OpenMP: *Tasks*

Paulo Sérgio Lopes de Souza
pssouza@icmc.usp.br

Universidade de São Paulo / ICMC / SSC – São Carlos
Laboratório de Sistemas Distribuídos e Programação Concorrente

CNPq Conselho Nacional de Desenvolvimento Científico e Tecnológico
CAPES
FAPESP

Relembrando da 1ª aula de OMP: Diretiva section

- Diretiva ***sections***
  - Divide computações distintas em *threads* distintas (paralelismo funcional)

```
#pragma omp sections [clause[ [, ] clause] ...]
    {
    [#pragma omp section]
        structured-block
    [#pragma omp section
structured-block]
...
    }
clause:
      private(list)
      firstprivate(list)
      lastprivate(list)
      reduction(reduction-identifier: list)
      nowait
```

```
1       #pragma omp parallel
2       {
3           #pragma omp sections
4           {
5               #pragma omp section
6               {
7                   taskA();
8               }
9               #pragma omp section
10              {
11                  taskB();
12              }
13              #pragma omp section
14              {
15                  taskC();
16              }
17          }
18      }
```

Grama et al. (2003)

Tarefas

- Tarefas:
  - Introduzidas em 2008 como a maior extensão do OpenMP 3.0
  - Permitem paralelizar algoritmos com uma execução irregular e dependente
  - Exemplo com um *loop-while* com porções independentes no corpo do *loop*
    - Teria que ser transformado em um *loop-for* e adaptado para o ***#pragma omp for***

- Tarefas permitem uma solução elegante
  - Um sistema de fila atribui dinamicamente a atribuição de *threads* para porções de trabalho que precisam ser computados
  - *Threads* consomem o trabalho da fila até o fim

```
#pragma omp task [clause[[,] clause] ...] new-line
    structured-block
```

- Uma tarefa OpenMP é um bloco de código contido em uma região paralela que pode ser executado simultaneamente com outras tarefas na mesma região paralela
  - Cada tarefa define seu ambiente de dados de acordo com as cláusulas *private*, *firstprivate* e *shared*
  - Diferente de construtores de compartilhamento
    - Como o ***#pragma omp for*** que divide o trabalho sobre as *threads*

Tarefas

- Cada *thread* pode executar diferentes cargas de trabalho que não são conhecidas antes
  - Por isso precisa de um tratamento diferente!

- Tarefas:
  - São geradas apenas uma vez: usamos a diretiva ***single*** ou eventualmente a *master*
  - Não são executadas necessariamente onde elas são definidas no código
  - Garante-se a execução em pontos do código específicos e bem definidos
    - Com controle explícito do programador

- Uma *thread* gera as tarefas, enquanto outras *threads* executam essas tarefas
  - Assim que elas (as threads)se tornam disponíveis para execução
  - A *thread* geradora das tarefas também pode executar as tarefas!
    - Depois que terminar de gerar as tarefas

- Cláusulas diferentes:
  - *if: geração das tarefas é suspensa*
  - *final: execuções tarefas feitas sequencialmente pela thread geradora*
  - *untied*: qualquer *thread* pode finalizar a tarefa
  - *mergable*: usa mesmo ambiente de dados

#pragma omp task *[clause[[,]clause] ...]*
structured-block

clause:

if(*scalar-expression*)
final(*scalar-expression*)
untied
default(**shared** | **none**)
mergeable
private(*list*)
firstprivate(*list*)
shared(*list*)
4.0 **depend**(*dependence-type*: *list*)

OpenMP API 4.0 C/C++ (2013)

Imprimindo um Palíndromo

- Algoritmo para imprimir 1x ou “***race car*** ” ou “***car race*** ”, sem preferência por uma das ordens

Sequencial

```
int main(int argc, char *argv[ ])
{
        printf("race ");
        printf("car ");
        printf("\n");

        return(0);
}
```

