How OpenMP* is Compiled

Barbara Chapman

University of Houston

Acknowle dgeme nts :

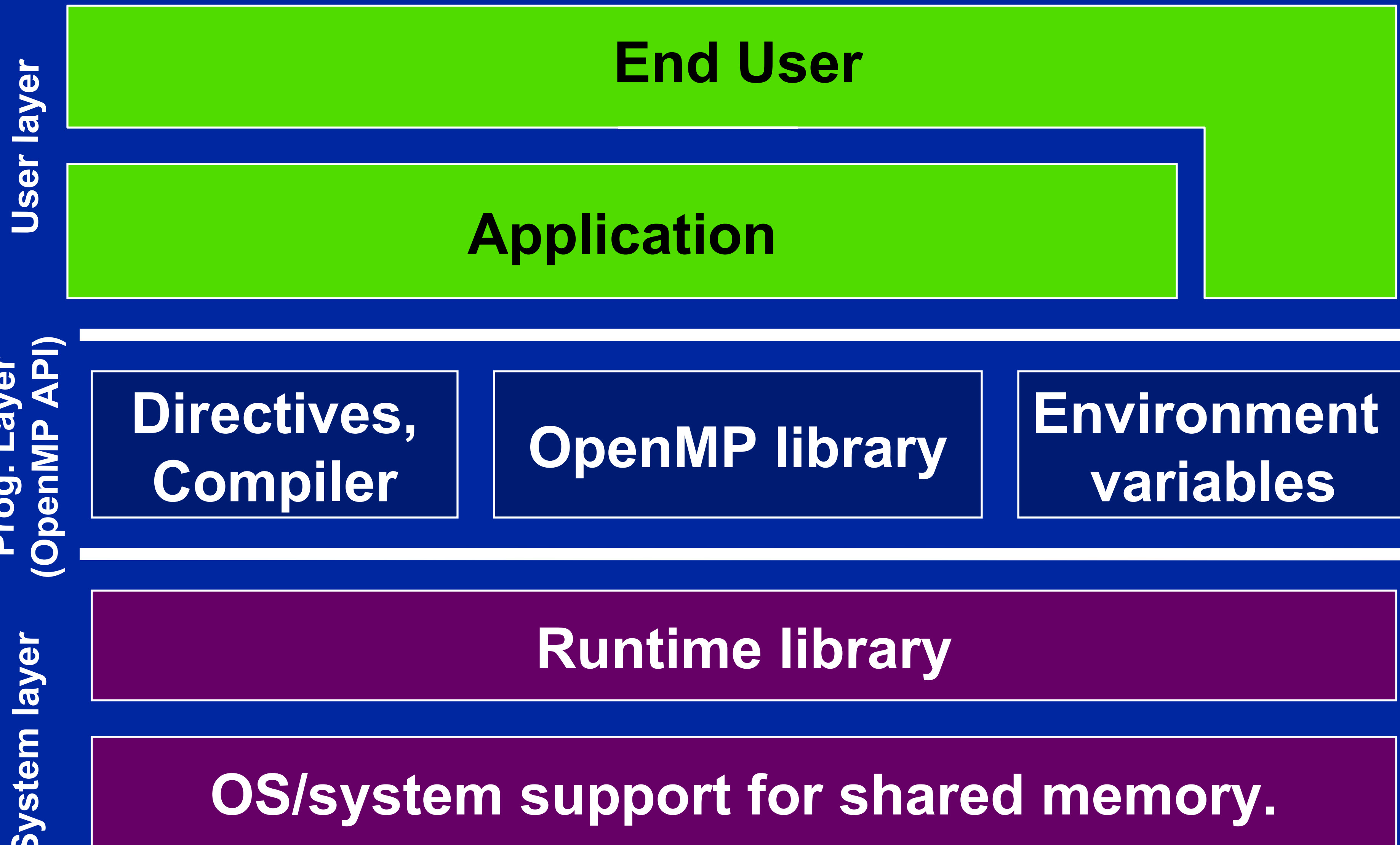
Slides here were also contributed by Chunhua Liao and Lei Huang from the HPCTools Group of the University of Houston.

* The name “OpenMP” is the property of the OpenMP Architecture Review Board.

How Does OpenMP Enable Us to Exploit Threads?

- OpenMP provides thread programming model at a "high level".
  - The user does not need to specify all the details
    - Especially with respect to the assignment of work to threads
    - Creation of threads
- User makes strategic decisions
- Compiler figures out details
- Alternatives:
  - MPI
  - POSIX thread library is lower level
  - Automatic parallelization is even higher level (user does nothing)
    - But usually successful on simple codes only

OpenMP Parallel Computing Solution Stack



Recall Basic Idea: How OpenMP Works

- User must decide what is parallel in program
  - Makes any changes needed to original source code
  - E.g. to remove any dependences in parts that should run in parallel
- User inserts directives telling compiler how statements are to be executed
  - what parts of the program are parallel
  - how to assign code in parallel regions to threads
  - what data is private (local) to threads

