FTC Optimization Notice

Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel.

Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Notice revision #20110804

Trademarks

Intel, Xeon, and Intel Xeon Phi are trademarks of Intel Corporation in the U.S. and/or other countries.

Other names and brands may be claimed as the property of others.

The OpenMP name and the OpenMP logo are registered trademarks of the OpenMP Architecture Review Board.

This document is Copyright ©2013, Intel Corporation. All rights reserved.
Contents

1 Intel® OpenMP® Runtime Library Interface .................................................. 1
  1.1 Introduction ......................................................................................... 1
  1.2 Building the Runtime .......................................................................... 1
  1.3 Supported RTL Build Configurations .................................................. 2
  1.4 Front-end Compilers that work with this RTL ...................................... 2
  1.5 Outlining ............................................................................................. 2
    1.5.1 Addressing shared variables ......................................................... 3
      1.5.1.1 Current Technique .................................................................. 3
      1.5.1.2 Future Technique .................................................................. 3
  1.6 Library Interfaces .................................................................................. 3
  1.7 Examples ............................................................................................. 4
    1.7.1 Work Sharing Example .................................................................. 4

2 Module Index ............................................................................................... 7
  2.1 Modules ................................................................................................ 7

3 Hierarchical Index ......................................................................................... 9
  3.1 Class Hierarchy .................................................................................... 9

4 Class Index .................................................................................................. 11
  4.1 Class List ............................................................................................. 11

5 Module Documentation ................................................................................. 13
  5.1 Atomic Operations ................................................................................ 13
    5.1.1 Detailed Description .................................................................... 13
  5.2 Wait/Release operations ......................................................................... 20
    5.2.1 Detailed Description .................................................................... 20
    5.2.2 Enumeration Type Documentation ................................................ 20
      5.2.2.1 flag_type .............................................................................. 20
  5.3 Basic Types ........................................................................................... 21
    5.3.1 Detailed Description .................................................................... 21
    5.3.2 Macro Definition Documentation ................................................ 21
      5.3.2.1 KMP_IDENT_ATOMIC_REDUCE ........................................... 21
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.9.2.4 __kmpc_end_barrier_master</td>
<td>39</td>
</tr>
<tr>
<td>5.9.2.5 __kmpc_end_reduce</td>
<td>39</td>
</tr>
<tr>
<td>5.9.2.6 __kmpc_end_reduce_nowait</td>
<td>39</td>
</tr>
<tr>
<td>5.9.2.7 __kmpc_flush</td>
<td>39</td>
</tr>
<tr>
<td>5.9.2.8 __kmpc_reduce</td>
<td>39</td>
</tr>
<tr>
<td>5.9.2.9 __kmpc_reduce_nowait</td>
<td>40</td>
</tr>
<tr>
<td>5.10 Thread private data support</td>
<td>41</td>
</tr>
<tr>
<td>5.10.1 Detailed Description</td>
<td>41</td>
</tr>
<tr>
<td>5.10.2 Typedef Documentation</td>
<td>41</td>
</tr>
<tr>
<td>5.10.2.1 kmpc_cctor</td>
<td>41</td>
</tr>
<tr>
<td>5.10.2.2 kmpc_cctor_vec</td>
<td>41</td>
</tr>
<tr>
<td>5.10.2.3 kmpc_ctor</td>
<td>41</td>
</tr>
<tr>
<td>5.10.2.4 kmpc_ctor_vec</td>
<td>41</td>
</tr>
<tr>
<td>5.10.2.5 kmpc_dtor</td>
<td>41</td>
</tr>
<tr>
<td>5.10.2.6 kmpc_dtor_vec</td>
<td>42</td>
</tr>
<tr>
<td>5.10.3 Function Documentation</td>
<td>42</td>
</tr>
<tr>
<td>5.10.3.1 __kmpc_copyprivate</td>
<td>42</td>
</tr>
<tr>
<td>5.10.3.2 __kmpc_threadprivate Cached</td>
<td>42</td>
</tr>
<tr>
<td>5.10.3.3 __kmpc_threadprivate_register</td>
<td>43</td>
</tr>
<tr>
<td>5.10.3.4 __kmpc_threadprivate_register_vec</td>
<td>43</td>
</tr>
<tr>
<td>5.11 Statistics Gathering from OMPTB</td>
<td>44</td>
</tr>
<tr>
<td>5.11.1 Detailed Description</td>
<td>44</td>
</tr>
<tr>
<td>5.11.2 Environment Variables</td>
<td>44</td>
</tr>
<tr>
<td>5.11.3 Macro Definition Documentation</td>
<td>45</td>
</tr>
<tr>
<td>5.11.3.1 KMP_COUNT_BLOCK</td>
<td>45</td>
</tr>
<tr>
<td>5.11.3.2 KMP_COUNT_VALUE</td>
<td>45</td>
</tr>
<tr>
<td>5.11.3.3 KMP_FOREACH_COUNTER</td>
<td>45</td>
</tr>
<tr>
<td>5.11.3.4 KMP_FOREACH_EXPLICIT_TIMER</td>
<td>46</td>
</tr>
<tr>
<td>5.11.3.5 KMP_INIT_PARTITIONED_TIMERS</td>
<td>46</td>
</tr>
<tr>
<td>5.11.3.6 KMP_OUTPUT_STATS</td>
<td>47</td>
</tr>
<tr>
<td>5.11.3.7 KMP_RESET_STATS</td>
<td>47</td>
</tr>
<tr>
<td>5.11.3.8 KMP_START_EXPLICIT_TIMER</td>
<td>47</td>
</tr>
<tr>
<td>5.11.3.9 KMP_STOP_EXPLICIT_TIMER</td>
<td>47</td>
</tr>
<tr>
<td>5.11.3.10 KMP_TIME_BLOCK</td>
<td>48</td>
</tr>
<tr>
<td>5.11.4 Enumeration Type Documentation</td>
<td>48</td>
</tr>
<tr>
<td>5.11.4.1 stats_flags_e</td>
<td>48</td>
</tr>
<tr>
<td>5.12 Tasking support</td>
<td>49</td>
</tr>
<tr>
<td>5.12.1 Detailed Description</td>
<td>49</td>
</tr>
<tr>
<td>5.12.2 Function Documentation</td>
<td>49</td>
</tr>
<tr>
<td>5.12.2.1 __kmpc_omp_task_with_deps</td>
<td>49</td>
</tr>
</tbody>
</table>
# Class Documentation

