Programação Concorrente e Paralela

Multithreading na camada de aplicação

Noemi Rodriguez

2016

PUC RIO

DEPARTAMENTO DE INFORMÁTICA PUC-RIO

voltando ao PCAM

- particionamento
- comunicação
- aglomeração
- mapeamento

voltando ao PCAM

- particionamento
- comunicação
- aglomeração
- mapeamento

- frequentemente resulta em parte um tanto artficial ou complicada do código de aplicação científica

Aglomeração

qual seu papel em arquiteturas de memória compartilhada?

- threads de SO geram sobrecarga
- comunicação e compartilhamento de caches geram custos
- ... tipicamente não é eficiente termos um número de threads de SO muito maior que o número de núcleos disponíveis!

Distribuição de trabalho

para tarefas descritas por dados

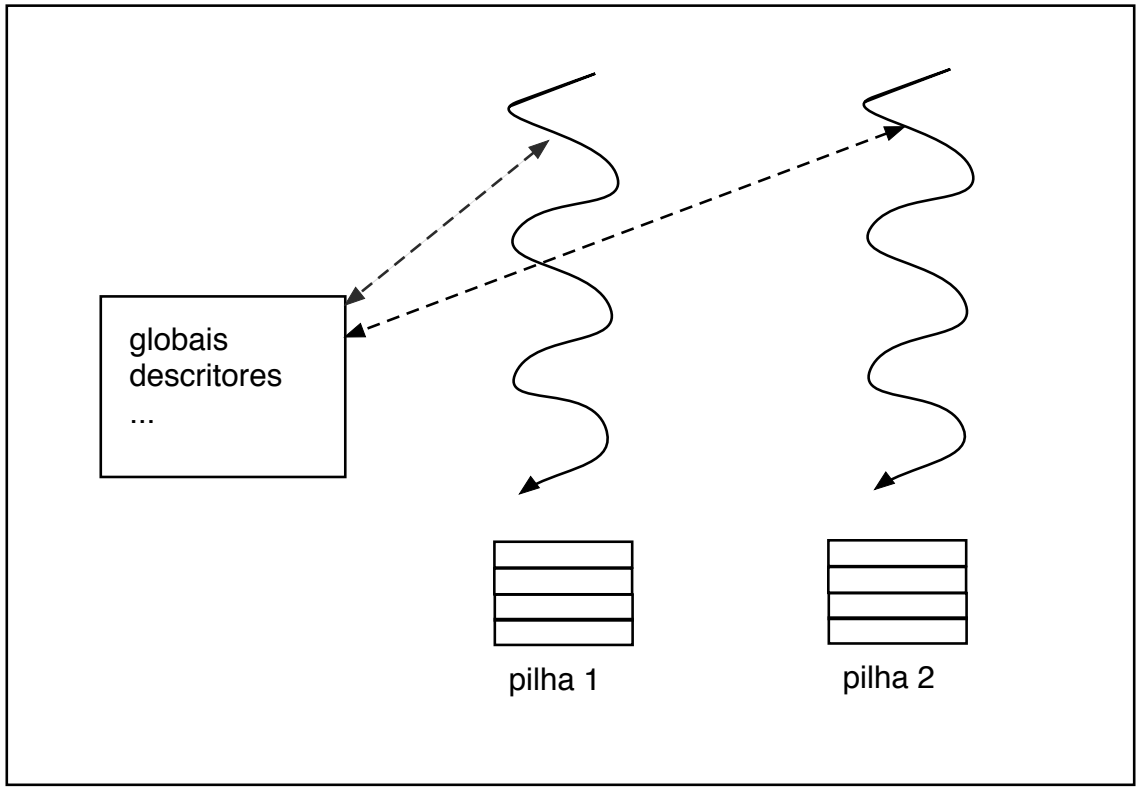
- distribuição de trabalho sob demanda
  - cada thread repetidamente busca dados para trabalhar

- e se cada tarefa demanda código diferente?
- e se tarefas podem ter longa duração mas não tanta demanda de CPU?