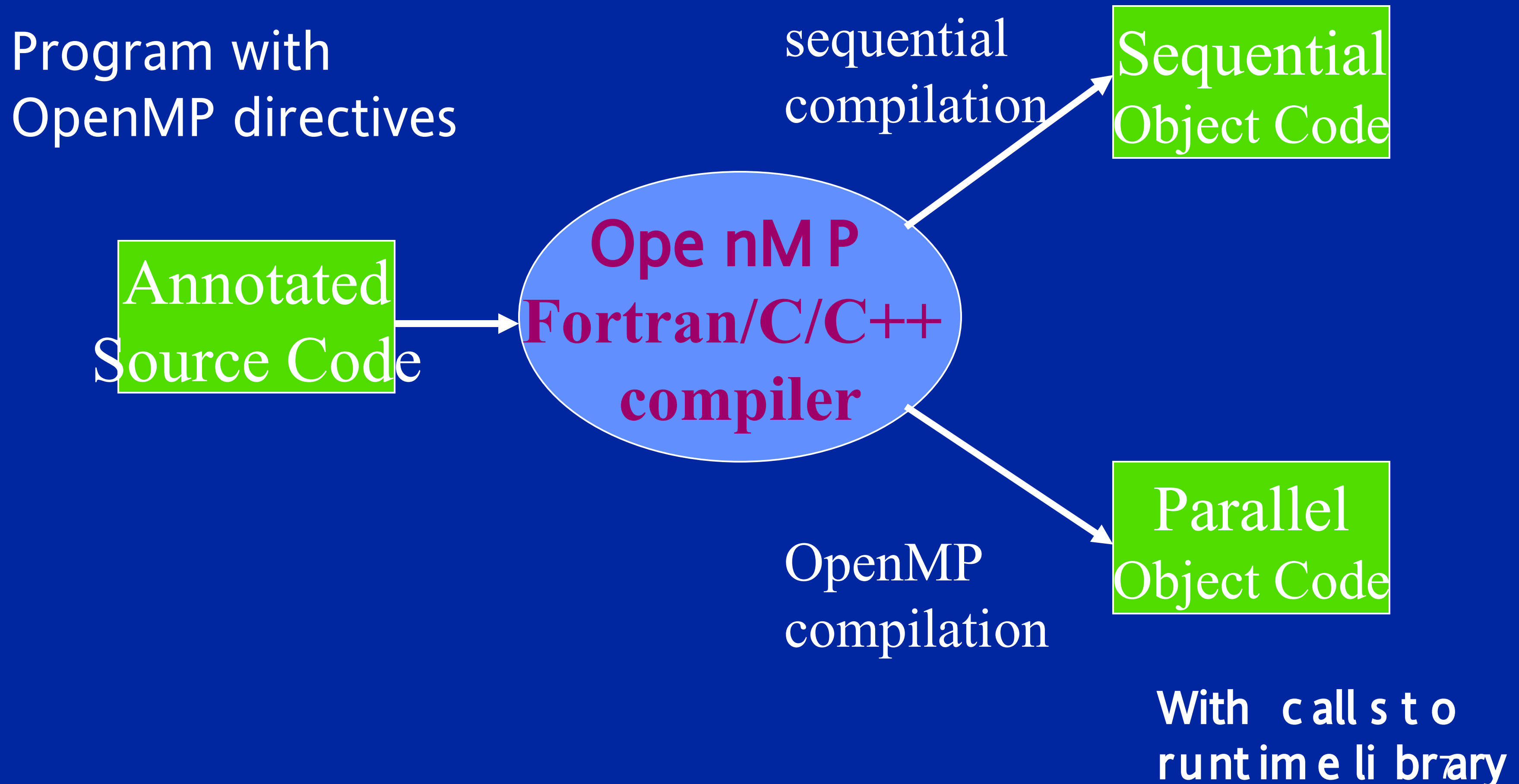
How The User Interacts with Compiler

- Compiler generates explicit threaded code
  - shields user from many details of the multithreaded code
- Compiler figures out details of code each thread needs to execute
- Compiler does not check that programmer directives are correct!
  - Programmer must be sure the required synchronization is inserted
- The result is a multithreaded object program

Recall Basic Idea of OpenMP

- The program generated by the compiler is executed by multiple threads
  - One thread per processor or core
- Each thread performs part of the work
  - Parallel parts executed by multiple threads
  - Sequential parts executed by single thread
- Dependences in parallel parts require synchronization between threads

OpenMP Implementation



OpenMP Implementation

- If program is compiled sequentially
  - OpenMP comments and pragmas are ignored
- If code is compiled for parallel execution
  - comments and/or pragmas are read, and
  - drive translation into parallel program
- Ideally, one source for both sequential and parallel program (big maintenance plus)

Usually this is accomplished by choosing a specific compiler option

How is OpenMP Invoked ?

The user provides the required option or switch

- Sometimes this also needs a specific optimization level, so manual should be consulted
- May also need to set threads' stacksize explicitly

Examples of compiler options

- Commercial:

  -openmp (Intel, Sun, NEC), -mp (SGI, PathScale, PGI), --openmp (Lahey, Fujitsu), -qsmp=omp (IBM) /openmp flag (Microsoft Visual Studio 2005), etc.
- Freeware: Omni, OdinMP, OMPi, OpenUH, …

Check information at http://www.compunity.org

How Does OpenMP Really Work?

