14- Scheduling Loops

We want to parallelize this loop.

```
sum = 0.0;
for (i = 0; i <= n; i++)
   sum += f(i);
```

In practice, how are iterations assigned to threads?

Thread	Iterations
0	0, 1, 2, ..., n/t – 1
1	n/t, n/t + 1, ..., 2n/t
...	...
t-1	n(t-1)/t, ..., n-1

Default partitioning.

Thread	Iterations
0	$0, n/t, 2n/t, \ldots$
1	$1, n/t+1, 2n/t+1, \ldots$
$\vdots$	$\vdots$
$t-1$	$t-1, n/t+t-1, 2n/t+t-1, \ldots$

Cyclic partitioning.

```
double f(int i) {
   int j, start = i*(i+1)/2, finish = start + i;
   double return_val = 0.0;

   for (j = start; j <= finish; j++) {
      return_val += sin(j);
   }
   return return_val;
}  /* f */
```

```
sum = 0.0;
for (i = 0; i <= n; i++)
   sum += f(i);
```

- i.e., f(i) calls the sin function i times.
- assume the time to execute f(2i) requires approximately twice as much time as the time to execute f(i).
- How are iterations assigned to threads? What's the best way of doing it?

Risultati

#Threads	1	2 (default scheduling)	2 (cyclic scheduling)
Runtime	3.67	2.76	1.84
Speedup	1	1.33	1.99

La clausola di programmazione

- Default schedule:

```
        sum = 0.0;
#       pragma omp parallel for num_threads(thread_count) \
           reduction(+:sum)
        for (i = 0; i <= n; i++)
           sum += f(i);
```

- Cyclic schedule:

```
        sum = 0.0;
#       pragma omp parallel for num_threads(thread_count) \
           reduction(+:sum) schedule(static,1)
        for (i = 0; i <= n; i++)
           sum += f(i);
```

schedule (type , chunksize)

- Il tipo può essere:
  - **statico**: le iterazioni possono essere assegnate ai thread prima dell'esecuzione del ciclo. prima dell'esecuzione del ciclo.
  - **dinamico** o **guidato**: le iterazioni sono assegnate ai thread durante l'esecuzione del ciclo. thread durante l'esecuzione del ciclo.
  - **auto**: il compilatore e/o il sistema di run-time determinano il programma.
  - **runtime**: la programmazione viene determinata in fase di esecuzione.

  Il chunksize è un numero intero positivo.

The Static Schedule Type

e.g., twelve iterations, 0, 1, . . . , 11, and three threads

```
schedule(static,1)
```

Thread 0 : 0,3,6,9
Thread 1 : 1,4,7,10
Thread 2 : 2,5,8,11

e.g., twelve iterations, 0, 1, . . . , 11, and three threads

```
schedule(static,2)
```

Thread 0 : 0,1,6,7
Thread 1 : 2,3,8,9
Thread 2 : 4,5,10,11

e.g., twelve iterations, 0, 1, . . . , 11, and three threads

```
schedule(static,4)
```

$$\begin{aligned} \text{Thread } 0: &\quad 0,1,2,3 \\ \text{Thread } 1: &\quad 4,5,6,7 \\ \text{Thread } 2: &\quad 8,9,10,11 \end{aligned}$$

The Dynamic Schedule Type

- Anche le iterazioni sono suddivise in parti iterazioni consecutive di dimensioni di blocchi (**chunksize**).
- Ogni thread esegue un pezzo, e quando un thread termina un pezzo, ne richiede un altro al sistema di run-time.
- Questo continua finché non vengono completate tutte le iterazioni completato.
- La dimensione del blocco può essere omessa. Quando viene omesso, a viene utilizzata la dimensione del blocco pari a 1.
- Migliore bilanciamento del carico, ma costi generali più elevati pianificare i blocchi (può essere ottimizzato tramite il file dimensione del pezzo)

