KOÇ UNIVERSITY

Variable Scopes and Data Dependence

Didem Unat

COMP 429/529 Parallel Programming

A Programmer's View of OpenMP

OpenMP will:

- Allow a programmer to separate a program into *serial regions* and *parallel regions,* rather than concurrently-executing threads.
- Hide stack management
- Provide synchronization constructs

OpenMP will not:

- Parallelize automatically
- Guarantee speedup
- Provide freedom from data races

Data Dependence

Parallel Loop Construct

- Compiler calculates loop bounds for each thread directly from *serial* source (computation decomposition)
- Compiler also manages data partitioning
- Implicit barrier at the end of the loop
- This assumes a parallel region has already been initiated, otherwise it executes in serial on a single processor.

<table>
<tr>
<th>Serial Program</th>
<th>Parallel Program</th>
<th>Parallel Program</th>
</tr>
<tr>
<td><pre>double res[N];

for(int i=0; i < N; i++)
  do_huge_comp(res[i]);</pre></td>
<td><pre>double res[N];

#pragma omp parallel
{
  #pragma omp for
  for(int i=0; i < N; i++)
      do_huge_comp(res[i]);
}</pre></td>
<td><pre>double res[N];

#pragma omp parallel for
for(int i=0; i < N; i++)
      do_huge_comp(res[i]);</pre></td>
</tr>
</table>

Data Dependence

- A ***data dependence*** is an ordering on a pair of memory operations that must be preserved to maintain correctness.

- **Question:** When is parallelization guaranteed to be safe?
- **Answer:** If there are no data dependences across reordered computations.

- **Definition: Two memory accesses** are involved in a data dependence if they may refer to the same memory location and one of the accesses is a **write.**

Bernstein' s conditions (1966)

- True Dependence - Read after Write (RAW)
  - If process P_i writes to a memory cell M_i, then no process P_j can read the cell M_i.
- Anti-Dependence - Write after Read (WAR)
  - If process P_i read from a memory cell M_i, then no process P_j can write to the cell M_i.
- Output Dependence - Write after Write (WAW)
  - If process P_i writes to a memory cell M_i, then no process P_j can write to the cell M_i.

Data Dependence Types

Flow dependence (True dependence) (RAW)

S1: X = A+B

S2: C= X+A

Anti dependence (WAR)

S1: A = X + B

S2: X= C + D

Output dependence (WAW)

S1: X = A+B

S2: X= C + D

- **Definition: Data dependence exists from a reference instance i to j iff**
  - either i or j is a write operation
  - i and j refer to the same variable
  - i executes before j

- Actually, parallelizing compilers must formalize this to guarantee correct code.

Preserve Dependences

- Fundamental Theorem of Dependence:
  - Any reordering transformation that preserves every dependence in a program preserves the meaning of that program.
- Parallelization
  - Computations that execute in parallel between synchronization points are potentially reordered.
  - Is that reordering safe? According to our definition, it is safe if it preserves the dependences in the code.

```
#pragma omp barrier

. . .
#pragma omp for nowait
for()
. . .
#pragma omp barrier
```

Data Dependence for Arrays

```
for (i=2; i<N; i++)
    A[i] = A[i-2]+1;
```

Loop- Carried dependence

```
for (i=1; i<=N; i++)
    A[i] = A[i]+1;
```

Loop-Independent dependence

- Recognizing parallel loops (intuitively)
  - Find data dependences in loop
  - No dependences crossing iteration boundary → parallelization of loop iterations is safe

Dependences in Loops (I)

- Dependences in loops are easy to understand if the loops are unrolled. Now the dependences are between statement "executions".

```
for (i=0; i<n; i++){
      S1    a[i] = b[i] + 1;
      S2    c[i] = a[i] + 2;
}
```

Dependences in Loops (I)

- Dependences in loops are easy to understand if the loops are unrolled. Now the dependences are between statement "executions".

```
for (i=0; i<n; i++){
    S1    a[i] = b[i] + 1;
    S2    c[i] = a[i] + 2;
}
```

i=0	i=1	i=2
S1: a[0] = b[0] + 1 S2: c[0] = a[0] + 2	S1: a[1] = b[1] + 1 S2: c[1] = a[1] + 2	S1: a[2] = b[2] + 1 S2: c[2] = a[2] + 2

Dependences in Loops (I)

- Dependences in loops are easy to understand if the loops are unrolled. Now the dependences are between statement "executions".