## hierarchy_info Class Reference

### Detailed Description

### Member Data Documentation

- **depth**: 51
- **maxLeaves**: 51
- **maxLevels**: 52
- **numPerLevel**: 52

## ident Struct Reference

### Detailed Description

### Member Data Documentation

- **flags**: 52
- **psource**: 52
- **reserved_1**: 53
- **reserved_2**: 53
- **reserved_3**: 53

## kmp_flag\(<P>\) Class Template Reference

### Detailed Description

### Member Function Documentation

- **get**: 53
- **get_type**: 54
- **set**: 54

### Member Data Documentation

- **loc**: 54
- **t**: 54

---

Index

54
Chapter 1

Intel® OpenMP® Runtime Library Interface

1.1 Introduction

This document describes the interface provided by the Intel® OpenMP® runtime library to the compiler. Routines that are directly called as simple functions by user code are not currently described here, since their definition is in the OpenMP specification available from http://openmp.org

The aim here is to explain the interface from the compiler to the runtime.

The overall design is described, and each function in the interface has its own description. (At least, that's the ambition, we may not be there yet).

1.2 Building the Runtime

For the impatient, we cover building the runtime as the first topic here.

A top-level Makefile is provided that attempts to derive a suitable configuration for the most commonly used environments. To see the default settings, type:

% make info

You can change the Makefile's behavior with the following options:

- **omp_root**: The path to the top-level directory containing the top-level Makefile. By default, this will take on the value of the current working directory.

- **omp_os**: Operating system. By default, the build will attempt to detect this. Currently supports "linux", "macos", and "windows".

- **arch**: Architecture. By default, the build will attempt to detect this if not specified by the user. Currently supported values are
  - "32" for IA-32 architecture
  - "32e" for Intel® 64 architecture
  - "mic" for Intel® Many Integrated Core Architecture (If "mic" is specified then "icc" will be used as the compiler, and appropriate k1om binutils will be used. The necessary packages must be installed on the build machine for this to be possible, but an Intel® Xeon Phi™ coprocessor is not required to build the library).