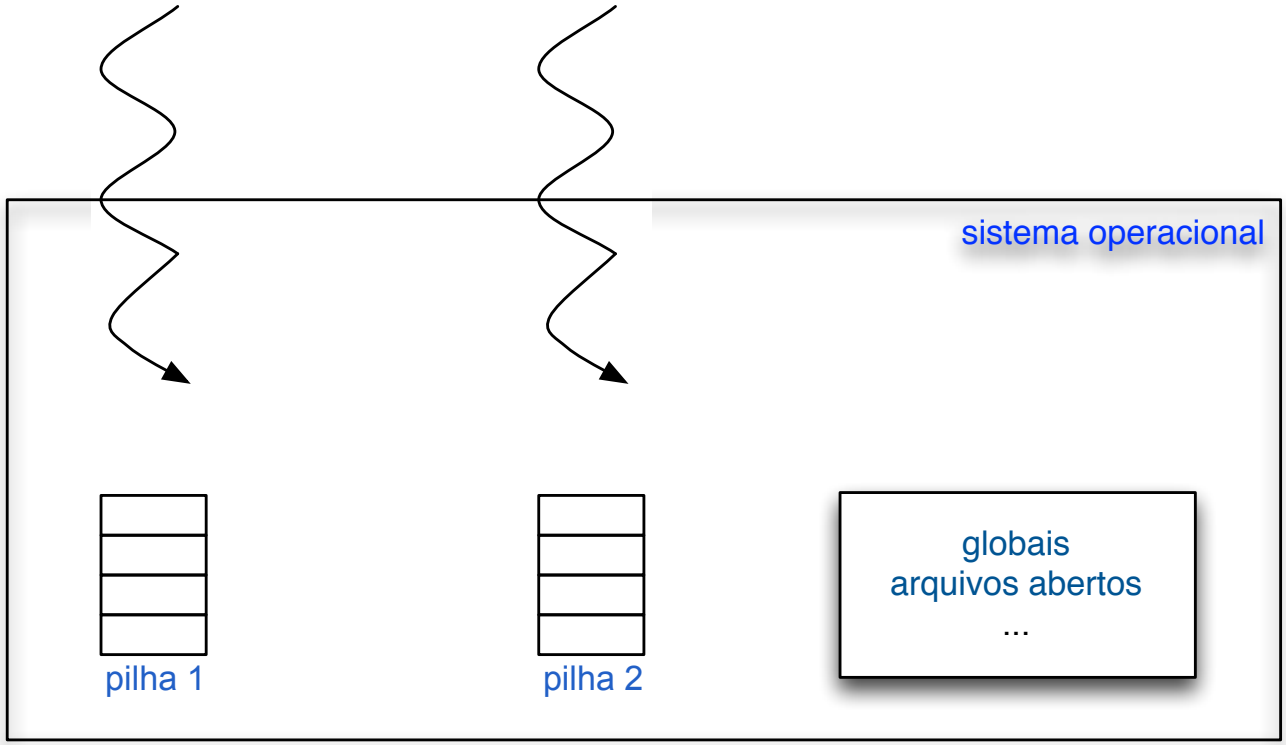
Multithreading leve

- implementado em nível usuário (bibliotecas ou aplicação)
- muitas threads leves executadas por poucas threads de SO
  - um pouco como nossas threads que buscam dados quando livres...

