OpenMP Contd.

High-Performance Computing

Summer 2021 at GIST

Tae-Hyuk (Ted) Ahn
Department of Computer Science
Program of Bioinformatics and Computational Biology
Saint Louis University

SAINT LOUIS
UNIVERSITY
EST. 1818

Compiling

Compiler / Platform	Compiler	Flag
Intel Linux Opteron/Xeon	icc icpc ifort	-openmp
PGI Linux Opteron/Xeon	pgcc pgCC pgf77 pgf90	-mp
GNU Linux Opteron/Xeon IBM Blue Gene	gcc g++ g77 gfortran	-fopenmp
IBM Blue Gene	bgxlc_r, bgcc_r bgxlC_r, bgxlc++_r bgxlc89_r bgxlc99_r bgxlf_r bgxlf90_r bgxlf95_r bgxlf2003_r *Be sure to use a thread-safe compiler - its name ends with **_r**	-qsmp=omp

How to compile and run OpenMP programs?

- gcc 4.2 and above supports OpenMP 3.0
  - `$ gcc -fopenmp a.c`
  - `$ g++ -fopenmp a.cpp`

- To run: 'a.out'
  - To change the number of threads:
    - For Bash Shell

      `$ export OMP_NUM_THREADS=4`

Compiler	Compiler Options	Default behavior for # of threads (OMP_NUM_THREADS not set)
GNU (gcc, g++, gfortran)	-fopenmp	as many threads as available cores
Intel (icc ifort)	-openmp	as many threads as available cores
Portland Group (pgcc,pgCC,pgf77,pgf90)	-mp	one thread

helloworld_omp.cpp

- Don’t change source code.
- $ export OMP_NUM_THREADS=4 as below.
- Do not build.
- Run!

```cpp
#include <iostream>
#include <stdio.h>
#include <omp.h>

using namespace std;

int main()
{
        #pragma omp parallel
        {
        int id = omp_get_thread_num();
        printf("Hello World from ID=%d!\n",id);
        }

        return 0;
}
```

```
ai@ubuntu-20-04:~/Lab/OpenMP$ export OMP_NUM_THREADS=4
ai@ubuntu-20-04:~/Lab/OpenMP$ ./helloworld_omp
Hello World from ID=0!
Hello World from ID=2!
Hello World from ID=1!
Hello World from ID=3!
```

OpenMP API Overview

Three Components:

- The OpenMP API is comprised of three distinct components. As of version 4.0:
  - Compiler Directives (44)
  - Runtime Library Routines (35)
  - Environment Variables (13)
- The application developer decides how to employ these components. In the simplest case, only a few of them are needed.
- Implementations differ in their support of all API components.

Set the number of threads in various ways

- Compiler Directives

```
#pragma omp parallel num_threads(4)
```

- Run-time Library

```
omp_set_num_threads(4);
```

- Environment Variables

```
export OMP_NUM_THREADS=4
```

Compiler Directives

- Compiler directives appear as comments in your source code and are ignored by compilers unless you tell them otherwise - usually by specifying the appropriate compiler flag.
- OpenMP compiler directives are used for various purposes:
  - Spawning a parallel region
  - Dividing blocks of code among threads
  - Distributing loop iterations between threads
  - Serializing sections of code
  - Synchronization of work among threads
- For example:

Fortran	`!$OMP PARALLEL DEFAULT(SHARED) PRIVATE(BETA,PI)`
C/C++	`#pragma omp parallel default(shared) private(beta,pi)`

Compiler Directives

- private (list), shared (list)
- firstprivate (list), lastprivate (list)
- reduction (operator: list)
- schedule (method [, chunk_size])
- nowait
- if (scalar_expression)
- num_thread (num)
- threadprivate(list), copyin (list)
- ordered
- collapse (n)
- tie, untie
- And more …