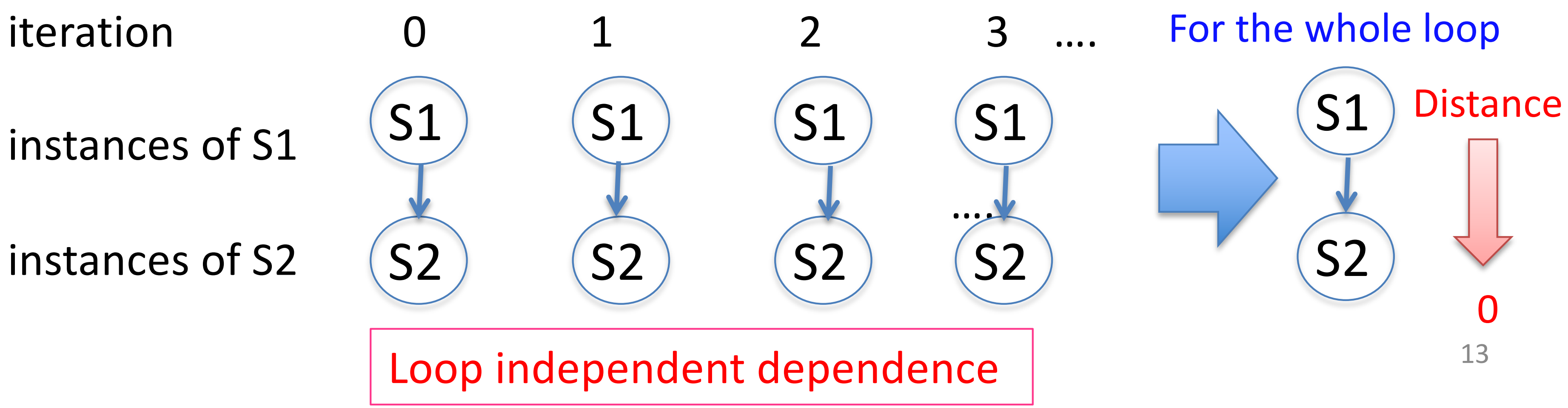
```
for (i=0; i<n; i++){
      S1    a[i] = b[i] + 1;
      S2    c[i] = a[i] + 2;
}
```

i=0	i=1	i=2
S1: a[0] = b[0] + 1 S2: c[0] = a[0] + 2	S1: a[1] = b[1] + 1 S2: c[1] = a[1] + 2	S1: a[2] = b[2] + 1 S2: c[2] = a[2] + 2

Dependences in Loops (I)

- Dependences in loops are easy to understand if the loops are unrolled. Now the dependences are between statement "executions".

```
for (i=0; i<n; i++){
      S1     a[i] = b[i] + 1;
      S2     c[i] = a[i] + 2;
}
```



Dependences in Loops (I)

- Dependences in loops are easy to understand if the loops are unrolled. Now the dependences are between statement "executions".

```
for (i=0; i<n; i++){
      S1    a[i] = b[i] + 1;
      S2    c[i] = a[i] + 2;
}
```

For the dependences shown here, we assume that arrays do not overlap in memory (**no aliasing**).