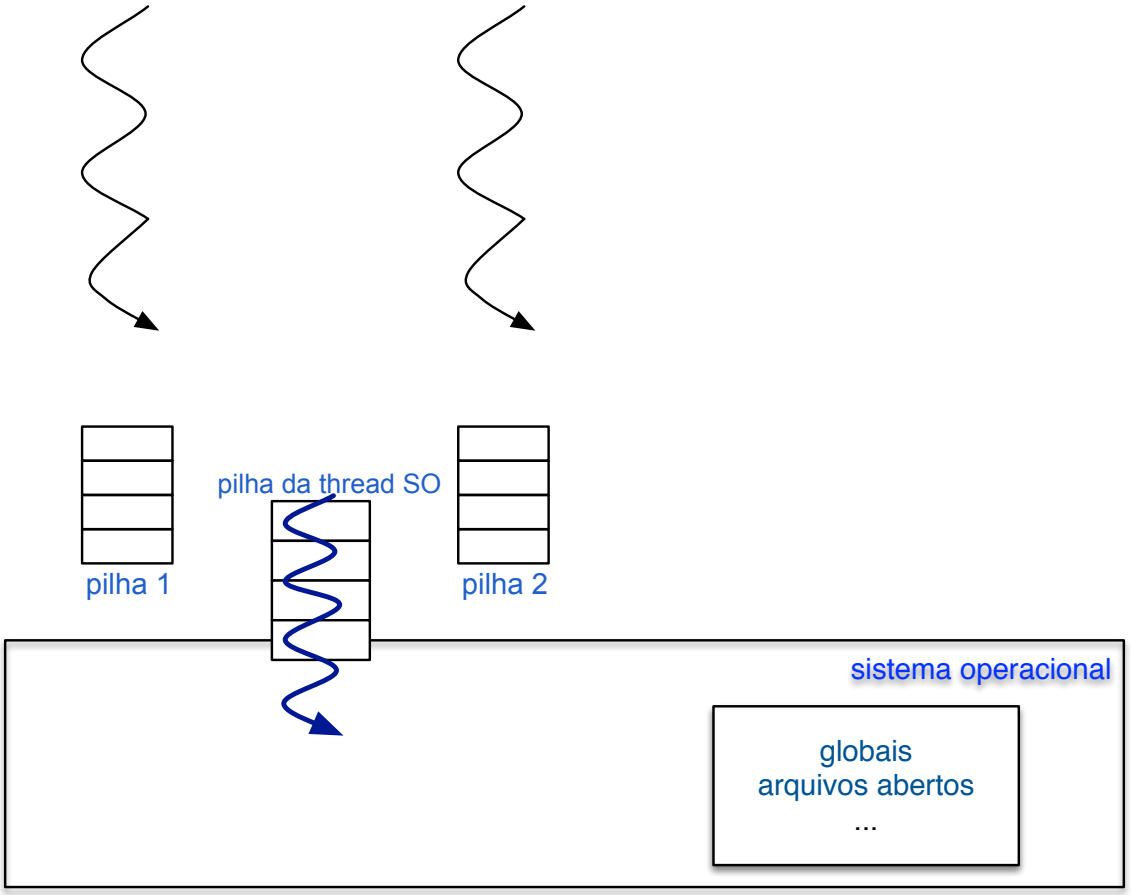
Multithreading



Threads de sistema operacional



Threads de nível de aplicação



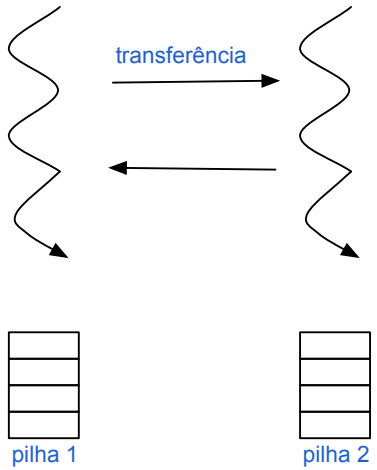
Multithreading de nível de aplicação

- controle transferido entre linhas de execução de uma mesma aplicação
- tipicamente: multithreading cooperativo

ausência de preempção

Transferência de Controle

- como ambiente pode prover suporte para cooperação?
- uma maneira bastante tradicional é através de *corotinas*
  - simétricas
  - assimétricas



Corotinas

modelos

simétricas: transferência sempre tem destinatário explicito

- `transfer`

assimétricas: hierarquia

- `yield`
- `resume`

Corotinas simétricas: Modula-2

```
MODULE M;
    CONST
      WKSIZE = 512;
    VAR
      wkspA, wkspB : ARRAY [1..WKSIZE] OF BYTE;
      main, cA, cB : ADDRESS;
      x : ADDRESS;      (* a shared context variable *)
    PROCEDURE A;                    PROCEDURE B;
    BEGIN                           BEGIN
      LOOP                            LOOP
        ...                             ...
        TRANSFER(x,x);                  TRANSFER(x,x);
      END;                            END;
    END A;                          END B;
  BEGIN (* M *)
      (* create two processes out of procedure A and B *)
    NEWPROCESS( A, ADR(wkspA), WKSIZE, cA );
    NEWPROCESS( B, ADR(wkspB), WKSIZE, cB );
    x := cB;
    TRANSFER(main,cA);
  END M;
```