Run-time Library Routines

- These routines are used for a variety of purposes:
  - Setting and querying the number of threads
  - Querying a thread's unique identifier (thread ID), a thread's ancestor's identifier, the thread team size
  - Setting and querying the dynamic threads feature
  - Querying if in a parallel region, and at what level
  - Setting and querying nested parallelism
  - Setting, initializing and terminating locks and nested locks
  - Querying wall clock time and resolution
- For example:

Fortran	`INTEGER FUNCTION OMP_GET_NUM_THREADS()`
C/C++	`#include <omp.h>` `int omp_get_num_threads(void)`

Run-time Library Routines

- Number of threads: omp_{set,get}_num_threads
- Thread ID: omp_get_thread_num
- Scheduling: omp_{set,get}_dynamic
- Nested parallelism: omp_in_parallel
- Locking: omp_{init,set,unset}_lock
- Active levels: omp_get_thread_limit
- Wallclock Timer: omp_get_wtime
- thread private
- call function twice, use difference between end time and start time
- And more …

Environment Variables

- OpenMP provides several environment variables for controlling the execution of parallel code at run-time.
- These environment variables can be used to control such things as:
  - Setting the number of threads
  - Specifying how loop iterations are divided
  - Binding threads to processors
  - Enabling/disabling nested parallelism; setting the maximum levels of nested parallelism
  - Enabling/disabling dynamic threads
  - Setting thread stack size
  - Setting thread wait policy
- For example,

csh/tcsh	`setenv OMP_NUM_THREADS 8`
sh/bash	`export OMP_NUM_THREADS=8`

Environment Variables

- OMP_NUM_THREADS
- OMP_SCHEDULE
- OMP_STACKSIZE
- OMP_DYNAMIC
- OMP_NESTED
- OMP_WAIT_POLICY
- OMP_ACTIVE_LEVELS
- OMP_THREAD_LIMIT
- And more …

OpenMP Constructs

OpenMP's constructs:

- Parallel Regions
- Worksharing (for/DO, sections, …)
- Data Environment (shared, private, …)
- Synchronization (barrier, flush, …)
- Runtime functions/environment variables (omp_get_num_threads(), …)

Recap: helloworld_omp.cpp

```cpp
#include <iostream>
#include <stdio.h>
#include <omp.h>

using namespace std;

int main()
{
    #pragma omp parallel
    {
        int id = omp_get_thread_num();
        printf("Hello World from ID=%d!\n", id);
    }

    return 0;
}
```

```
Hello World from ID=6!
Hello World from ID=14!
Hello World from ID=1!
Hello World from ID=7!
Hello World from ID=8!
Hello World from ID=4!
Hello World from ID=5!
Hello World from ID=12!
Hello World from ID=13!
Hello World from ID=2!
Hello World from ID=11!
Hello World from ID=9!
Hello World from ID=15!
Hello World from ID=10!
Hello World from ID=3!
Hello World from ID=0!
```

```
ai@ubuntu-20-04:~/Lab/OpenMP$ g++ -fopenmp helloworld_omp.cpp -o helloworld_omp
ai@ubuntu-20-04:~/Lab/OpenMP$ ./helloworld_omp
```

Recap: helloworld_omp.cpp

- Don’t change source code.
- Export OMP_NUM_THREADS=4 as below.
- Do not build.
- Run!

```cpp
#include <iostream>
#include <stdio.h>
#include <omp.h>

using namespace std;

int main()
{
    #pragma omp parallel
    {
        int id = omp_get_thread_num();
        printf("Hello World from ID=%d!\n", id);
    }

    return 0;
}
```

```
ai@ubuntu-20-04:~/Lab/OpenMP$ export OMP_NUM_THREADS=4
ai@ubuntu-20-04:~/Lab/OpenMP$ ./helloworld_omp
Hello World from ID=0!
Hello World from ID=3!
Hello World from ID=1!
Hello World from ID=2!
```

How to check my BASH environment?