We have seen what the application programmer does

- States what is to be carried out in parallel by multiple threads
- Gives strategy for assigning work to threads
- Arranges for threads to synchronize
- Specify data sharing attributes: shared, private, firstprivate, threadprivate,…

Overview of OpenMP Translation Process

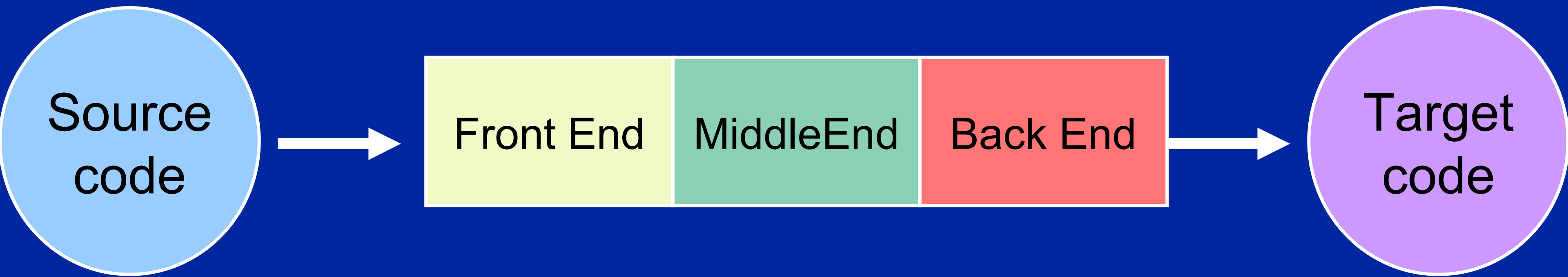
- Compiler processes directives and uses them to create explicitly multithreaded code
- Generated code makes calls to a runtime library
  - The runtime library also implements the OpenMP user-level run-time routines
- Details are different for each compiler, but strategies are similar
- Runtime library and details of memory management also proprietary
- Fortunately the basic translation is not all that difficult

The OpenMP Implementation…

- Transforms OpenMP programs into multi-threaded code
- Figures out the details of the work to be performed by each thread
- Arranges storage for different data and performs
- their initializations: shared, private…
- Manages threads: creates, suspends, wakes up, terminates threads
- Implements thread synchronization

The details of how OpenMP is implemented varies from one compiler to another. We can only give an idea of how it is done here!!

Structure of a Compiler



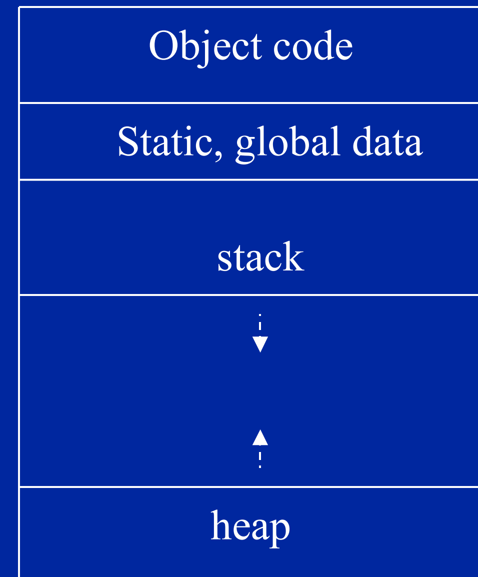
- Front End:
  - Read in source program, ensure that it is error-free, build the intermediate representation (IR)
- Middle End:
  - Analyze and optimize program as much as possible. "Lower" IR to machine-like form
- Back End:
  - Determine layout of program data in memory. Generate object code for the target architecture and optimize it

Compiler Sets Up Memory Allocation

At *run time*, code and objects must have locations in memory. The compiler arranges for this

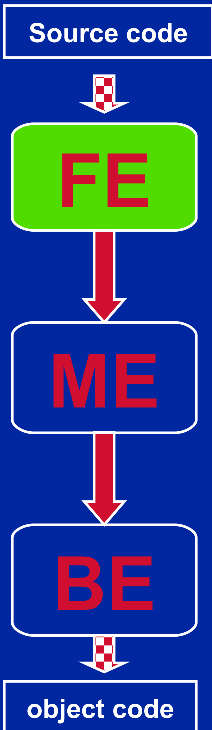
(Not all programming languages need a heap: e.g. Fortran 77 doesn't, C does.)

- *Stack and heap grow and shrink over time*
- *Grow toward each other*
- *Very old strategy*
- *Code, data may be interleaved*



But in a multithreaded program, each thread needs its own stack

OpenMP Compiler Front End

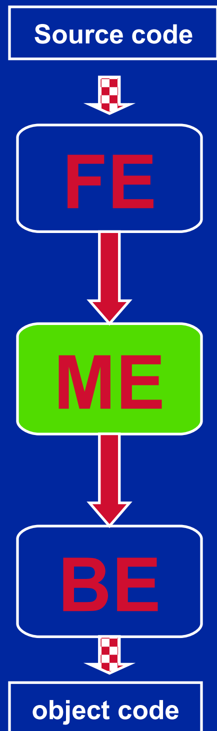


In addition to reading in the base language (Fortran, C or C++)

- Read (parse) OpenMP directives
- Check them for correctness
    - Is directive in the right place? Is the information correct? Is the form of the for loop permitted? ….
- Create an intermediate representation with OpenMP annotations for further handling

Nasty problem: incorrect OpenMP sentinel means directive may not be recognized. And there might be no error message!!