- **compiler**: Which compiler to use for the build. Defaults to "icc" or "icl" depending on the value of omp_os. Also supports "gcc" when omp_os is "linux" for gcc* versions 4.6.2 and higher. For icc on OS X®, OS X®
versions greater than 10.6 are not supported currently. Also, icc version 13.0 is not supported. The selected compiler should be installed and in the user’s path. The corresponding Fortran compiler should also be in the path.

• **mode**: Library mode: default is "release". Also supports "debug".

To use any of the options above, simple add `<option_name>=<value>`. For example, if you want to build with gcc instead of icc, type:

% make compiler=gcc

Underneath the hood of the top-level Makefile, the runtime is built by a perl script that in turn drives a detailed runtime system make. The script can be found at `tools/build.pl`, and will print information about all its flags and controls if invoked as

% tools/build.pl --help

If invoked with no arguments, it will try to build a set of libraries that are appropriate for the machine on which the build is happening. There are many options for building out of tree, and configuring library features that can also be used. Consult the --help output for details.

### 1.3 Supported RTL Build Configurations

The architectures supported are IA-32 architecture, Intel® 64, and Intel® Many Integrated Core Architecture. The build configurations supported are shown in the table below.

<table>
<thead>
<tr>
<th></th>
<th>icc/icl</th>
<th>gcc</th>
</tr>
</thead>
<tbody>
<tr>
<td>Linux* OS</td>
<td>Yes(1,5)</td>
<td>Yes(2,4)</td>
</tr>
<tr>
<td>OS X*</td>
<td>Yes(1,3,4)</td>
<td>No</td>
</tr>
<tr>
<td>Windows* OS</td>
<td>Yes(1,4)</td>
<td>No</td>
</tr>
</tbody>
</table>

(1) On IA-32 architecture and Intel® 64, icc/icl versions 12.x are supported (12.1 is recommended).
(2) gcc version 4.6.2 is supported.
(3) For icc on OS X*, OS X* version 10.5.8 is supported.
(4) Intel® Many Integrated Core Architecture not supported.
(5) On Intel® Many Integrated Core Architecture, icc/icl versions 13.0 or later are required.

### 1.4 Front-end Compilers that work with this RTL

The following compilers are known to do compatible code generation for this RTL: icc/icl, gcc. Code generation is discussed in more detail later in this document.

### 1.5 Outlining

The runtime interface is based on the idea that the compiler "outlines" sections of code that are to run in parallel into separate functions that can then be invoked in multiple threads. For instance, simple code like this

```c
void foo()
{
    #pragma omp parallel
    {
        ...do something...
    }
}
```
is converted into something that looks conceptually like this (where the names used are merely illustrative; the real library function names will be used later after we’ve discussed some more issues...)

```c
static void outlinedFooBody()
{
    ... do something ...
}

void foo()
{
    __OMP_runtime_fork(outlinedFooBody, (void*)0); // Not the real function name!
}
```

1.5.1 Addressing shared variables

In real uses of the OpenMP API there are normally references from the outlined code to shared variables that are in scope in the containing function. Therefore the containing function must be able to address these variables. The runtime supports two alternate ways of doing this.

1.5.1.1 Current Technique

The technique currently supported by the runtime library is to receive a separate pointer to each shared variable that can be accessed from the outlined function. This is what is shown in the example below.

We hope soon to provide an alternative interface to support the alternate implementation described in the next section. The alternative implementation has performance advantages for small parallel regions that have many shared variables.

1.5.1.2 Future Technique

The idea is to treat the outlined function as though it were a lexically nested function, and pass it a single argument which is the pointer to the parent's stack frame. Provided that the compiler knows the layout of the parent frame when it is generating the outlined function it can then access the up-level variables at appropriate offsets from the parent frame. This is a classical compiler technique from the 1960s to support languages like Algol (and its descendants) that support lexically nested functions.

The main benefit of this technique is that there is no code required at the fork point to marshal the arguments to the outlined function. Since the runtime knows statically how many arguments must be passed to the outlined function, it can easily copy them to the thread's stack frame. Therefore the performance of the fork code is independent of the number of shared variables that are accessed by the outlined function.

If it is hard to determine the stack layout of the parent while generating the outlined code, it is still possible to use this approach by collecting all of the variables in the parent that are accessed from outlined functions into a single `struct` which is placed on the stack, and whose address is passed to the outlined functions. In this way the offsets of the shared variables are known (since they are inside the struct) without needing to know the complete layout of the parent stack-frame. From the point of view of the runtime either of these techniques is equivalent, since in either case it only has to pass a single argument to the outlined function to allow it to access shared variables.