- Type env

```
$ env
```

- Type and check the OMP_NUM_THREADS

```
ai@ubuntu-20-04:~/Lab/OpenMP$ env | grep -i OMP
OMP_NUM_THREADS=4
ai@ubuntu-20-04:~/Lab/OpenMP$
```

Again, set the number of threads in various ways

- Compiler Directives

```
#pragma omp parallel num_threads(4)
```

- Run-time Library

```
omp_set_num_threads(4);
```

- Environment Variables

```
export OMP_NUM_THREADS=4
```

Question: What if does the number of threads set-up in both approaches? Which approach has the preference? Any override rule among them?

Test 1: Set up the “num_threads” clause

- The **`num_threads`** compiler directive lause can be added to a parallel directive.
- It allows the programmer to specify the number of threads that should execute the following block.

```
# pragma omp parallel num_threads(thread_count)
```

Test 1: Using “num_threads (2)” under “OMP_NUM_THREADS=4”

- Check the ”env”

```
ai@ubuntu-20-04:~/Lab/OpenMP$ env | grep OMP
OMP_NUM_THREADS=4
```

- Update the code.
  - `#pragma omp parallel num_threads(2)`
- Build and Run!

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

using namespace std;

int main()
{
        #pragma omp parallel num_threads(2)
        {
                int id = omp_get_thread_num();
                printf("Hello World from ID=%d!\n", id);
        }

        return 0;
}
```

```
ai@ubuntu-20-04:~/Lab/OpenMP$ g++ -fopenmp helloworld_omp.cpp -o helloworld_omp
ai@ubuntu-20-04:~/Lab/OpenMP$ ./helloworld_omp
Hello World from ID=0!
Hello World from ID=1!
```

Test 1: Using “num_threads (6)” under “OMP_NUM_THREADS=4”

- Update the code.
  - `#pragma omp parallel num_threads(6)`
- Build and Run!

```
#pragma omp parallel num_threads(6)
{
        int id = omp_get_thread_num();
        printf("Hello World from ID=%d!\n", id);
}
```

```
ai@ubuntu-20-04:~/Lab/OpenMP$ g++ -fopenmp helloworld_omp.cpp -o helloworld_omp
ai@ubuntu-20-04:~/Lab/OpenMP$ ./helloworld_omp
Hello World from ID=0!
Hello World from ID=3!
Hello World from ID=5!
Hello World from ID=4!
Hello World from ID=1!
Hello World from ID=2!
```

Override relationship

- **`OMP_NUM_THREADS`** specifies initially the number of threads.
- The presence of the **`num_threads`** clause overrides the environmental setting.

Test 2: Set up the "omp_set_num_threads (count)" routine

- The **`omp_set_num_threads`** routine affects the number of threads to be used for subsequent parallel regions that do not specify a **`num_threads`** clause, by setting the value.

```
omp_set_num_threads(thread_count)
```

Test 2: Using “omp_set_num_threads (3)”

- Check the ”env”

```
ai@ubuntu-20-04:~/Lab/OpenMP$ env | grep OMP
OMP_NUM_THREADS=4
```

- Add the code.
  - `omp_set_num_threads(3);`
- Check `num_threads(2)`
- Build and Run! Still 2 threads are used.

```
7  int main()
8  {
9          omp_set_num_threads(3);
10
11         #pragma omp parallel num_threads(2)
12         {
13                 int id = omp_get_thread_num();
14                 printf("Hello World from ID=%d!\n", id);
15         }
16
17         return 0;
18 }
```

```
ai@ubuntu-20-04:~/Lab/OpenMP$ g++ -fopenmp helloworld_omp.cpp -o helloworld_omp
ai@ubuntu-20-04:~/Lab/OpenMP$ ./helloworld_omp
Hello World from ID=0!
Hello World from ID=1!
```

Test 2: Use “omp_set_num_threads (3)” and remove “num_threads”

- Check the ”env”

```
ai@ubuntu-20-04:~/Lab/OpenMP$ env | grep OMP
OMP_NUM_THREADS=4
```

- Delete `num_threads(2)`
- Build and Run! OK, 3 threads are used.