Transferência – exemplo implementação em asm

```
icoro_transfer:
  push  %ebp
  mov   %esp, %ebp
  mov   8(%ebp), %ecx     # obtem o descritor da corotina ativa
  mov   12(%ebp), %edx    # obtem o descritor da corotina a (re)ativar
  mov   16(%ebp), %eax    # valor para a corotina
  push  %ebx              # salva os registradores da corotina ativa
  push  %edi              # %eax, %ecx e %edx nÃ£o precisam ser salvos!
  push  %esi
  push  %ebp
  call transfer           # coloca o endereço (%eip) do ponto de retorno

/* ponto de retorno quando o controle voltar */
  pop   %ebp              # restaura registradores salvos
  pop   %esi
  pop   %edi
  pop   %ebx
  mov   %ebp, %esp
  pop   %ebp
  ret
```

Transferência

```
/* transfere o controle */
transfer:
  mov   %esp, (%ecx)     # salva o topo da pilha corrente
  mov  (%edx), %esp      # agora a pilha é a da corotina a reativar!
  ret                    # retorna para o endereço de retorno na nova pi
```

- se LP não tem alguma construção apropriada, precisamos ir ao nível da linguagem de máquina.

Corotinas assimétricas: Lua

```
function ping ()
  for i = 1,10 do
    print ("ping", i)
    coroutine.yield()
  end
  print("fim")
end

co = coroutine.create (ping)
while coroutine.resume(co) do
  print("pong")
end
```

Corotinas assimétricas

Usos

- iteradores
- filtros
- multithreading não preemptivo

Escalonamento cooperativo

```
function create_task(f)  -- cria uma tarefa
  local co = coroutine.create(f)
  table.insert(tasks, co)
end
function dispatcher()  -- escalonador de tarefas
  local i = 1
  while true do
    if tasks[i] == nil then
      if tasks[1] == nil then break end
      i= 1
    end
    local status = coroutine.resume(tasks[i])
    if status==false then
      print ("acabou uma tarefa")
      table.remove(tasks, i)
    else
      i= i + 1
    end
  end
  print ("acabaram as tarefas")
end
```

Escalonamento cooperativo

```
tasks = {}
...
function ping ()
  i = 0
  for i = 1, 10 do
     print ("ping --", i)
     i = i+1
     coroutine.yield()
  end
end
function pong ()
  i = 0
  for i = 1, 10 do
     print ("pong --", i)
     i = i+1
     coroutine.yield()
  end
end
create_task (ping)
create_task (pong)
dispatcher()
```

Escalonamento cooperativo

- pontos de possíveis entrelaçamento ficam explícitos no código
- por outro lado, necessidade de transferência explícita

- tipicamente transferência é encapsulada em chamada a biblioteca

Escalonamento cooperativo – vantagens e limitações

- sobrecarga de criação de threads é muito menor que de threads de SO
- dificuldades com condições de corrida também menores
- porém... todo o processamento é feito em *uma* thread de SO

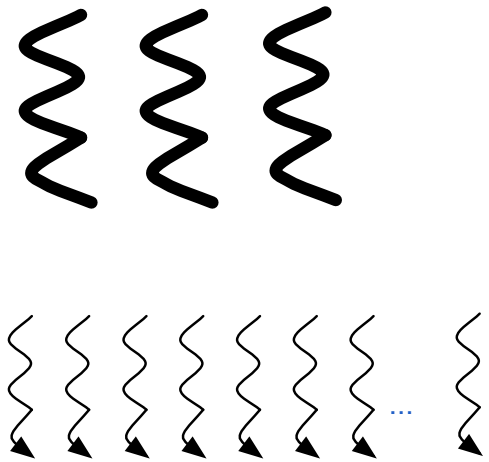
- e em máquinas multiCPU?

