remote proc is indeed late
            my_task := queue[remote]
            queue[remote] := queue[remote] + 1
          else
            my_task := -1
        UNLOCK(Lock[remote])
        if my_task != -1
          // Code that uses my_task work descriptor
    BARRIER(Barrier)
```

Próxima Aula

Andrews, G. and Schneider, F. "Concepts and Notations for Concurrent Programming", Computing Surveys, March 1983

Bal, H. et al. "Programming Languages for Distributed Computing", Computing Surveys, Sept 1989

Aluno responsável por esclarecer dúvidas