```
 7 int main()
 8 {
 9         omp_set_num_threads(3);
10
11         #pragma omp parallel
12         {
13                 int id = omp_get_thread_num();
14                 printf("Hello World from ID=%d!\n", id);
15         }
16
17         return 0;
```

```
ai@ubuntu-20-04:~/Lab/OpenMP$ g++ -fopenmp helloworld_omp.cpp -o helloworld_omp
ai@ubuntu-20-04:~/Lab/OpenMP$ ./helloworld_omp
Hello World from ID=1!
Hello World from ID=2!
Hello World from ID=0!
```

Override relationship

- OMP_NUM_THREADS (if present) specifies initially the number of threads;
- calls to omp_set_num_threads() override the value of OMP_NUM_THREADS;
- the presence of the num_threads clause overrides both other values.

How Many Threads?

- The number of threads in a parallel region is determined by the following factors, in order of precedence:
  - Evaluation of the **IF** clause
  - Setting of the **NUM_THREADS** clause
  - Use of the **omp_set_num_threads()** library function
  - Setting of the **OMP_NUM_THREADS** environment variable
  - Implementation default - usually the number of CPUs on a node, though it could be dynamic.
- Threads are numbered from 0 (master thread) to N-1

PARALLEL Region Construct

- A parallel region is a block of code that will be executed by multiple threads. This is the fundamental OpenMP parallel construct.

C/C++

```
#pragma omp parallel [clause ...]  newline
                     if (scalar_expression)
                     private (list)
                     shared (list)
                     default (shared | none)
                     firstprivate (list)
                     reduction (operator: list)
                     copyin (list)
                     num_threads (integer-expression)

   structured_block
```

Clauses

- **IF** clause: If present, it must evaluate to .TRUE. (Fortran) or non-zero (C/C++) in order for a team of threads to be created. Otherwise, the region is executed serially by the master thread.

Restrictions

- A parallel region must be a structured block that does not span multiple routines or code files
- It is illegal to branch (goto) into or out of a parallel region
- Only a single IF clause is permitted
- Only a single NUM_THREADS clause is permitted
- A program must not depend upon the ordering of the clauses

Parallel region and if clause example

```
1 #include <cstdio>
2 #include <omp.h>
3
4 int main()
5 {
6   int nthreads, tid;
7
8   /* Fork a team of threads with each thread having a private tid variable */
9   #pragma omp parallel private(tid)
10  {
11    /* Obtain and print thread id */
12    tid = omp_get_thread_num();
13    printf("Hello World from thread = %d\n", tid);
14
15    /* Only master thread does this */
16    if (tid == 0)
17    {
18      nthreads = omp_get_num_threads();
19      printf("Number of threads = %d\n", nthreads);
20    }
21  }  /* All threads join master thread and terminate */
22  return 0;
23 }
```

Parallel region and if clause example

- Build and Run

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

int main()
{
  int nthreads, tid;

  /* Fork a team of threads with each thread having a private tid variable */
  #pragma omp parallel private(tid)
  {
    /* Obtain and print thread id */
    tid = omp_get_thread_num();
    printf("Hello World from thread = %d\n", tid);

    /* Only master thread does this */
    if (tid == 0)
    {
      nthreads = omp_get_num_threads();
      printf("Number of threads = %d\n", nthreads);
    }
  }  /* All threads join master thread and terminate */
  return 0;
}
```

