Utilizando Locks em seções críticas de OpenMP

Semântica de Locks

- Um "lock" representa uma posição de memória com dois estados ("locked" e "unlocked") e duas operações:
  - **set_lock (lock)**, que implementa a entrada na região crítica;
  - **unset_lock (lock)**, que implementa a saída na região crítica
- Semântica de **SET_LOCK (lock)**
  - A *thread* que invoca esta função permanece bloqueada até que o *lock* esteja no estado "unlocked"
  - A execução da função torna o estado do *lock* em "locked"
  - É garantido que apenas uma *thread* obtêm o estado "unlocked"
- Semântica de **UNSET_LOCK (lock)**
  - A execução da função torna o *lock* "unlocked"
- Como implementar "locks"?
  - Garantir que uma única *thread* adquira o "lock" é um problema de exclusão mútua;
    - A implementação requer suporte de hardware: (semáforos, etc).

Implementação de Locks

- Representa-se os estados “unlocked” por 0 e “locked” por 1
- A operação **set_lock** é tipicamente implementado por uma operação atômica (ou seja, indivisível) em hardware como, por exemplo, test-and-set:
  - test-and-set (t&s) executa atomicamente:
    - lê o conteúdo de uma posição de memória para um registrador
    - escreve 1 na posição de memória
- Código para **set_lock**:

```
set_lock: t&s r0, lock     // r0<-(lock) e (lock)<-1
          bnz r0, set_lock // tenta de novo se (lock) estava 1
```

- Se *lock* estava 1 (“locked”), a *thread* continua tentando adquirir o *lock*
- Se *lock* estava 0 (“unlocked”), a *thread* adquire o *lock*, que torna-se 1 (“locked”)
- Como **t&s** é atômica, uma única *thread* adquire o *lock*

Implementação de Locks (cont.)

- Para implementar **UNSET_LOCK**, basta armazenar 0 (“unlocked”) no *lock*
- Observações sobre operações atômicas:
  - Em máquinas com um único processador, a atomicidade de **t&s** é garantida
  - Em máquinas multiprocessadas, é necessário suporte da arquitetura de memória para garantir a atomicidade de **t&s**
  - Há múltiplas operações atômicas similares a **t&s** como, por exemplo, **fetch-and-add** ou **compare-and-swap**, que armazenam em registrador o conteúdo de uma posição de memória enquanto, simultaneamente, modificam o valor dessa posição

Locks em OpenMP

- *Locks* permitem implementar exclusão mútua em trechos arbitrários de um programa de forma similar a cláusula CRITICAL
  - A região crítica é definida dinamicamente, pois *locks* são implementados por invocações a funções (subrotinas em Fortran)
  - enquanto CRITICAL é definido estaticamente e obrigatoriamente começa e termina no mesmo procedimento
- **omp_set_lock(svar)**
  - A thread que invoca esta função permanece bloqueada até que *svar* (o *lock*) esteja no estado "unlocked"
    - A execução da função torna o estado do *lock* em "locked" e o *lock* pertence à thread
- **omp_unset_lock(svar)**
  - Só pode ser invocada pela *thread* que detém o *lock*
    - A execução da função torna o *lock* "unlocked"

Locks em OpenMP (Cont.)

- **omp_test_lock(svar)**
  - Similar a **OMP_SET_LOCK**, exceto que *threads* que não obtém o *lock* não ficam bloqueadas; retornam com valor *.FALSE.*
  - A *thread* que obtém o *lock* retorna *.TRUE.*
- **omp_init_lock(svar)**
  - Cria o *lock* no estado "unlocked"
- **omp_destroy_lock(svar)**
  - Destrói o *lock* que está, obrigatoriamente, no estado "unlocked"
- *Lock* é uma variável do tipo:
  - *INTEGER (KIND=OMP_LOCK_KIND)* em Fortran
  - *omp_lock_t* em C/C++

Exemplo de uso:

```
omp_lock_t my_lock[n];
for(i=0; i<n; i++)
  omp_init_lock(&my_lock[i]);


#pragma omp for
for(j=0; j<n; j++){
    omp_set_lock(&my_lock[ind[j]]);
    C[ind[j]]++; //núm.elem.de tamanho i
    omp_unset_lock(&my_lock[ind[j]]); }


for(i=0; i<n; i++)
  omp_destroy_lock(&my_lock[i]);
```

Funções de Tempo

Funções de Tempo

- **omp_get_wtime()**
  - Retorna *wall clock* em dupla precisão desde algum momento fixo no passado, em segundos

- **omp_get_wtick()**
  - Retorna a resolução em dupla precisão do relógio utilizado em **omp_get_wtime()**, em segundos

Variáveis de Ambiente

Variáveis de Ambiente

- **OMP_NUM_THREADS**
  - É o número de *threads* a usar durante a computação
    - Sobrescrito por invocação à função omp_set_num_threads()
- **OMP_NESTED**
  - Liga (.TRUE.) ou desliga (.FALSE.) aninhamento de seções paralelas
    - Default .FALSE.
- **OMP_DYNAMIC**
  - Permite (.TRUE.) ou impede (.FALSE.) ajustar dinamicamente o número de *threads* para otimizar o uso dos recursos do sistema
    - Default é dependente da implementação (tipicamente .FALSE.)
- **OMP_SCHEDULE**
  - Define como laços *DO* com clausula *SCHEDULE(RUNTIME)* compartilham o trabalho do laço
    - Veja o padrão para possíveis valores