Escalonamento com múltiplas threads de SO

- modelos n para m: n threads de SO e m (>>> n) de aplicação

- escalabilidade

Misturando C e Lua – LuaProc



- biblioteca em C cria threads de sistema operacional
- cada thread continuamente busca tarefas Lua para execução

LuaProc — loop de threads trabalhadoras

```
/* main worker loop */
while ( TRUE ) {
  ...
  pthread_mutex_lock( &mutex_sched );
  ...
  /* remove lua process from the ready queue */
  lp = list_remove( &ready_lp_list );
  pthread_mutex_unlock( &mutex_sched );
  /* execute the lua code specified in the lua process struct */
  procstat = luaproc_resume( luaproc_get_state( lp ), NULL,
                             luaproc_get_numargs( lp ));
  ...
  /* has the lua process sucessfully finished its execution? */
  ...
  /* has the lua process yielded? */
  else if ( procstat == LUA_YIELD ) {
    /* yield attempting to send a message */
    if ( luaproc_get_status( lp ) == LUAPROC_STATUS_BLOCKED_SEND ) {
        ...
    }
    /* yield attempting to receive a message */
    else if ( luaproc_get_status( lp ) == LUAPROC_STATUS_BLOCKED_RECV ) {
        ...
    }

    /* yield on explicit coroutine.yield call */
    else {
      ...
    }
  }
```

LuaProc — loop de threads trabalhadoras

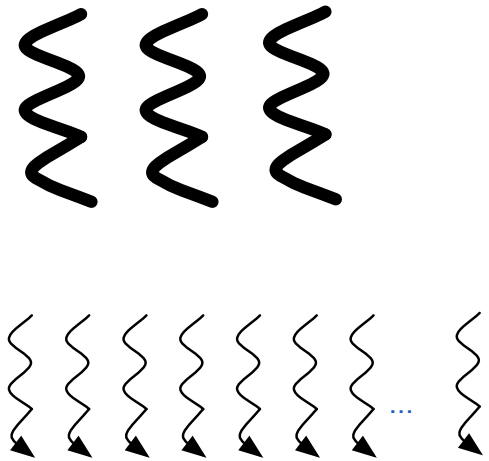
```
/* main worker loop */
  ...
  /* has the lua process yielded? */
  else if ( procstat == LUA_YIELD ) {
    /* yield attempting to send a message */
    if ( luaproc_get_status( lp ) == LUAPROC_STATUS_BLOCKED_SEND ) {
      luaproc_queue_sender( lp );  /* queue lua process on channel */
      ...
    }
    /* yield attempting to receive a message */
    else if ( luaproc_get_status( lp ) == LUAPROC_STATUS_BLOCKED_RECV ) {
      luaproc_queue_receiver( lp );  /* queue lua process on channel */
      ...
    }

    /* yield on explicit coroutine.yield call */
    else {
      /* re-insert the job at the end of the ready process queue */
      pthread_mutex_lock( &mutex_sched );
      list_insert( &ready_lp_list, lp );
      pthread_mutex_unlock( &mutex_sched );
    }
  }
```

Misturando C e Lua – LuaProc

```
static int luaproc_send( lua_State *L ) {
  ...
  chan = channel_locked_get( chname );
  /* remove first lua process, if any, from channel's receive list */
  dstlp = list_remove( &chan->recv );
  if ( dstlp != NULL ) { /* found a receiver? */
    /* try to move values between lua states' stacks */
    ...
    sched_queue_proc( dstlp );
    /* unlock channel access */
    }
  }
  else {
    self = luaproc_getself( L );
    if ( self != NULL ) {
      self->status = LUAPROC_STATUS_BLOCKED_SEND;
      self->chan   = chan; }
    /* yield. channel will be unlocked by the scheduler */
    return lua_yield( L, lua_gettop( L ));
  }
}
```

Misturando C e Lua – LuaProc



- facilidade Lua: criação de *estados* independentes
  - cada estado tem seu próprio conjunto de variáveis globais
- tarefas Lua são criadas cada uma em um estado independente
- tarefas só se comunicam por troca de mensagens

LuaProc – estados independentes e troca de mensagens

- `yield` encapsulados em envios e recebimentos e em IO
- modelo: C/Lua/C (embedding/extending)

Executores em Java — pools de threads

Executores em Java — pools de threads

```
1  public class MatrixTask {
2    static ExecutorService exec = Executors.newCachedThreadPool();
3    ...
4    static Matrix add(Matrix a, Matrix b) throws ExecutionException {
5      int n = a.getDim();
6      Matrix c = new Matrix(n);
7      Future<?> future = exec.submit(new AddTask(a, b, c));
8      future.get();
9      return c;
10   }
11   static class AddTask implements Runnable {
12     Matrix a, b, c;
13     public AddTask(Matrix myA, Matrix myB, Matrix myC) {
14       a = myA; b = myB; c = myC;
15     }
```