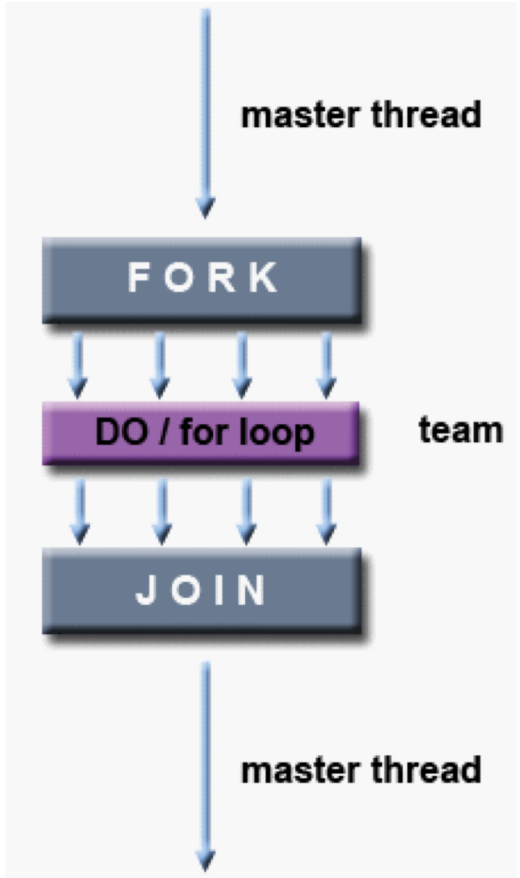
```
ai@ubuntu-20-04:~/Lab/OpenMP$ g++ -fopenmp parallelregion_omp.cpp -o parallelregion_omp
ai@ubuntu-20-04:~/Lab/OpenMP$ ./parallelregion_omp
Hello World from thread = 0
Number of threads = 4
Hello World from thread = 3
Hello World from thread = 2
Hello World from thread = 1
```

Work-Sharing Constructs

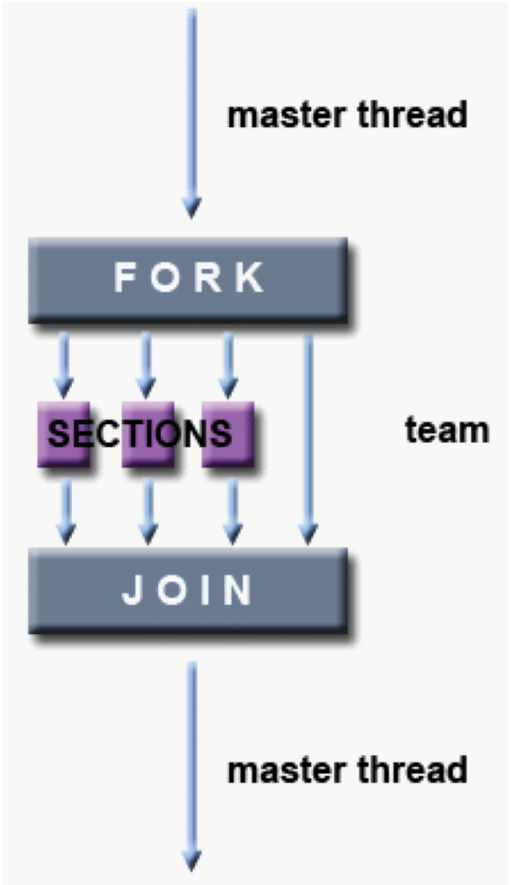
- A work-sharing construct divides the execution of the enclosed code region among the members of the team that encounter it.
- Work-sharing constructs do not launch new threads
- There is no implied barrier upon entry to a work-sharing construct, however there is an implied barrier at the end of a work sharing construct.