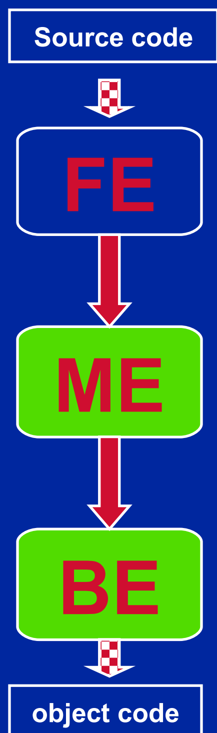
OpenMP Compiler Middle End



- Preprocess OpenMP constructs
  - Translate SECTIONs to DO/FOR constructs
  - Make implicit BARRIERs explicit
  - Apply even more correctness checks
- Apply some optimizations to code to ensure it performs well
  - Merge adjacent parallel regions
  - Merge adjacent barriers

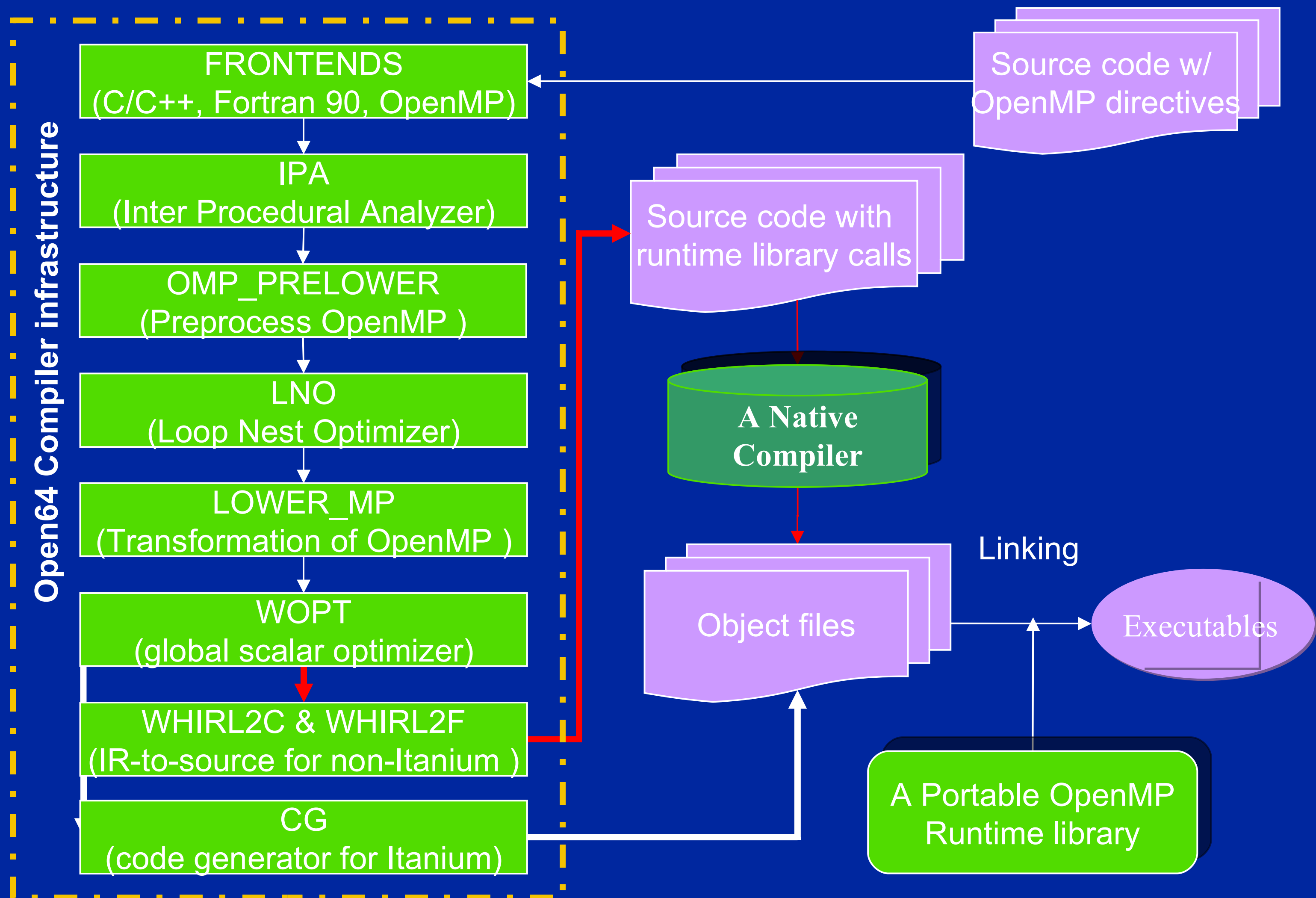
OpenMP directives reduce scope in which some optimizations can be applied. Compiler writer must work hard to avoid a negative impact on performance.