The Guided Schedule Type

- Ogni thread esegue anche un pezzo e quando un thread finisce un pezzo, ne richiede un altro.
- Tuttavia, in una pianificazione guidata, man mano che i blocchi vengono completati la dimensione dei nuovi blocchi diminuisce.
- I pezzi hanno una dimensione num_iterazioni/num_thread, dove num_iterazioni è il numero di iterazioni non assegnate
- Se non viene specificata la dimensione dei blocchi, la dimensione dei blocchi diminuisce fino a 1.
- Se viene specificata la dimensione del blocco, diminuisce fino alla dimensione del blocco, con l'eccezione che l'ultimo pezzo può essere più piccolo rispetto alla dimensione del pezzo.
- Pezzi più piccoli verso la fine per evitare sbandati

Thread	Chunk	Size of Chunk	Remaining Iterations
0	1 – 5000	5000	4999
1	5001 – 7500	2500	2499
1	7501 – 8750	1250	1249
1	8751 – 9375	625	624
0	9376 – 9687	312	312
1	9688 – 9843	156	156
0	9844 – 9921	78	78
1	9922 – 9960	39	39
1	9961 – 9980	20	19
1	9981 – 9990	10	9
1	9991 – 9995	5	4
0	9996 – 9997	2	2
1	9998 – 9998	1	1
0	9999 – 9999	1	0

Assignment of trapezoidal rule iterations 1–9999 using a guided schedule with two threads.

The Runtime Schedule Type

Il sistema utilizza la variabile d'ambiente `OMP_SCHEDULE` per determinare in fase di esecuzione come pianificare il ciclo.
La variabile d'ambiente `OMP_SCHEDULE` può assumerne qualsiasi dei valori che possono essere utilizzati per un'operazione statica, dinamica o programma guidato.
Esempio: `export OMP_SCHEDULE=" static,1 "`

Un altro modo per specificare il tipo di pianificazione è impostarlo con la funzione `omp_set_schedule(omp_sched_t kind, int Chunk_size)`;

Come selezionare una opzione di schedule?

- **Static**: se le iterazioni sono *omogenee*
- **Dynamic/Guided**: Se il costo di esecuzione *varia*
  - Se in dubbio settare:
    - `#pragma omp parallel for schedule ( runtime )`
- **Non vi è alcuna garanzia** che selezionerà di più esecuzione dell'opzione di pianificazione.
- Misurare/Provare diverse opzioni (la scelta migliore potrebbe essere diverso a seconda dell'input nel file programma)

Mutual Exclusion

Sezioni critiche nominate

OpenMP offre la possibilità di aggiungere un nome a una direttiva
direttiva critica:
`#pragma omp critical (name)`

- In questo modo, due blocchi protetti con direttive critiche con direttive critiche con nomi diversi possono essere eseguiti simultaneamente (cioè, è come agire su due diversi

blocchi)

- Tuttavia, queste direttive devono essere impostate in fase di compilazione.
- Cosa succede se vogliamo avere più blocchi/sezioni critiche, ma non sappiamo quanti ma non sappiamo quanti sono in fase di compilazione? tempo di compilazione? (ad esempio, un elenco collegato con un blocco per ogni nodo)

Locks in OpenMP

```
omp_lock_t writelock;
omp_init_lock(&writelock);

#pragma omp parallel for
for ( i = 0; i < x; i++ )
{
    // some stuff
    omp_set_lock(&writelock);
    // one thread at a time stuff
    omp_unset_lock(&writelock);
    // some stuff
}

omp_destroy_lock(&writelock);
```

critical, atomic, o locks?

1. In generale, la direttiva atomica è potenzialmente il metodo più veloce per ottenere la mutua esclusione.
2. Tuttavia, le specifiche di OpenMP consentono alla direttiva atomica di applicare la mutua esclusione a tutte le direttive atomiche del programma. cioè, le seguenti potrebbero essere eseguite in modo mutuamente esclusivo (dipende dall'implementazione).
   `#pragma omp atomic x++;` `#pragma` omp atomic y++;`
3. L'uso dei lock dovrebbe probabilmente essere riservato a situazioni in cui la mutua esclusione è necessaria per una struttura di dati piuttosto che per un blocco di codice.

Some Caveats

1. Non si devono mischiare i diversi tipi di mutua esclusione per una singola sezione critica.
2. Non c'è garanzia di equità nei costrutti di mutua esclusione.
   `#pragma omp atomic x+=f(y);` `#pragma` omp atomic x=g(x);`
3. Può essere pericoloso “annidare” i costrutti di mutua esclusione.