Types of Work-Sharing Constructs

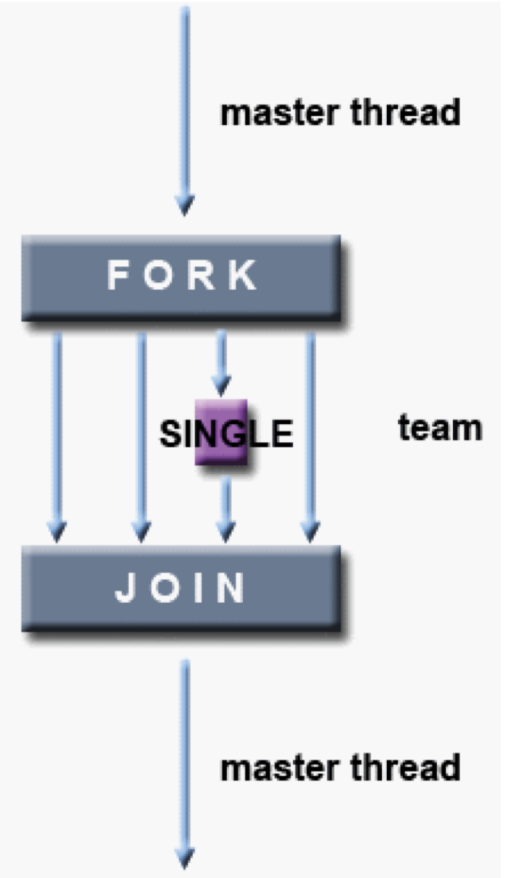
DO / for - shares iterations of a loop across the team. Represents a type of "data parallelism".



SECTIONS - breaks work into separate, discrete sections. Each section is executed by a thread. Can be used to implement a type of "functional parallelism".



SINGLE - serializes a section of code



DO / for Directive

- The DO / for directive specifies that the iterations of the loop immediately following it must be executed in parallel by the team. This assumes a parallel region has already been initiated, otherwise it executes in serial on a single processor.

C/C++

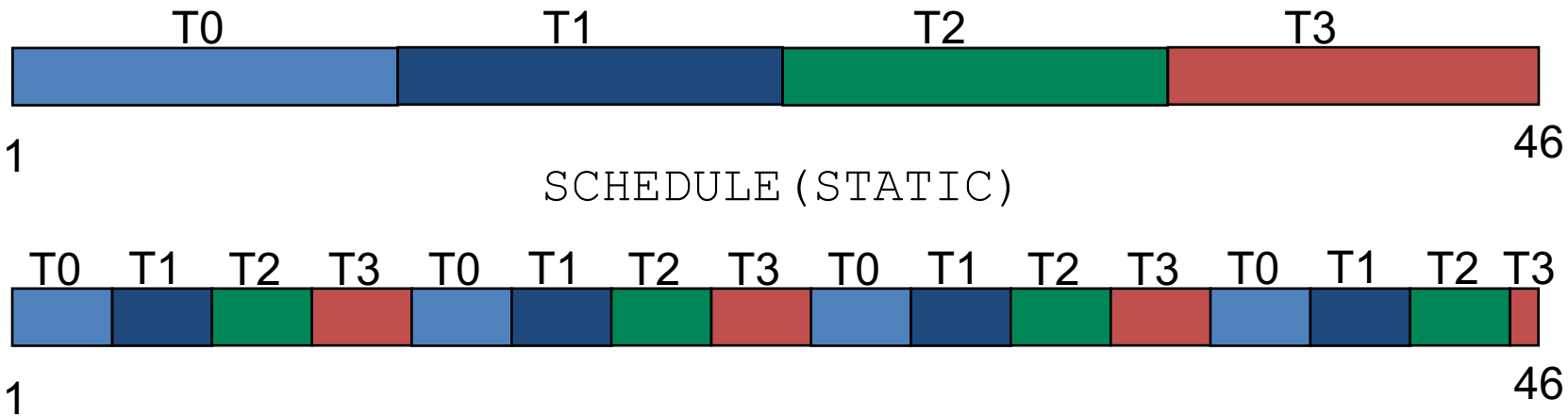
```
#pragma omp for [clause ...]  newline
                schedule (type [,chunk])
                ordered
                private (list)
                firstprivate (list)
                lastprivate (list)
                shared (list)
                reduction (operator: list)
                collapse (n)
                nowait

   for_loop
```

SCHEDULE clause

- Decide how the iterations are executed in parallel
  - schedule (static | dynamic | guided [, chunk] | runtime | auto)
  - default is **static**
- There is always a trade off between load balance and overhead
- Always start with static and go to more complex schemes as load balance requires.