Dependences in Loops (II)

- Dependences in loops are easy to understand if the loops are unrolled. Now the dependences are between statement "executions".

```
for (i=1; i<n; i++){
    S1    a[i] = b[i] + 1;
    S2    c[i] = a[i-1] + 2;
}
```

Dependences in Loops (II)

- Dependences in loops are easy to understand if the loops are unrolled. Now the dependences are between statement “executions”.

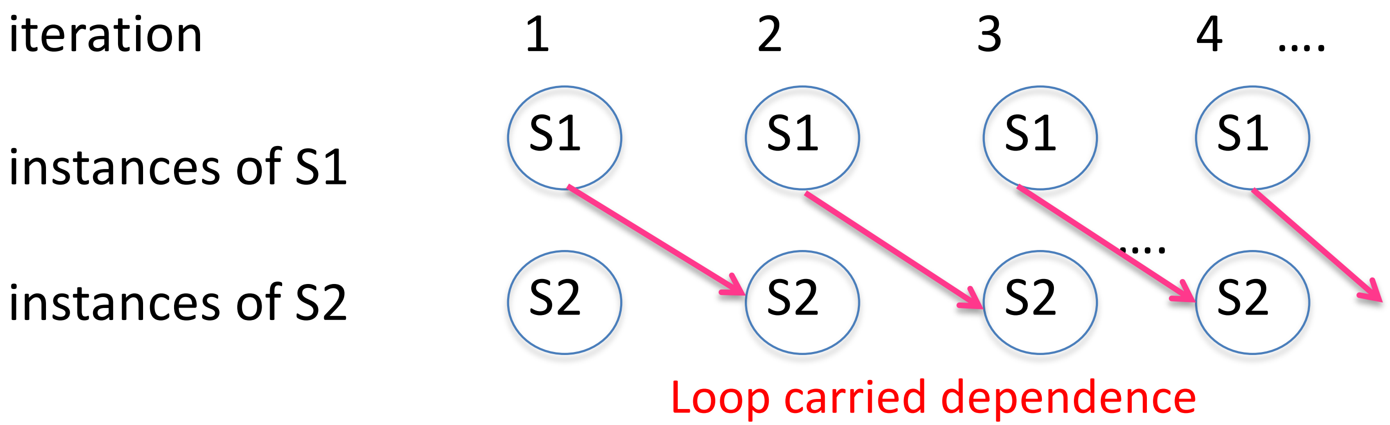
```
for (i=1; i<n; i++){
      S1      a[i] = b[i] + 1;
      S2      c[i] = a[i-1] + 2;
}
```

i=1	i=2	i=3
S1: a[1] = b[1] + 1	S1: a[2] = b[2] + 1	S1: a[3] = b[3] + 1
S2: c[1] = a[0] + 2	S2: c[2] = a[1] + 2	S2: c[3] = a[2] + 2

Dependences in Loops (II)

- Dependences in loops are easy to understand if the loops are unrolled. Now the dependences are between statement “executions”.

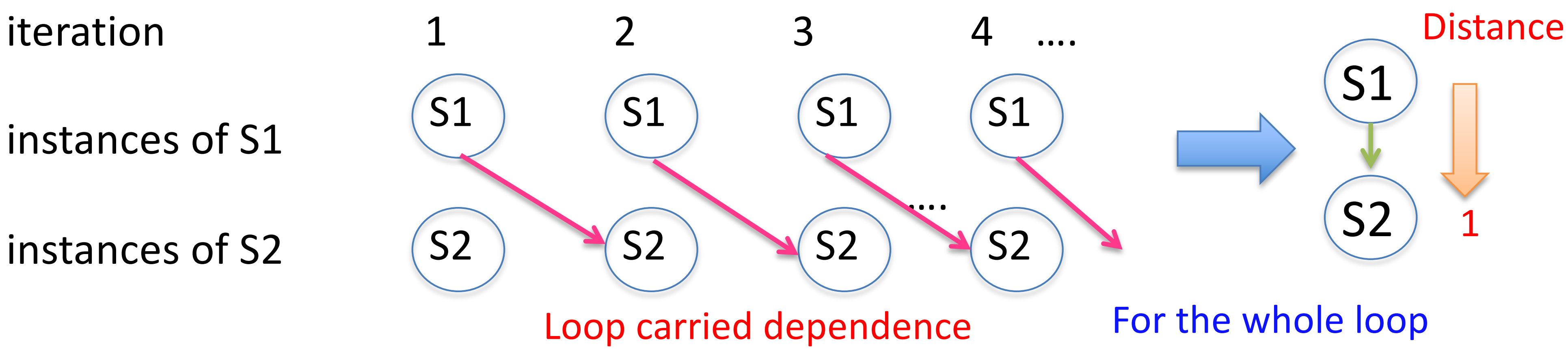
```
for (i=1; i<n; i++){
      S1     a[i] = b[i] + 1;
      S2     c[i] = a[i-1] + 2;
}
```