OpenMP Compiler: Rest of Processing



- Translate OpenMP constructs to multithreaded code
  - Sometimes simple
    - Replace certain OpenMP constructs by calls to runtime routines.
    - e.g.: barrier, atomic, flush, etc
  - Sometimes a little more complex
    - Implement parallel construct by creating a separate task that contains the code in a parallel region
    - For master thread: fork slave threads so they execute their tasks, as well as carrying out the task along with slave threads.
    - Add necessary synchronization via runtime library
    - Translate parallel and worksharing constructs and clauses e.g.: parallel, for, etc
- Also implement variable data attributes, set up storage and arrange for initialization
  - Thread's stack might be used to hold all private data
  - Instantiate new variables to implement private, reduction, etc
  - Add assignment statements to realize firstprivate, lastprivate, etc

OpenUH Compiler Infrastructure



Collaboration between University of Houston and Tsinghua University

Implementing a Parallel Region: Outlining

Compiler creates a new procedure containing the region enclosed by a parallel construct

- Each thread will execute this procedure
- Shared data passed as arguments
  - Referenced via their address in routine
- Private data stored on thread's stack
  - Threadprivate may be on stack or heap

Outlining introduces a few overheads, but makes the translation straightforward.

It makes the scope of OpenMP data attributes explicit.

An Outlining Example: Hello world

- **Original Code**

```
#include <omp.h>
void main()
{
#pragma omp parallel
 {
 int ID=omp_get_thread_num();
 printf("Hello world(%d)",ID);
 }
}
```

- Translated multi-threaded code with runtime library calls

```
//here is the outlined code
void __ompregion_main1(…)
{
 int ID =ompc_get_thread_num();
 printf("Hello world(%d)",ID);
} /* end of ompregion_main1*/

void main()
{
…
__ompc_fork(&__ompregion_main1,…);
…
}
```

OpenMP Transformations – Do/For

- Transform original loop so each thread performs only its own portion
- Most of scheduling calculations usually hidden in runtime
- Some extra work to handle firstprivate, lastprivate

- Original Code

```
#pragma omp for
for( i = 0; i < n; i++ )
{   …}
```

- Transformed Code

```
tid = ompc_get_thread_num();
ompc_static_init (tid, lower,upper,
incr,.);
for( i = lower;i < upper;i += incr )
{  …  }
// Implicit BARRIER
ompc_barrier();
```

OpenMP Transformations – Reduction

- **Reduction variables can be translated into a two-step operation**
- **First, each thread performs its own reduction using a private variable**
- **Then the global sum is formed**
- **The compiler must ensure atomicity of the final reduction**

- Original Code

```
#pragma omp parallel for \
reduction (+:sum) private (x)
for(i=1;i<=num_steps;i++)
{  …
sum=sum+x ;}
```

- Transformed Code

```
float local_sum;
…
ompc_static_init (tid, lower,uppder,
incr,.);
for( i = lower;i < upper;i += incr )
{ … local_sum = local_sum +x;}
ompc_barrier();
ompc_critical();
sum = (sum + local_sum);
ompc_end_critical();
```

OpenMP Transformation – Single/Master

- **Master thread has a threadid of 0, very easy to test for.**
- **The runtime function for the single construct might use a lock to test and set an internal flag in order to ensure only one thread get the work done**

- Original Code

```
#pragma omp parallel
{#pragma omp master
  a=a+1;
#pragma omp single
  b=b+1;}
```

- Transformed Code

```
Is_master= ompc_master(tid);
if((Is_master == 1))
{   a = a + 1;   }
Is_single = ompc_single(tid);
if((Is_single == 1))
{   b = b + 1;   }
ompc_barrier();
```

OpenMP Transformations – Threadprivate

- Every threadprivate variable reference becomes an indirect reference through an auxiliary structure to the private copy
- Every thread needs to find its index into the auxiliary structure – This can be expensive
  - Some OS'es (and codegen schemes) dedicate register to identify thread
  - Otherwise OpenMP runtime has to do this

- Original Code

```
static int px;

int foo() {
 #pragma omp threadprivate(px)
  bar( &px );
 }
```

- Transformed Code

```
static int px;
static int ** thdprv_px;

int _ompregion_foo1() {
int* local_px;
 …
tid = ompc_get_thread_num();
local_px=get_thdprv(tid,thdprv_px,
 &px);
bar( local_px );
}
```

OpenMP Transformations – WORKSHARE

- WORKSHARE can be translated to OMP DO during preprocessing phase
- If there are several different array statements involved, it requires a lot of work by the compiler to do a good job
- So there may be a performance penalty