A scheme like this is how gcc∗ generates outlined functions.

1.6 Library Interfaces

The library functions used for specific parts of the OpenMP∗ language implementation are documented in different modules.

- **Basic Types** fundamental types used by the runtime in many places
- ** Deprecated Functions** functions that are in the library but are no longer required
- **Startup and Shutdown** functions for initializing and finalizing the runtime
- **Parallel (fork/join)** functions for implementing `omp parallel`
- **Thread Information** functions for supporting thread state inquiries
- **Work Sharing** functions for work sharing constructs such as `omp for`, `omp sections`
- **Thread private data support** functions to support thread private data, copyin etc
- **Synchronization** functions to support `omp critical`, `omp barrier`, `omp master`, `reductions` etc
- **Atomic Operations** functions to support atomic operations
- **Statistics Gathering from OMPTB** macros to support developer profiling of libiomp5
- Documentation on tasking has still to be written...

1.7 Examples

1.7.1 Work Sharing Example

This example shows the code generated for a parallel for with reduction and dynamic scheduling.

```c
extern float foo( void );

int main () {
    int i;
    float r = 0.0;
    #pragma omp parallel for schedule(dynamic) reduction(+:r)
    for ( i = 0; i < 10; i ++ ) {
        r += foo();
    }
}
```

The transformed code looks like this.

```c
extern float foo( void );

int main () {
    static int zero = 0;
    auto int gtid;
    auto float r = 0.0;
    __kmpc_begin( & loc3, 0 );
    // The gtid is not actually required in this example so could be omitted;
    // We show its initialization here because it is often required for calls into
    // the runtime and should be locally cached like this.
    gtid = __kmpc_global thread num( & loc3 );
    __kmpc_fork call( & loc7, 1, main_7_parallel_3, & r );
    __kmpc_end( & loc0 );
    return 0;
}
```

```c
struct main_10_reduction_t_5 { float r_10_rpr; }
static kmp_critical_name lck = { 0 }; // loc10.flags should contain KMP_IDENT_ATOMIC_REDUCE bit set
// if compiler has generated an atomic reduction.
void main_7_parallel_3( int *gtid, int *btid, float *r_7_shp ) {
    auto int i_7_pr;
    auto int lower, upper, liter, incr;
    auto struct main_10_reduction_t_5 reduce;
    reduce.r_10_rpr = 0.F;
    liter = 0;
    __kmpc_dispatch_init_4( & loc7,*gtid, 35, 0, 9, 1, 1 );
    while ( __kmpc_dispatch_next_4( & loc7,*gtid, & liter, & lower, & upper, & incr ) ) {
        for( i_7_pr = lower; upper >= i_7_pr; i_7_pr ++ )
            reduce.r_10_rpr += foo();
    }
    switch( __kmpc_reduce_nowait{ & loc10, *gtid, 1, 4, & reduce, main_10_reduce_5, & lck } ) { case 1:
        *r_7_shp += reduce.r_10_rpr;
        __kmpc_end_reduce_nowait{ & loc10, *gtid, & lck };
        break;
```

Generated on Mon Aug 8 2016 17:18:34 for Intel® OpenMP® Runtime Library by Doxygen
case 2:
    __kmpe_atomic_float4_add( & loc10, *gtid, r_7_shp, reduce.r_10_rpr);
    break;
default;
}

void main_10_reduce_5( struct main_10_reduction_t_5 *reduce_lhs,
                        struct main_10_reduction_t_5 *reduce_rhs )
{
    reduce_lhs->r_10_rpr += reduce_rhs->r_10_rpr;
}
Chapter 2

Module Index

2.1 Modules

Here is a list of all modules:

- Atomic Operations ............................................. 13
- Wait/Release operations ...................................... 20
- Basic Types ..................................................... 21
- Deprecated Functions ........................................ 23
- Startup and Shutdown ......................................... 24
- Parallel (fork/join) ............................................ 25
- Thread Information ............................................ 27
- Work Sharing ................................................... 29
- Synchronization ............................................... 38
- Thread private data support ................................. 41
- Statistics Gathering from OMPTB ......................... 44
- Tasking support ............................................... 49
- User visible functions ....................................... 50
Chapter 3