Dependences in Loops (II)

- Dependences in loops are easy to understand if the loops are unrolled. Now the dependences are between statement "executions".

```
for (i=1; i<n; i++){
    S1    a[i] = b[i] + 1;
    S2    c[i] = a[i-1] + 2;
}
```



Loop-independent vs. -carried dependences

```
for (i=1; i<n; i++)
{
  S1: a[i] = a[i-1] + 1;
  S2: b[i] = a[i];
}

for (i=1; i<n; i++)
 for (j=1; j< n; j++)
   S3: a[i][j] = a[i][j-1] + 1;

for (i=1; i<n; i++)
 for (j=1; j< n; j++)
   S4: a[i][j] = a[i-1][j] + 1;
```

S1[i] →S1[i+1]: loop-carried

S1[i] →S2[i]: loop-independent

S3[i,j] → S3[i,j+1]:

- loop-carried on for j loop
- no loop-carried dependence

in for i loop

S4[i,j] →S4[i+1,j]:

- no loop-carried dependence in for j loop
- loop-carried on for i loop

"Source: Yan Solihin, Fundamentals of Parallel Computer Architecture, 2008".

OpenMP For

```
for (i=1; i<n; i++)
{
  S1: a[i] = a[i-1] + 1;
  S2: b[i] = a[i];
}
#pragma omp for
for (i=1; i<n; i++)

 for (j=1; j< n; j++)
   S3: a[i][j] = a[i][j-1] + 1;

for (i=1; i<n; i++)
#pragma omp for
 for (j=1; j< n; j++)
   S4: a[i][j] = a[i-1][j] + 1;
```

- For the first loop, S2 can be put in a separate loop in order for it to be parallelized

OpenMP compilers don’t check for dependences among iterations in a loop.
It’s programmer’s responsibility.

Dependence

- Loop-carried dependence: dependence exists across iterations;
  - if the loop is removed, the dependence no longer exists
- Loop-independent dependence: dependence exists within an iteration;
  - i.e., if the loop is removed, the dependence still exists.

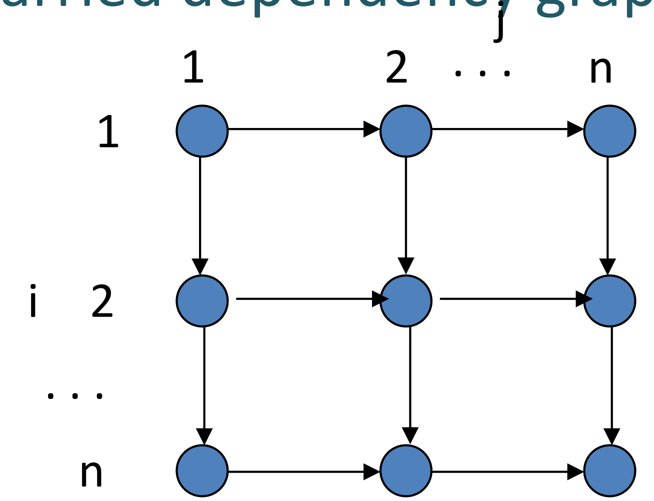
Another Example

- What is the iteration space graph and loop-carried dependency graph for this loop nest?
- How do you parallelize it?

```
for (i=1; i<=n; i++)
  for (j=1; j<=n; j++)
    S1: a[i][j] = a[i][j-1] + a[i][j+1] +
                  a[i-1][j] + a[i+1][j];
```

Another Example –cont.