- Original Code

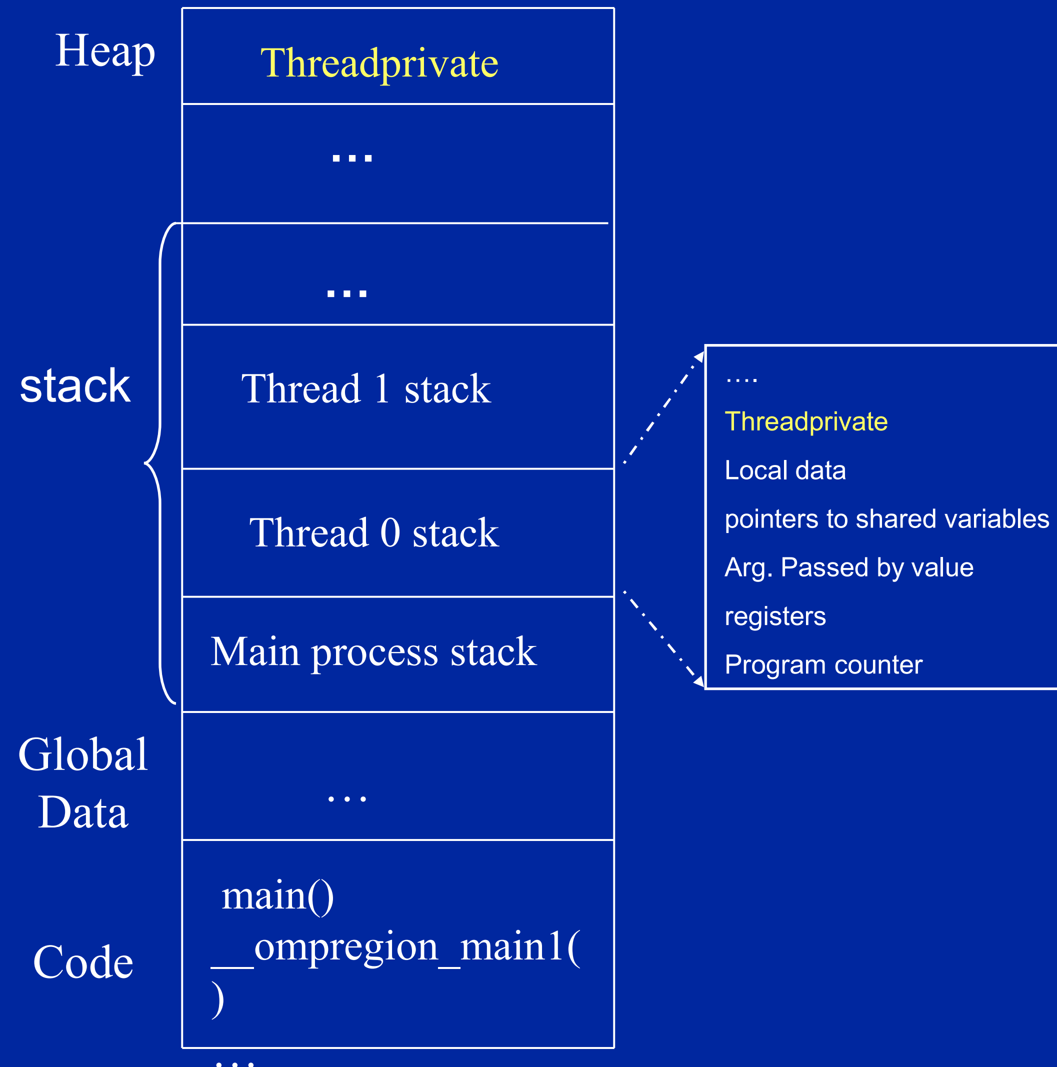
```
      REAL AA(N,N), BB(N,N)
     !$OMP PARALLEL
     !$OMP WORKSHARE
           AA = BB
     !$OMP END WORKSHARE
     !$OMP END PARALLEL
```

- Transformed Code

```
  REAL AA(N,N), BB(N,N)
  !$OMP PARALLEL
  !$OMP DO
     DO J=1,N,1
        DO I=1,N,1
           AA(I,J) = BB(I,J)
        END DO
     END DO
  !$OMP END PARALLEL
```

Runtime Memory Allocation

One possible organization of memory



- **Outlining creates a new scope: private data become local variables for the outlined routine.**
- **Local variables can be saved on stack**
  - **Includes compiler-generated** temporaries
  - Private **variables, including** firstprivate **and** lastprivate
  - **Could be a lot of data**
  - **Local variables in a procedure called within a parallel region are private by default**
- Location of threadprivate data depends on implementation
  - **On heap**
  - **On local stack**

Role of Runtime Library

- Thread management and work dispatch
  - Routines to create threads, suspend them and wake them up/ spin them, destroy threads
  - Routines to schedule work to threads
    - Manage queue of work
    - Provide schedulers for static, dynamic and guided
- Maintain internal control variables
  - threadid, numthreads, dyn-var, nest-var, sched_var, etc
- Implement library routines omp_..() and some simple constructs (e.g. barrier, atomic)

Some routines in runtime library – e.g. to return the threadid - are heavily accessed, so they must be carefully implemented and tuned. The runtime library should avoid any unnecessary internal synchronization.

Synchronization

- Barrier is main synchronization construct since many other constructs may introduce it implicitly. It in turn is often implemented using locks.

One simple way to implement barrier

• Each thread team maintains a barrier counter and a barrier flag.

• Each thread increments the barrier counter when it enters the barrier and waits for a barrier flag to be set by the last one.

• When the last thread enters the barrier and increment the counter, the counter will be equal to the team size and the barrier flag is reset.