Executores em Java — pools de threads

```
11    static class AddTask implements Runnable {
12      Matrix a, b, c;
13      public AddTask(Matrix myA, Matrix myB, Matrix myC) {
14        a = myA; b = myB; c = myC;
15      }
16      public void run() {
17        try {
18          int n = a.getDim();
19          if (n == 1) {
20            c.set(0, 0, a.get(0,0) + b.get(0,0));
21          } else {
22            Matrix[][] aa = a.split(), bb = b.split(), cc = c.split();
23            Future<?>[][] future = (Future<?>[][]) new Future[2][2];
24            for (int i = 0; i < 2; i++)
25              for (int j = 0; j < 2; j++)
26                future[i][j] =
27                  exec.submit(new AddTask(aa[i][j], bb[i][j], cc[i][j]));
28            for (int i = 0; i < 2; i++)
29              for (int j = 0; j < 2; j++)
30                future[i][j].get();
31          }
32        } catch (Exception ex) {
33          ex.printStackTrace();
34        }
35      }
36    }
37  }
```

Figure 16.4 The MatrixTask class: parallel matrix addition.

Futuros e Promessas

- Barbara Liskov; Liuba Shrira (1988). *Promises: Linguistic Support for Efficient Asynchronous Procedure Calls in Distributed Systems*. Proceedings of the SIGPLAN '88 Conference on Programming Language Design and Implementation; Atlanta, Georgia, United States, pp. 260¿267. Also published in ACM SIGPLAN Notices, Volume 23, Issue 7, July 1988.

Futuros em ProActive

Pools de threads e filas de tarefas

- como implementar a fila?
  - LuaProc: fila única com locks
  - outras propostas: filas individuais e *workstealing*

Work stealing

```
1  public class WorkStealingThread {
2    DEQueue[] queue;
3    int me;
4    Random random;
5    public WorkStealingThread(DEQueue[] myQueue) {
6      queue = myQueue;
7      random = new Random();
8    }
9    public void run() {
10     int me = ThreadID.get();
11     Runnable task = queue[me].popBottom();
12     while (true) {
13       while (task != null) {
14         task.run();
15         task = queue[me].popBottom();
16       }
17       while (task == null) {
18         Thread.yield();
19         int victim = random.nextInt(queue.length);
20         if (!queue[victim].isEmpty()) {
21           task = queue[victim].popTop();
22         }
23       }
24     }
25   }
26 }
```

- uso de técnicas *lock-free* para otimizar acessos

Work stealing - filas com retirada nas duas "pontas"

```
1  public class BDEQueue {
2    Runnable[] tasks;
3    volatile int bottom;
4    AtomicStampedReference<Integer> top;
5    public BDEQueue(int capacity) {
6      tasks = new Runnable[capacity];
7      top = new AtomicStampedReference<Integer>(0, 0);
8      bottom = 0;
9    }
10   public void pushBottom(Runnable r){
11     tasks[bottom] = r;
12     bottom++;
13   }
14   // called by thieves to determine whether to try to steal
15   boolean isEmpty() {
16     return (top.getReference() < bottom);
17   }
18  }
19 }
```

Work stealing - filas com retirada nas duas "pontas"

```
1   public Runnable popTop() {
2     int[] stamp = new int[1];
3     int oldTop = top.get(stamp), newTop = oldTop + 1;
4     int oldStamp = stamp[0], newStamp = oldStamp + 1;
5     if (bottom <= oldTop)
6       return null;
7     Runnable r = tasks[oldTop];
8     if (top.compareAndSet(oldTop, newTop, oldStamp, newStamp))
9       return r;
10    return null;
11  }
12  public Runnable popBottom() {
13    if (bottom == 0)
14      return null;
15    bottom--;
16    Runnable r = tasks[bottom];
17    int[] stamp = new int[1];
18    int oldTop = top.get(stamp), newTop = 0;
19    int oldStamp = stamp[0], newStamp = oldStamp + 1;
20    if (bottom > oldTop)
21      return r;
22    if (bottom == oldTop) {
23      bottom = 0;
24      if (top.compareAndSet(oldTop, newTop, oldStamp, newStamp))
25        return r;
26      }
27    top.set(newTop,newStamp);
28    return null;
29  }
```