STATIC schedule

- With no *chunksize* specified
  - Iteration space is divided into (approximately) equal chunks, and one chunk is assigned to each thread (**block** schedule)
- If *chunksize* is specified
  - Iteration space is divided into chunks, each of *chunksize* iterations. The chunks are assigned cyclically to each thread (**block cyclic** schedule)

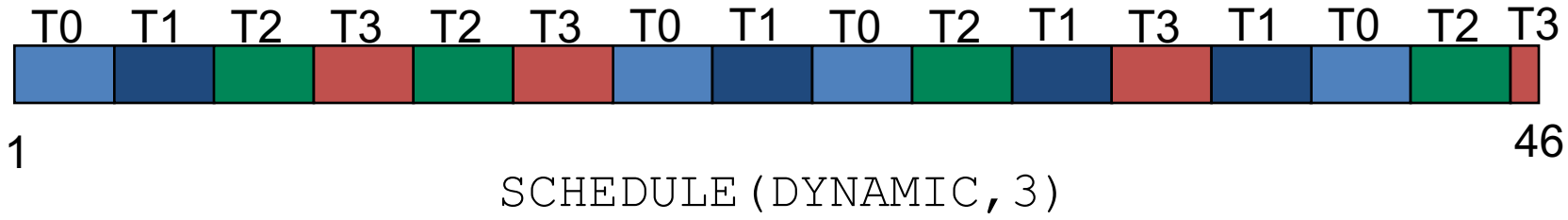


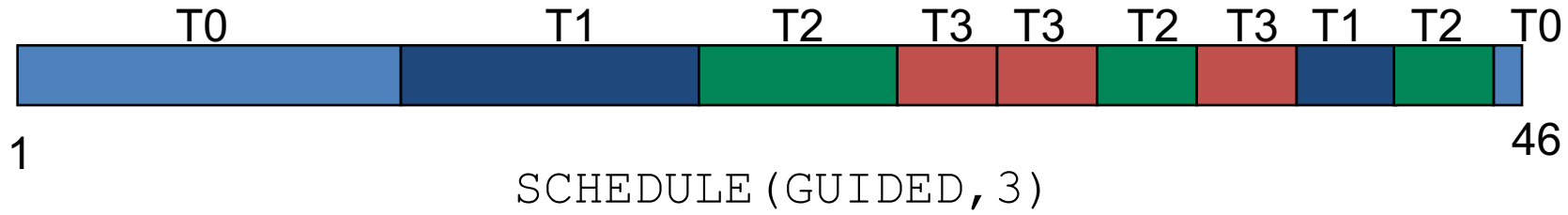
DYNAMIC and GUIDED schedules

- DYNAMIC
  - Divides the iteration space up into chunks of size *chunksize* and assigns them to threads on a first-come first-served basis.
  - i.e. as a thread finishes a chunk, it is assigned the next chunk in the list.
  - When no *chunksize* is specified it defaults to 1.
- GUIDED schedule
  - Similar to DYNAMIC, but the chunks start off large and get smaller exponentially.
  - The size of the next chunk is (roughly) the number of remaining iterations divided by the number of threads.
  - The *chunksize* specifies the minimum size of the chunks
  - When no *chunksize* is specifies it defaults to 1.

DYNAMIC and GUIDED schedules

DYNAMIC and GUIDED





Run Time Library & Environment Schedule

- Allows the choice of schedule to be deferred until runtime

```
schedule (runtime)
```

- Set by environment variable `OMP_SCHEDULE:`
  - STATIC, no chunk size specified
  - Sets the run-time schedule type and an optional chunk size.

```
$ export OMP_SCHEDULE="guided,4"
```

AUTO schedule

- (OpenMP 3.0) gives implementation freedom to choose best mapping of iterations to threads