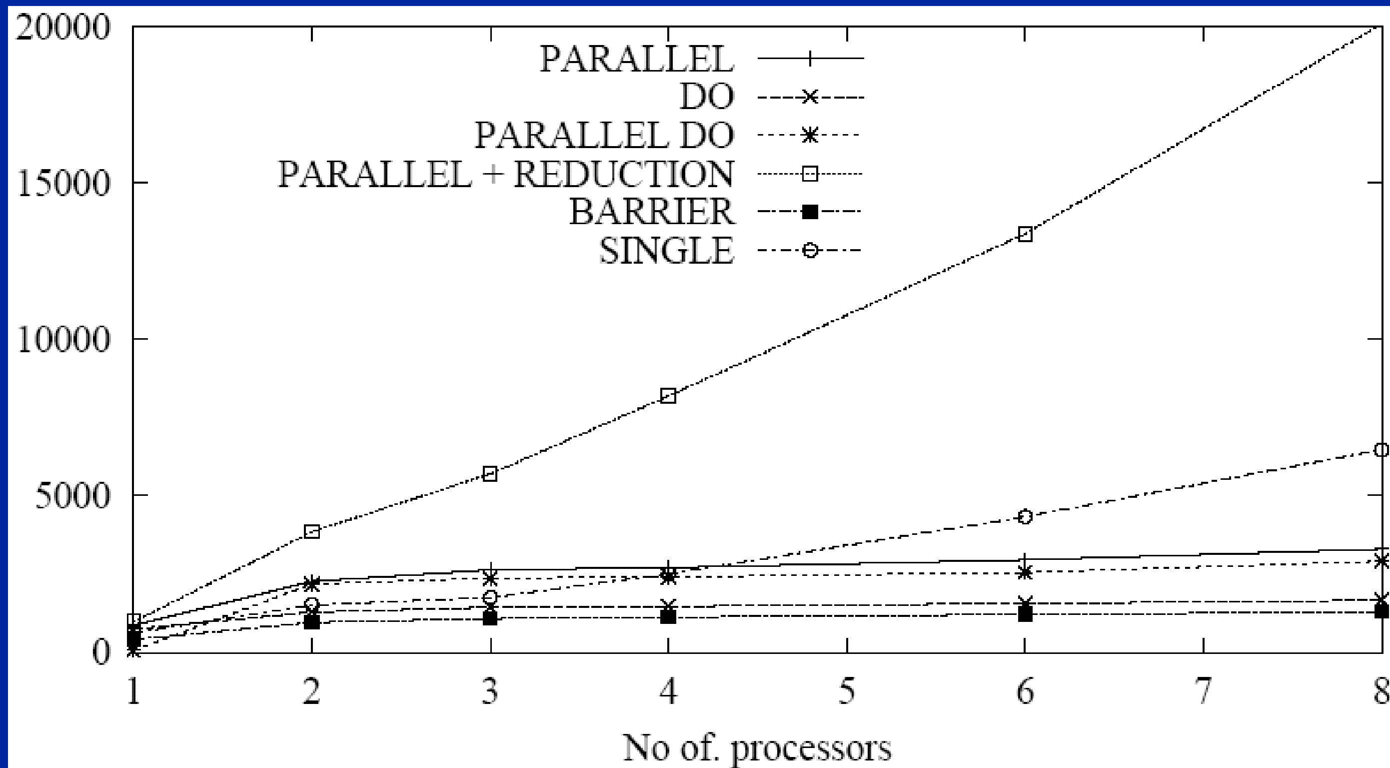
• All other waiting threads can then proceed.

```
void __ompc_barrier (omp_team_t *team)
{
        …
        pthread_mutex_lock(&(team->barrier_lock));
        team->barrier_count++;
        barrier_flag = team->barrier_flag;

       /* The last one reset flags*/
        if (team->barrier_count == team->team_size)
        {
              team->barrier_count = 0;
               team->barrier_flag = barrier_flag ^ 1; /* Xor: toggle*/
               pthread_mutex_unlock(&(team->barrier_lock));
               return;
        }
        pthread_mutex_unlock(&(team->barrier_lock));

        /* Wait for the last to reset the barrier*/
        OMPC_WAIT_WHILE(team->barrier_flag == barrier_flag);
}
```

Constructs That Use a Barrier



Synchronization Overheads (in cycles) on SGI Origin 2000*

- Careful implementation can achieve modest overhead for most synchronization constructs.
- Parallel reduction is costly because it often uses critical region to summarize variables at the end.

* Courtesy of J. M. Bull, "Measuring Synchronisation and Scheduling Overheads in OpenMP", EWOMP '99, Lund, Sep., 1999.

Static Scheduling: Under The Hood

```
// The OpenMP code
// possible unknown loop upper bound: n
// unknown number of threads to be used
#pragma omp for
schedule(static)
  for (i=0;i<n;i++)
   {
    do_sth();
   }
```

```
/ *Static even: static without specifying
    chunk size; scheduler divides loop
    iterations evenly onto each thread. */
// the outlined task for each thread
 _gtid_s1 = __ompc_get_thread_num();
  temp_limit = n – 1
  __ompc_static_init(_gtid_s1 , static,
    &_do_lower, &_do_upper, &_do_stride,..);
  if(_do_upper > temp_limit)
  {    _do_upper = temp_limit;  }
 for(_i = _do_lower; _i <= _do_upper; _i ++)
  {
      do_sth();
  }
```

- Most (if not all) OpenMP compilers choose static as default scheduling method
- Number of threads and loop bounds possibly unknown, so final details usually deferred to runtime
- Two simple runtime library calls are enough to handle static case:
  Constant overhead