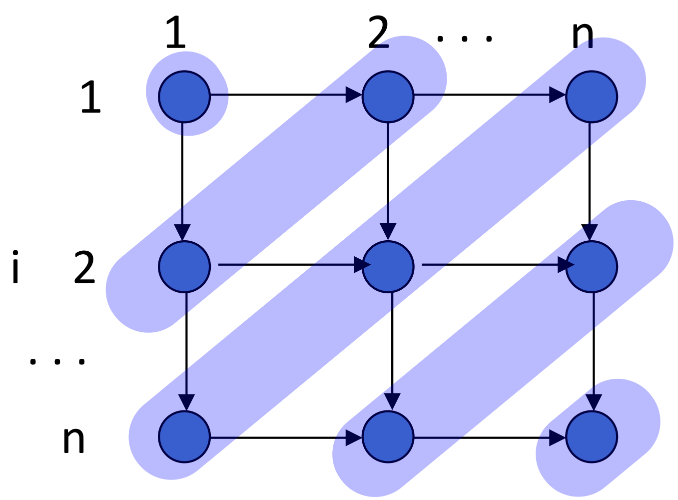
- True dependences:
  - S1[i,j] →True S1[i,j+1]
  - S1[i,j] →True S1[i+1,j]
- Output dependences:
  - None
- Anti dependences:
  - S1[i,j] →Anti S1[i+1,j]
  - S1[i,j] →Anti S1[i,j+1]
- Loop-carried dependency graph:



Note: each edge represents both true, and anti dependences

Another Example –cont.

- Identify which nodes are not dependent on each other
- In each anti-diagonal, the nodes are independent of each other



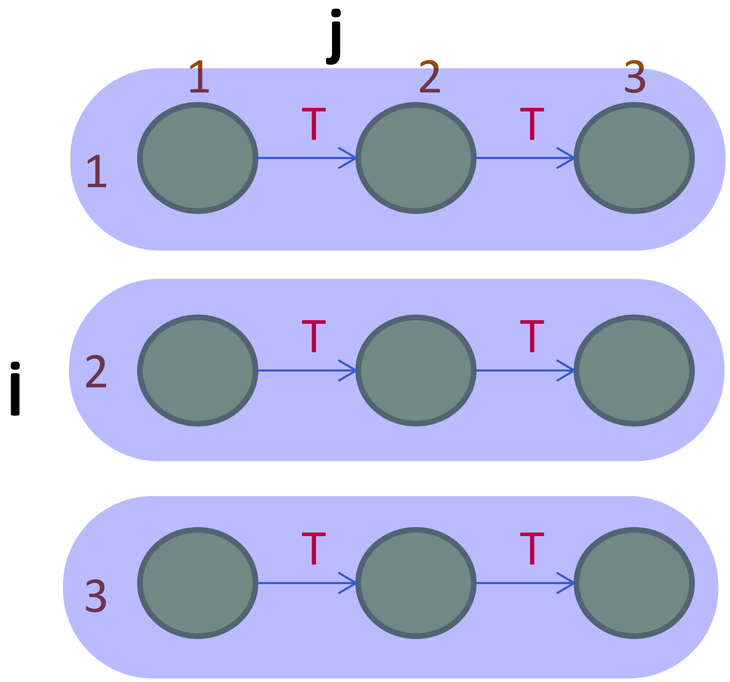
Note: each edge represents both true, and anti dependences

- Need to rewrite the code to iterate over anti-diagonals

Loop-carried Dependence Graph

- Shows the true/anti/output dependence relationships graphically
- A node is a point in the iteration space
- A directed edge represents the dependence
- Dependent iterations cannot be parallelized

```
for (i=1; i<4; i++)
 for (j=1; j< 4; j++)
   S3: a[i][j] = a[i][j-1] + 1;
```



"Source: Yan Solihin, Fundamentals of Parallel Computer Architecture, 2008".