Choosing a schedule

- STATIC is best for balanced loops –least overhead.
- STATIC is good for loops with mild or smooth load imbalance – but can introduce "false sharing" (see later).
- DYNAMIC is useful if iterations have widely varying loads, but ruins data locality (can get cache hit).
- GUIDED is often less expensive than DYNAMIC, but beware of loops where first iterations are the most expensive!
- Use RUNTIME for convenient experimentation

Count the primes

- Example file can be accessed at course website: `prime_count_omp.cpp`

prime_count_omp.cpp

- Open the code!

```cpp
# include <cstdlib>
# include <iostream>
# include <iomanip>
# include <omp.h>

using namespace std;

//****************************************************************************
int prime_count ( int n )
//****************************************************************************
{
    int i, j, prime, total_prime=0;

    #pragma omp parallel for reduction(+:total_prime) \
                             schedule(static) private(i, j, prime)
    for ( i = 2; i <= n; i++ ) {
        prime = 1;
        for ( j = 2; j < i; j++ ) {
            if ( i % j == 0 ) {
                prime = 0;
                break;
            }
        }
        total_prime += prime;
    }
    return total_prime;
}

```

```cpp
//****************************************************************************80
int main (int argc, char *argv[])
//****************************************************************************80
{
    int n, num_procs, num_threads, total_prime=0;
    double start, end;

    // Returns the number of processors that are available to the program
    num_procs = omp_get_num_procs();

    // Returns the maximum value that can be returned by a call to the
    // OMP_GET_NUM_THREADS function
    num_threads = omp_get_max_threads();

    cout << "\n";
    cout << "SCHEDULE_OPENMP\n";
    cout << "  C++/OpenMP version\n";
    cout << "  Count the primes from 1 to N.\n";
    cout << "  This is an unbalanced work load, particular for two threads.\n";
    cout << "  Demonstrate static and dynamic scheduling.\n";
    cout << "\n";
    cout << "  Number of processors available = " << num_procs << "\n";
    cout << "  Number of threads =              " << num_threads << "\n\n";
    cout << "Type N and enter: ";
    cin >> n ;

    start = omp_get_wtime();  // start time check
    total_prime = prime_count(n);
    end = omp_get_wtime();    // end time check

    cout << "  " << setw(8) << "N"
         << "  " << setw(18) << "Count of primes"
         << "  " << setw(18) << "Numer of threads"
         << "  " << setw(30) << "Elapsed wall clock time (sec)" << "\n";

    cout << "  " << setw(8) << n
         << "  " << setw(18) << total_prime
         << "  " << setw(18) << num_threads
         << "  " << setw(30) << end-start << "\n";

    return 0;
}
```

Hands On: Let us test “schedule”

- **SCHEDULE**: Describes how iterations of the loop are divided among the threads in the team. The default schedule is implementation dependent. For a discussion on how one type of scheduling may be more optimal than others, see http://openmp.org/forum/viewtopic.php?f=3&t=83.

STATIC
Loop iterations are divided into pieces of size *chunk* and then statically assigned to threads. If chunk is not specified, the iterations are evenly (if possible) divided contiguously among the threads.

DYNAMIC
Loop iterations are divided into pieces of size *chunk*, and dynamically scheduled among the threads; when a thread finishes one chunk, it is dynamically assigned another. The default chunk size is 1.

GUIDED
Iterations are dynamically assigned to threads in blocks as threads request them until no blocks remain to be assigned. Similar to DYNAMIC except that the block size decreases each time a parcel of work is given to a thread.
The size of the initial block is proportional to: **`number_of_iterations / number_of_threads`**
Subsequent blocks are proportional to **`number_of_iterations_remaining / number_of_threads`**
The chunk parameter defines the minimum block size. The default chunk size is 1.
Note: compilers differ in how GUIDED is implemented as shown in the "Guided A" and "Guided B" examples below.

RUNTIME
The scheduling decision is deferred until runtime by the environment variable OMP_SCHEDULE. It is illegal to specify a chunk size for this clause.

AUTO
The scheduling decision is delegated to the compiler and/or runtime system.