OpenMP Tasks

Leitura Obrigatória:
http://goo.gl/3slUxb

A partir do OpenMP 3.0

- Grande novidade: introdução do modelo de tarefas (tasks)
- Antes (OpenMP 2.5), as maneiras de distribuir trabalho entre tarefas (worksharing) era majoritariamente baseada em diretivas voltadas para aplicações baseadas em arrays
  - Exemplo: parallel for
  - Não havia uma forma padronizada de expressar e explorar paralelismo não estruturado de forma eficiente e elegante

Exemplo: Paralelismo não estruturado

- Queremos atravessar uma lista ligada de forma paralela (items da lista serão processados em paralelo).
- Uma forma de fazer com OpenMP antes de tasks:

```
1  p = listhead; num_elements=0;
2  while (p) {
3    list_item[num_elements++]=p;
4    p=next(p);
5  }
6
7  #pragma omp parallel for
8    for (int i=0; i < num_elements; i++)
9      process(list_item[i]);
```

- Vantagem:
  - Funciona
- Desvantagem:
  - Ineficiente
    - Custo de criação e atribuição dos elementos de um array “inútil”
  - Deselegante
- Outra forma de fazer:

```
1 #pragma omp parallel private (p)
2 {
3   p = listhead;
4   while(p) {
5     #pragma omp single nowait
6       process(p);
7     p = next(p);
8   }
9 }
```

- Vantagens:
  - Funciona
  - Mais elegante que o anterior
- Desvantagem:
  - Ineficiente
    - Custo do omp single nowait
    - Cada thread percorre toda a lista
  - Não intuitivo

Solução: OpenMP Tasks (>= 3.0)

- Tasks
  - concebidas para expressar paralelismo não estruturado
  - Elegante e eficiente
  - Mantém os princípios de OpenMP
    - Paralelização incremental
    - Consistência sequencial
  - Mudança de foco: de threads para tasks

Tasks x Threads

- Conceitualmente, pode-se dizer que OpenMP já possuía tasks
  - Cada parte de um programa OpenMP já faz parte de uma ou outra tarefa
  - O mecanismo de tasks introduz a possibilidade de criar tasks explicitamente
  - Uma diretiva como parallel criava uma thread executando implicitament uma task durante a região paralela
- Com o conceito explícito de tasks, uma task pode ser executada por diferente threads, se o programador assim o especificar.

Tasks

- Definição:
  - “a specific instance of executable code and its data environment”
- Uma task explícita pode ser executada por qualquer thread no time atual, em paralelo com outras tarefas, e a execução pode ser imediata ou adiada até um momento posterior.
- Uma tarefa que está sendo executada atualmente por uma thread é chamada de tarefa corrente (atual).

Exemplo

```
1 #pragma omp task shared(tot), private(st), firstprivate(p)
2 {
3    st = process(p);
4    #pragma omp critical
5       tot += st;
6 }
```

- Variáveis podem ser shared, private ou firstprivate
  - Shared: é a variável anteriormente definida, compartilhada por todos
  - Private: é criada uma nova variável privada, sem inicialização
  - Firstprivate: é criada uma nova variável privada, inicializada para o valor que tinha antes da task

Depth-first tree traversal, possible preorder

```
1  void traverse(binarytree *p, int preorder) {
2    #pragma omp task if(!preorder)
3      process(p);
4    if (p->left) {
5      #pragma omp task
6        traverse(p->left, preorder);
7    }
8    if (p->right) {
9      #pragma omp task
10       traverse(p->right, preorder);
11   }
12 }
```

- Se uma cláusula if estiver presente na construção da task e o valor da expressão avaliar para false, a thread que a encontrou deve suspender sua própria execução e iniciar imediatamente a task encontrada. A task suspensa só vai reiniciar até que a task iniciada seja concluída.

Depth-first tree traversal, postorder

```
1  int postorder(binarytree *p) {
2    int l, r;
3    l = r = 0;
4    if (p->left) {
5      #pragma omp task shared(l)
6        l = postorder(p->left);
7    }
8    if (p->right) {
9      #pragma omp task shared(r)
10       r = postorder(p->right);
11   }
12   #pragma omp taskwait
13   return l + r + process(p);
14 }
```

- taskwait faz com que a execução da task corrente seja suspensa até que todas as suas tasks filhas sejam completadas. A espera é feita somente pelas tasks filhas, e não por suas descendentes (se houver).

Tied tasks

- Once a thread in the current team starts execution of a task, the two become tied together: the same thread will execute the task region from beginning to end.
- This does not imply that execution is continuous. Thread may suspend execution of a task region at a task scheduling point, to resume it at a later time. In tied tasks, task scheduling points may only occur at task, taskwait, explicit or implicit barrier constructs, and upon completion of the task. When a thread suspends the current task,may perform a task switch, i.e., resume execution of a task previously suspended, or start execution of a new task.

Untied tasks

- Most of the aforementioned restrictions are lifted for untied tasks (indicated by the untied clause on the task construct). Any thread in the team reaching a task scheduling point may resume any suspended untied task, or start any new untied task. Also, task scheduling points may in principle occur at any point in an untied task region.
- Beware of code that depends on thread_ids or anything that assumes that a tasks begins and ends its execution in the same thread!
- A untied task may be preempted anywhere, even inside a critical region
  - Usage of `critical` constructs in an untied task is discouraged

Exemplo inicial com tasks

```
1  #pragma omp parallel
2  {
3    /* a single thread traverses the list */
4    #pragma omp single
5    {
6      p = listhead;
7      while(p) {
8        /* create a task for each element */
9        #pragma omp task
10         process(p)
11       p=next(p);
12     }
13   }
14 }
```