Lab 2: Parallel Sum

- ssh username@login.kuacc.ku.edu.tr
- Copy the lab from here to your home directory
  - /kuacc/users/dunat/COMP429/OpenMP/Labs/Lab2-sum.c
- Request time in an interactive queue
  - srun -N 1 -n 4 -p short --time=00:30:00 --pty bash

Lab 2: Parallel Sum

- TODO 1:
  - Define private and shared variables
  - private(my_sum, my_x) shared(sum)
  - Compiler and run
- TODO 2:
  - Add critical section
  - #pragma omp critical
  - Compile and run

Lab 2: Parallel Sum

- TODO 3:
  - Define my_sum as firstprivate
  - firstprivate(my_sum)
  - Compiler and run
- TODO 4:
  - Comment out main, scroll down comment in the new main
  - Observe how reduction is implemented

OpenMP version of sum example

- Critical section should be in a parallel region

```
int items_per_task = n/t;
mutex m;
int my_sum=0, my_x, sum=0;
int start = thread_id * items_per_task;
#pragma omp parallel private(my_x, my_sum) shared(sum)
{
    my_sum = 0
    #pragma omp for
    for (i=0; i< N; i++)  {
          my_x = Compute_next_value(…);
          my_sum += my_x;
    }
    #pragma omp critical
    sum+= my_sum;
}
```

Variable Scope

```
 int my_sum=0, my_x, sum = 0;

#pragma omp parallel private(my_x) firstprivate(my_sum) shared(sum)
  {
        #pragma omp for
          for (i=0; i< N; i++)  {
              my_x = computeX(i);
              my_sum += my_x;
         }
        #pragma omp critical
              sum+= my_sum;
  }
```

Either declare **my_sum** as firstprivate or set **my_sum**=0 before **omp for** so that its value is initialized 0.

OpenMP Reduce

- OpenMP has reduce operation

Operator Variable

```
sum = 0;
#pragma omp parallel for private(my_x) reduction(+:sum)
for (i=0; i < 100; i++) {
    my_x = Compute_next_value(…);
    sum += my_x;
}
```

- Reduce ops and init() values (C and C++):

+	0	bitwise &	0	logical &	1
-	0	bitwise |	0	logical |	0
*	1	bitwise ^	0		

OpenMP Reduction

- OpenMP runtime/compiler has an ``efficient” implementation for parallel reduction

```
int sum=0;
#pragma omp parallel for reduction(+:sum)
   for (i=0; i< N; i++)   {
       sum += array[i];
   }
```

Scope of Variables

- In serial programming, the scope of a variable consists of those parts of a program in which the variable can be used.

- In OpenMP, the scope of a variable refers to the set of threads that can access the variable in a parallel block.

Scope in OpenMP

- A variable that can be accessed by all the threads in the team has a shared scope.

- A variable that can only be accessed by a single thread has private scope.

- The default scope for variables declared before a parallel block is shared.

Example:

```
#pragma omp parallel private(my_x, my_sum) shared(sum)
{}
```

Default Clause

- Lets the programmer specify the scope of each variable in a block.

```
default(none)
```

- With this clause the compiler will require that we specify the scope of each variable we use in the block and that has been declared outside the block.