Sezioni critiche annidate

```
int main() {

#  pragma omp critical
y = f(x);
…
}

double f(double x) {

#  pragma omp critical
z = g(x); /* z is shared */
…
return z;
}
```

Soluzione

Le sezioni critiche annidate andranno in stallo.

In questo esempio, è possibile risolvere il problema utilizzando sezioni critiche denominate.

```
int main() {

#  pragma omp critical(one)
y = f(x);
…
}

double f(double x) {

#  pragma omp critical (two)
z = g(x); /* z is shared */
…
return z;
}
```

Costrutti di sincronizzazione

Direttive master/singole

master, **single**:

Entrambi forzano l'esecuzione del seguente blocco strutturato da parte di un singolo thread.

C'è una differenza significativa: single implica una barriera all'uscita dal blocco.

Ci sono altre differenze (ad esempio, con master, il blocco viene garantito che il blocco

venga eseguito dal thread master).

```
int examined = 0;
int prevReported = 0;
#pragma omp for shared( examined, prevReported )
  for( int i = 0 ; i < N ; i++ )
   {
      // some processing

      // update the counter
#pragma omp atomic
      examined++;

      // use the master to output an update every 1000 newly ←
          finished iterations
#pragma omp master
      {
         int temp = examined;
         if( temp - prevReported >= 1000)
           {
             prevReported = temp;
             printf("Examined %.2lf%%\n", temp * 1.0 / N );
           }
      }
   }
```

Direttive barriera

barriera

blocca finché tutti i thread della squadra non raggiungono quel punto.

```
int main( )
{
    int a[5], i;
    #pragma omp parallel
    {
        // Perform some computation.
        #pragma omp for
        for (i = 0; i < 5; i++)
            a[i] = i*i;

        // Print intermediate results.
        #pragma omp master
            for (i = 0; i < 5; i++)
                printf("a[%d] = %d\n", i, a[i]);

        // Wait.
        #pragma omp barrier

        // Continue with the computation.
        #pragma omp for
        for (i = 0; i < 5; i++)
            a[i] += i;
    }
}
```

Le direttive di sezione/sezioni

Come possiamo inviare diversi task in parallelo?

```
#pragma omp parallel
switch (omp_get_thread_num())
    {
    case 0: {
          //concurrent block 0
          }
    break;
    case 1: {
          //concurrent block 1
          }
    break;
    }
}
```

Le singole voci di lavoro sono contenute in blocchi decorati da direttive di sezione

```
#pragma omp parallel sections
{
#pragma omp section
   {
      // concurrent block 0
   }
  ...
#pragma omp section
   {
      // concurrent block M-1
   }
}
```

La direttiva omp parallel sections combina le direttive omp parallel e omp sections.
Alla fine di un costrutto di sezione c'è una barriera implicita, a meno che non sia specificata una clausola nowait.

Synchronization Constructs

ordinato

usato all'interno di un parallelo for, per garantire che un blocco sarà eseguito come se fosse in ordine sequenziale.

```
double data[ N ];
#pragma omp parallel shared( data, N )
{

#pragma omp for ordered schedule( static, 1 )
  for(int i = 0; i < N; i++)
    {
       // process the data

       // print the results in order
#pragma omp ordered
       cout << data[i];
    }
}
```

ordered clause is required

False sharing

Condivisione fittizia

- condivisione di linee di cache senza condividere effettivamente i dati.
  Come risolvere il problema:
- Imbottire i dati (Pad the data)
- Modificare la mappatura dei dati ai thread/cores (Data mapping change)
- Utilizzare variabili private/locali (Using private variables)

Padding the data

- Original

```
double x[N];
#pragma omp parallel for schedule(static, 1)
  for( int i = 0; i < N; i++ )
     x[ i ] = someFunc( x [ i ] );
```

- Padded:

```
double x[N][8];
#pragma omp parallel for schedule(static, 1)
  for( int i = 0; i < N; i++ )
     x[ i ][ 0 ] = someFunc( x [ i ][ 0 ] );
```

- Can kill cache effectiveness.
- Wastes memory.