Usando apenas #omp parallel

```
int main(int argc, char *argv[])
{
   #pragma omp parallel num_threads(2)
   {
        printf("race ");
        printf("car ");
   } // End of parallel region
   printf("\n");

   return(0);

}
```

Usando #omp parallel e single

```
int main(int argc, char *argv[])
{
  #pragma omp parallel num_threads(2)
  {
    #pragma omp single
    {
        printf("race ");
        printf("car ");
    }
  } // End of parallel region
  printf("\n");

  return(0);
}
```

Van der Pas et al. (2017)

Paralelizando um Palíndromo com Tarefas

- Usando ***#pragma omp task*** para escrever 1x "***race car*** " ou "***car race*** "
  - Tarefas geradas executam independentemente
  - Programador é quem garante semântica correta

- Usa a diretiva *single*:
  - Assumindo duas *threads* paralelas, uma *thread* entra na região da *single*
    - Executa o bloco de comandos e gera as tarefas que não executam ainda
  - A segunda *thread* "salta" a single e espera na barreira implícita ao final da *single*

  - Se houver tarefas esperando para execução, *threads* não esperam em barreiras
    - Elas ficam disponíveis para executar as tarefas, conforme elas são geradas

  - **As tarefas só executam quando as *threads* chegam nas barreiras**
    - Ambas as *threads* podem executar as tarefas
    - Depende da chegada nas barreiras
    - A mesma *thread* pode servir mais de uma tarefa

```
1 #pragma omp parallel
2 {
3   #pragma omp single
4   {
5     #pragma omp task
6       {printf("race ");} // Task #1
7     #pragma omp task
8       {printf("car ");}  // Task #2
9   } // End of single region
10 } // End of parallel region
```

Van der Pas et al. (2017)

Executando Comandos Antes das Tarefas

- Agora imprimindo “**A *race car*** ” ou “**A *car race*** ” sem preferência por uma das ordens

```
 1 #pragma omp parallel
 2 {
 3   #pragma omp single
 4   {
 5     printf("A ");
 6     #pragma omp task
 7       {printf("race ");} // Task #1
 8     #pragma omp task
 9       {printf("car ");}  // Task #2
10   } // End of single region
11 } // End of parallel region
```

Van der Pas et al. (2017)

Executando Comandos após Tarefas

- Agora imprimindo acrescentando no fim "***is fun to watch.***"

```
1 #pragma omp parallel
2 {
3   #pragma omp single
4   {
5     printf("A ");
6     #pragma omp task
7       {printf("race ");} // Task #1
8     #pragma omp task
9       {printf("car ");}  // Task #2
10    printf("is fun to watch.\n");
11  } // End of single region
12 } // End of parallel region
```

- 1-2
  - ***Ts*** chega **antes** na barreira
- 3-4
  - ***Ts*** **intercala** impressão com a outra thread
- 5-6
  - ***Ts*** chega **depois** na barreira que as duas impressões

Produz essas saídas com duas *threads*
Thread executando região *single*: ***Ts***

Reference ID	Text printed
1	A is fun to watch. race car
2	A is fun to watch. car race
3	A race is fun to watch. car
4	A car is fun to watch. race
5	A race car is fun to watch.
6	A car race is fun to watch.

Van der Pas et al. (2017)

Usando a Diretiva #pragma omp taskwait

- Agora imprimindo acrescentando no fim "***is fun to watch.***" com ***#pragma omp taskwait***

```
 1 #pragma omp parallel
 2 {
 3   #pragma omp single
 4   {
 5     printf("A ");
 6     #pragma omp task
 7       {printf("race ");} // Task #1
 8     #pragma omp task
 9       {printf("car ");}  // Task #2
10     #pragma omp taskwait
11       printf("is fun to watch.\n");
12   } // End of single region
13 } // End of parallel region
```

- Imprimir "***is fun to watch.***" após a região paralela resolveria também
  - Mas e se isso não fosse possível?