Variable Scope

```
int *array = (int*) malloc(nthreads*sizeof(int));

#pragma omp parallel firstprivate(array)
{
  int t = omp_get_thread_num();
  array[t] = t;
}
```

Each thread gets a private pointer, but all pointers point to the same object.

http://pages.tacc.utexas.edu/~eijkhout/pcse/html/omp-data.html

More about OpenMP Loops

Parallel Loop Construct

- Compiler calculates loop bounds for each thread directly from *serial* source (computation decomposition)
- Compiler also manages data partitioning
- Implicit barrier at the end of the loop
- This assumes a parallel region has already been initiated, otherwise it executes in serial on a single processor.

<table>
<tr>
<td>

Serial Program

```
double res[N];


for(int i=0; i < N; i++)
  do_huge_comp(res[i]);
```

</td>
<td>

Parallel Program

```
double res[N];

#pragma omp parallel
{
  #pragma omp for
  for(int i=0; i < N; i++)
      do_huge_comp(res[i]);
}
```

</td>
<td>

Parallel Program

```
double res[N];

#pragma omp parallel for
for(int i=0; i < N; i++)
      do_huge_comp(res[i]);
```

</td>
</tr>
</table>

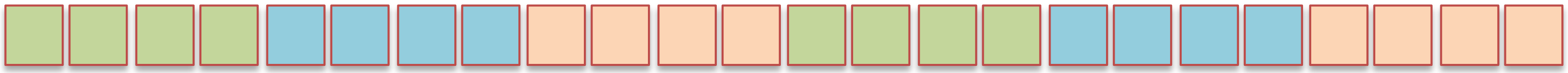
Loop Scheduling

- Schedule clause determines how loop iterations are divided among the thread team
  - **static([chunk])** divides iterations statically between threads
    - Each thread receives [chunk] iterations, rounding as necessary to account for all iterations
    - Default **[chunk]** is **ceil(# iterations / # threads)**
  - **dynamic([chunk])** allocates **[chunk]** iterations per thread, allocating an additional **[chunk]** iterations when a thread finishes
    - Forms a logical work queue, consisting of all loop iterations
    - Default **[chunk]** is 1
  - **guided([chunk])** allocates dynamically, but **[chunk]** is exponentially reduced with each allocation

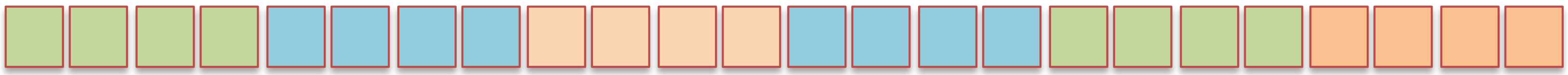
Loop Scheduling

- Static, dynamic vs guided for 3 threads

schedule(static, 4)

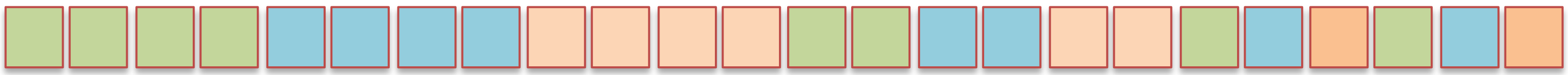


schedule(dynamic, 4)



Possible mapping

schedule(guided, 4)



Possible mapping

Impact of Loop Scheduling

- Load balance
  - Same work in each iteration?
  - Processors working at same speed?
- Scheduling overhead
  - Static decisions are cheap because they require no run-time coordination
  - Dynamic decisions have overhead that is impacted by complexity and frequency of decisions
- Data locality
  - Particularly for small chunk sizes
  - Also impacts data reuse on same processor

Nowait Clause

- **NO WAIT / nowait**: If specified, then threads do not need to synchronize at the end of the parallel loop.

```
#pragma omp parallel
{
    #pragma omp for nowait
    for()
     . . .

    #pragma omp for
    for()
     . . .
}
```

When can we insert nowait?

There shouldn't be any data dependency across the loops

What is the advantage?