Hierarchical Index

3.1 Class Hierarchy

This inheritance list is sorted roughly, but not completely, alphabetically:

- hierarchy_info ................................................. 51
- ident ............................................................... 52
- kmp_flag< P > .................................................. 53
- kmp_flag< FlagType > ......................................... 53
- kmp_flag< kmp_uint32 > ....................................... 53
- kmp_flag< kmp_uint64 > ....................................... 53
Chapter 4

Class Index

4.1 Class List

Here are the classes, structs, unions and interfaces with brief descriptions:

- hierarchy_info ............................................................. 51
- ident ........................................................................ 52
- kmp_flag<P> ................................................................. 53
Chapter 5

Module Documentation

5.1 Atomic Operations

5.1.1 Detailed Description

These functions are used for implementing the many different varieties of atomic operations. The compiler is at liberty to inline atomic operations that are naturally supported by the target architecture. For instance on IA-32 architecture an atomic like this can be inlined

```c
static int s = 0;
#pragma omp atomic
s++;
```

using the single instruction: `lock; incl s`

However the runtime does provide entrypoints for these operations to support compilers that choose not to inline them. (For instance, `__kmpc_atomic_fixed4_add` could be used to perform the increment above.)

The names of the functions are encoded by using the data type name and the operation name, as in these tables.

<table>
<thead>
<tr>
<th>Data Type</th>
<th>Data type encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>int8_t</td>
<td>fixed1</td>
</tr>
<tr>
<td>uint8_t</td>
<td>fixed1u</td>
</tr>
<tr>
<td>int16_t</td>
<td>fixed2</td>
</tr>
<tr>
<td>uint16_t</td>
<td>fixed2u</td>
</tr>
<tr>
<td>int32_t</td>
<td>fixed4</td>
</tr>
<tr>
<td>uint32_t</td>
<td>fixed4u</td>
</tr>
<tr>
<td>int32_t</td>
<td>fixed8</td>
</tr>
<tr>
<td>uint32_t</td>
<td>fixed8u</td>
</tr>
<tr>
<td>float</td>
<td>float4</td>
</tr>
<tr>
<td>double</td>
<td>float8</td>
</tr>
<tr>
<td>float10 (8087 eighty bit float)</td>
<td>float10</td>
</tr>
<tr>
<td>complex&lt;float&gt;</td>
<td>cmplx4</td>
</tr>
<tr>
<td>complex&lt;double&gt;</td>
<td>cmplx8</td>
</tr>
<tr>
<td>complex&lt;float10&gt;</td>
<td>cmplx10</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Operation</th>
<th>Operation encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>add</td>
</tr>
<tr>
<td>-</td>
<td>sub</td>
</tr>
<tr>
<td>*</td>
<td>mul</td>
</tr>
<tr>
<td>/</td>
<td>div</td>
</tr>
</tbody>
</table>
&
<<
>>
|
^&
||
maximum
minimum
.equiv.
.neqv.

andb
shl
shr
orb
xor
andl
orl
max
min
eqv
neqv

For non-commutative operations, _rev can also be added for the reversed operation. For the functions that capture the result, the suffix _cpt is added.

Update Functions

The general form of an atomic function that just performs an update (without a capture)

```c
void __kmpc_atomic_<datatype>_<operation>( ident_t *id_ref, int gtid, TYPE * lhs, TYPE rhs );
```

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ident_t</td>
<td>a pointer to source location</td>
</tr>
<tr>
<td>gtid</td>
<td>the global thread id</td>
</tr>
<tr>
<td>lhs</td>
<td>a pointer to the left operand</td>
</tr>
<tr>
<td>rhs</td>
<td>the right operand</td>
</tr>
</tbody>
</table>

**capture functions**

The capture functions perform an atomic update and return a result, which is either the value before the capture, or that after. They take an additional argument to determine which result is returned. Their general form is therefore

```c
TYPE __kmpc_atomic_<datatype>_<operation>_cpt( ident_t *id_ref, int gtid, TYPE * lhs, TYPE rhs, int flag );
```

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ident_t</td>
<td>a pointer to source location</td>
</tr>
<tr>
<td>gtid</td>
<td>the global thread id</td>
</tr>
<tr>
<td>lhs</td>
<td>a pointer to the left operand</td>
</tr>
<tr>
<td>rhs</td>
<td>the right operand</td>
</tr>
<tr>
<td>flag</td>
<td>one if the result is to be captured after the operation, zero if captured before.</td>
</tr>
</tbody>
</table>

The one set of exceptions to this is the complex<float> type where the value is not returned, rather an extra argument pointer is passed.