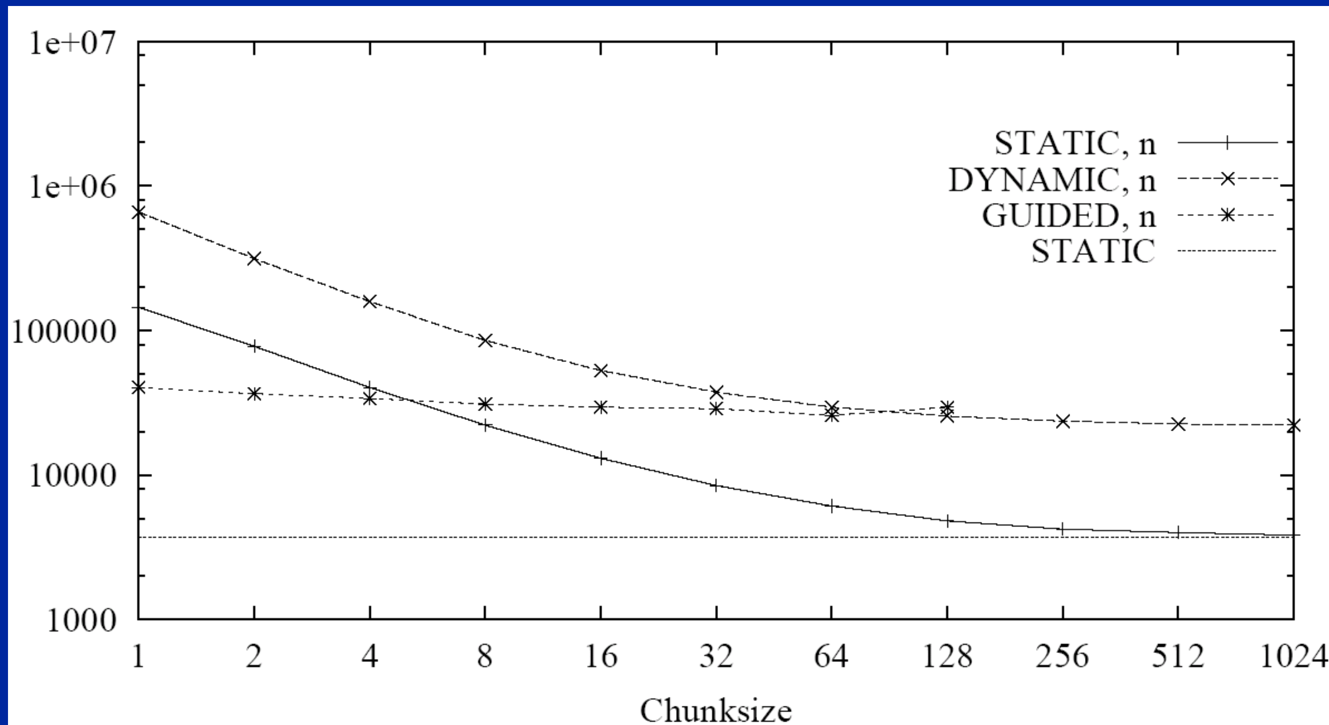
Dynamic Scheduling : Under The Hood

```
_gtid_s1 = __ompc_get_thread_num();
 temp_limit = n -1;                          // Schedule(dynamic, chunksize)
_do_upper = temp_limit;
_do_lower = 0;
__ompc_scheduler_init(__ompv_gtid_s1, dynamic ,do_lower, _do_upper, stride, chunksize..);
_i = _do_lower;
mpni_status = __ompc_schedule_next(_gtid_s1, &_do_lower, &_do_upper, &_do_stride);
while(mpni_status)
{
  if(_do_upper > temp_limit)
  {      _do_upper = temp_limit;    }
  for(_i = _do_lower; _i <= _do_upper; _i = _i + _do_stride)
  {          do_sth();    }
  mpni_status = __ompc_schedule_next(_gtid_s1, &_do_lower, &_do_upper, &_do_stride);
}
```

- Scheduling is performed during runtime.
- A while loop to grab available loop iterations from a work queue
  - Similar way to implement STATIC with a chunk size and GUIDED scheduling

Average overhead= c1*(iteration space/chunksize)+c2

Using OpenMP Scheduling Constructs



Scheduling Overheads (in cycles) on Sun HPC 3500*

- **Conclusion:**
  - **Use default static scheduling when work load is balanced and thread processing capability is constant.**
  - **Use dynamic/guided otherwise**

* Courtesy of J. M. Bull, "Measuring Synchronization and Scheduling Overheads in OpenMP", EWOMP '99, Lund, Sep., 1999.

Implementation-Defined Issues

- OpenMP also leaves some issues to the implementation
  - Default number of threads
  - Default schedule and default for schedule (runtime)
  - Number of threads to execute nested parallel regions
  - Behavior in case of thread exhaustion
  - And many others..

Despite many similarities, each implementation is a little different from all others.

Recap

- OpenMP-aware compiler uses directives to generate code for each thread
- It also arranges for the program's data to be stored in memory
- To do this, it:
  - Creates a new procedure for each parallel region
  - Gets each thread to invoke this procedure with the required arguments
  - Has each thread compute its set of iterations for a parallel loop
  - Uses runtime routines to implement synchronization as well as many other details of parallel object code
- Get to "know" a compiler by running microbenchmarks to see overheads (visit http://www.epcc.ed.ac.uk/~jmbull for more)

Questions?