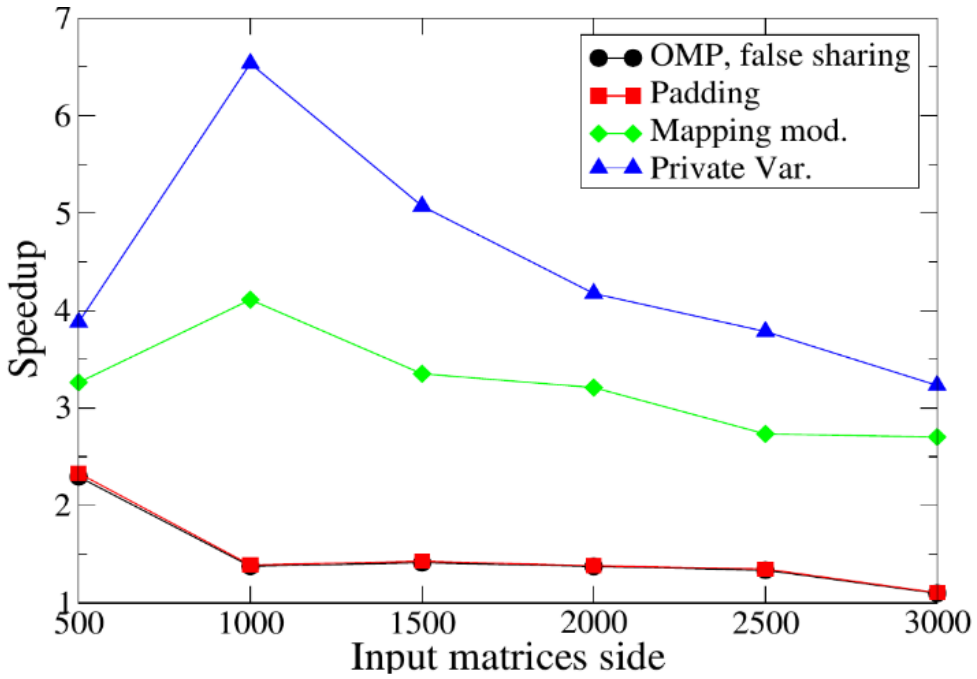
Data mapping change

```
double x[N];
#pragma omp parallel for schedule(static, 8)
  for( int i = 0; i < N; i++ )
     x[ i ] = someFunc( x [ i ] );
```

Using private variables

```
// assuming that N is a multiple of 8
double x[N];
#pragma omp parallel for schedule(static, 1)
  for( int i = 0; i < N; i += 8 )
  {
     double temp[ 8 ];
     for(int j = 0; j < 8; j++)
        temp[ j ] = someFunc( x [ i + j] );
     memcpy( x + i, temp, 8 * sizeof( double ));
  }
```

Impatto del false sharing nella moltiplicazione tra matrici



OpenMP + MPI

MPI definisce **4 livelli di sicurezza dei thread**:

- `MPI_THREAD_SINGLE:` esiste un solo thread nel programma
- `MPI_THREAD_FUNNELED` : solo il thread master può effettuare chiamate MPI. chiamate MPI. Master è quello che chiama MPI_Init_thread()
- `MPI_THREAD_SERIALIZED` : Multithread, ma solo un thread può effettuare chiamate MPI alla volta
- `MPI_THREAD_MULTIPLE` : Multithread e ogni thread può effettuare chiamate MPI in qualsiasi momento.

Più **sicuro** (più semplice) utilizzare `MPI_THREAD_FUNNELED`

- Si adatta bene alla maggior parte dei modelli OpenMP
- Loop costosi parallelizzati con OpenMP

- Comunicazione e chiamate MPI tra i loop

OpenMP/Pthreads + MPI

$ ./a.out 4
$ Time: 0.40 seconds

$ mpirun -n 1 ./a.out 4
$ Time: 1.17 seconds

Why?
Open MPI maps each process on a core. Thus, all the threads created by tha process will run on the same core (i.e., 4 threads will run on the same core)

How to fix it?

$ mpirun --bind-to-none -n 1 ./a.out 4
$ Time: 0.40 seconds

How to check how Open MPI is binding processes?
$ mpirun --report-bindings -n 1 ./a.out 4 1024 1024 1024
MCW rank 0 bound to socket 0[core 0[hwt 0-1]]:
[BB/../../../../../../..][../../../../../../../..][../../../../../../../..][../../../../../../../..]