Removes serialization bottleneck

Nowait Example

```
#pragma omp parallel
{
   #pragma omp for nowait
   for (i=1; i<n; i++)
     b[i] = (a[i] + a[i-1]) / 2.0;
   #pragma omp for nowait
   for (i=0; i<m; i++)
     y[i] = sqrt(z[i]);
}
```

NO dependency between loops

Collapse Clauses

- **Parallel for loop only** applies to the outer most loop
- **collapse**: Specifies how many loops in a nested loop should be collapsed into one large iteration space

```
#pragma omp for
for (i=0; i<N; i++)
  for (j=1; j<N; j++)
      . . .
```

```
#pragma omp for collapse(2)
for (i=0; i<N; i++)
  for (j=1; j<N; j++)
      . . .
```

- In order to collapse a nested loop, there **shouldn't be** any loop-carried dependence

For loop- Summary

Syntax:

`#pragma omp for` [clause [clause] ...] new-line

for-loop

clause can be one of the following:

`shared (list)`

`private(` list`)`

`reduction(` operator`:` list`)`

`schedule(` type [, chunk] `)`

`nowait`

`collapse(#)`

OpenMP Wall Time

- **OMP_GET_WTIME**
- Provides a portable wall clock timing routine
- Returns a double-precision floating point value equal to the number of elapsed seconds since some point in the past.

```
#include <omp.h>
double omp_get_wtime(void)
```

OpenMP Core Elements

OpenMP Programming Model

Parallel control structures	Work sharing	Data environment	Synchronization	Runtime functions, env. variables
Governs flow of control in the program **Parallel** directive	Distributes work among threads **Do/parallel do section, task** directives	Scopes variables **Shared** and **private** clauses	Coordinates thread execution **Critical** and **atomic** directives **barrier** directive	Runtime environment **omp_set_num_threads() omp_get_thread_num()** **OMP_NUM_THREADS OMP_SCHEDULE**

Environment Variables

OMP_NUM_THREADS

- Sets the number of threads to use during execution
- When dynamic adjustment of the number of threads is enabled, the value of this environment variable is the maximum number of threads to use
- If undefined one thread per CPU is used.

- For example,

  `setenv OMP_NUM_THREADS 16` **[csh, tcsh]**

  `export OMP_NUM_THREADS=16` **[sh, ksh, bash]**

- When a job is submitted through a scheduler, you can set the number of threads in the job script

There are other environment variables for OpenMP – will learn them later

No Compiler Support?

- In case the compiler doesn't support OpenMP

```
# include <omp.h>
```

```
#ifdef _OPENMP
# include <omp.h>
#endif
```

No support?

```
# ifdef _OPENMP
  int my_rank = omp_get_thread_num ( );
  int thread_count = omp_get_num_threads ( );
# e l s e
  int my_rank = 0;
  int thread_count = 1;
# endif
```

OpenMP Compilers

- GNU Compiler (gcc, gfrotran, g++)
  - - fopenmp
- Intel
  - -openmp
- PGI (Portland Group Compilers)
  - -mp
  - Acquired by Nvidia
- Cray, IBM and other compilers support OpenMP
- Version of the compiler may make a difference
  - New clauses may not be supported by all the compilers

Final Remarks

- Incremental parallelization
  - Parallelize individual computations in a program while leaving the rest of the program sequential
- Compiler based
  - Compiler generates thread program and synchronization
- System is viewed as collection of cores all of which have access to a shared main memory

- OpenMP1.0 for C/C++ in 1998
- OpenMP5.1 released in 2020
- OpenMP cheat sheet
  - https://www.openmp.org/wp-content/uploads/OpenMP-4.5-1115-CPP-web.pdf

Acknowledgments

- These slides are inspired and partly adapted from
  - Mary Hall (Univ. of Utah)
  - The course book (Pacheco)
  - Metin Aktulga (Machine State Univ.)
  - Yan Solihin, Fundamentals of Parallel Computer Architecture, 2008