Van der Pas et al. (2017)

Diretiva single vs master

- Uso das diretivas *single* & *master* para evitar replicar a geração de tarefas
  - Master não tem uma barreira implícita.
    - Normalmente é uma *thread* ocupada com outras questões (sobrecarrega)
  - A barreira não precisa ser da diretiva *single*, pode ser da *parallel*.
  - Uma diretiva *parallel sections* com apenas uma seção resolveria também
    - Ela tem uma barreira implícita no final

Van der Pas et al. (2017)

Determinando a Dependência de Tarefas

- Cláusula ***depend(type:list)***
  - Garante a ordem de execução das tarefas

  - Tipo ***in***: dependência de entrada para a tarefa em relação às variáveis da lista
    - a tarefa só pode executar quando as variáveis de entrada estiverem atualizadas por outras tarefas que indicam ***out***

  - Tipo ***out***: dependência de saída da tarefa em relação às variáveis da lista
    - A tarefa deve produzir os resultados esperados por outras tarefas que marcaram esta variável como ***in***

  - ***Tipo inout***: dependência tanto de saída quanto de entrada para as variáveis da lista

- Ver exemplo de código para dependência de tarefas

Van der Pas et al. (2017)

Tarefas e o Ambiente de Dados

- O controle do ambiente de dados cabe ao programador

- Deve-se controlar a execução, de modo que cada tarefa execute a sua funcionalidade com os seus dados especificamente

- Verifique o código exemplo:
  - Algoritmo gera ***tam*** tarefas e utiliza ***T*** *threads* para contar quantos números pares há no intervalo entre **0** e ***tam***.

Referências

Pas, Ruud van der.; Stotzer, Eric; Terboven, Christian. ***Using OpenMP-the next step: affinity, accelerators, tasking and SIMD***. The MIT Press, Cambridge, MA, 2017. ISBN 9780262534789

GRAMA,A.; KUMAR, U.; GUPTA,A.; KARYPIS, G. Introduction to Parallel Computing, 2nd Edition, 2003.

OpenMP API 5.0 C/C++ Syntax Quick Reference Card. 2018. Disponível em https://www.openmp.org/wp-content/uploads/OpenMPRef-5.0-111802-web.pdf . Último acesso em 28/09/2020.

OpenMP Application Program Inferface, Version 5.0 – 2018, Disponível em https://www.openmp.org/wp-content/uploads/OpenMP-API-Specification-5.0.pdf. Último acesso em 28/09/2020.

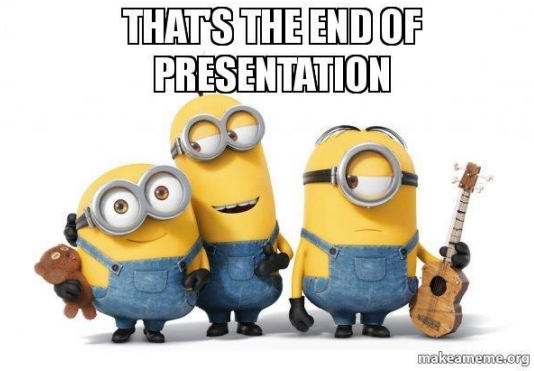
Reinders, James (Ed.) The Parallel Universe: flow Graphs, Speculative Locks, and Tasks Arenas. Intel. 2014. Disponível em https://www.openmp.org/about/whos-using-openmp/ Útlimo acesso em 28/09/2020

OpenMP Resources. NERSC Documentation (National Energy Research Scientific Computing Center. Berkeley Lab. Disponível em: https://docs.nersc.gov/development/programming-models/openmp/openmp-resources/. Último acesso em 28/09/2020.

OpenMP: *Tasks*

Paulo Sérgio Lopes de Souza
pssouza@icmc.usp.br

Universidade de São Paulo / ICMC / SSC – São Carlos
Laboratório de Sistemas Distribuídos e Programação Concorrente



CNPq Conselho Nacional de Desenvolvimento Científico e Tecnológico
CAPES
FAPESP