They look like

```c
void __kmpc_atomic_cmplx4_<op>_cpt( ident_t *id_ref, int gtid, kmp_cmplx32 * lhs, kmp_cmplx32 rhs, kmp_cmplx32 * out, int flag );
```

Read and Write Operations

The OpenMP® standard now supports atomic operations that simply ensure that the value is read or written atomically, with no modification performed. In many cases on IA-32 architecture these operations can be inlined since
the architecture guarantees that no tearing occurs on aligned objects accessed with a single memory operation of up to 64 bits in size.

The general form of the read operations is

```c
TYPE __kmpc_atomic_<type>_rd ( ident_t *id_ref, int gtid, TYPE * loc );
```

For the write operations the form is

```c
void __kmpc_atomic_<type>_wr ( ident_t *id_ref, int gtid, TYPE * lhs, TYPE rhs );
```

Full list of functions

This leads to the generation of 376 atomic functions, as follows.

Functions for integers

There are versions here for integers of size 1,2,4 and 8 bytes both signed and unsigned (where that matters).

```
__kmpc_atomic_fixed1_add
__kmpc_atomic_fixed1_add_cpt
__kmpc_atomic_fixed1_add_fp
__kmpc_atomic_fixed1_andb
__kmpc_atomic_fixed1_andb_cpt
__kmpc_atomic_fixed1_andl
__kmpc_atomic_fixed1_andl_cpt
__kmpc_atomic_fixed1_div
__kmpc_atomic_fixed1_div_cpt
__kmpc_atomic_fixed1_div_cpt_rev
__kmpc_atomic_fixed1_div_float8
__kmpc_atomic_fixed1_div_fp
__kmpc_atomic_fixed1_div_rev
__kmpc_atomic_fixed1_eqv
__kmpc_atomic_fixed1_eqv_cpt
__kmpc_atomic_fixed1_max
__kmpc_atomic_fixed1_max_cpt
__kmpc_atomic_fixed1_min
__kmpc_atomic_fixed1_min_cpt
__kmpc_atomic_fixed1_mul
__kmpc_atomic_fixed1_mul_cpt
__kmpc_atomic_fixed1_mul_float8
__kmpc_atomic_fixed1_mul_fp
__kmpc_atomic_fixed1_neqv
__kmpc_atomic_fixed1_neqv_cpt
__kmpc_atomic_fixed1_orb
__kmpc_atomic_fixed1_orb_cpt
__kmpc_atomic_fixed1_orl
__kmpc_atomic_fixed1_orl_cpt
__kmpc_atomic_fixed1_rd
__kmpc_atomic_fixed1_shl
__kmpc_atomic_fixed1_shl_cpt
__kmpc_atomic_fixed1_shl_cpt_rev
__kmpc_atomic_fixed1_shl_rev
__kmpc_atomic_fixed1_shr
__kmpc_atomic_fixed1_shr_cpt
__kmpc_atomic_fixed1_shr_cpt_rev
__kmpc_atomic_fixed1_shr_rev
__kmpc_atomic_fixed1_sub
__kmpc_atomic_fixed1_sub_cpt
__kmpc_atomic_fixed1_sub_cpt_rev
__kmpc_atomic_fixed1_sub_fp
__kmpc_atomic_fixed1_sub_rev
__kmpc_atomic_fixed1_swap
__kmpc_atomic_fixed1_wr
__kmpc_atomic_fixed1_xor
__kmpc_atomic_fixed1_xor_cpt
__kmpc_atomic_fixed1u_div
__kmpc_atomic_fixed1u_div_cpt
__kmpc_atomic_fixed1u_div_cpt_rev
__kmpc_atomic_fixed1u_div_fp
__kmpc_atomic_fixed1u_div_rev
__kmpc_atomic_fixed1u_shl
__kmpc_atomic_fixed1u_shl_cpt
__kmpc_atomic_fixed1u_shl_cpt_rev
__kmpc_atomic_fixed1u_shl_rev
__kmpc_atomic_fixed1u_shr
__kmpc_atomic_fixed1u_shr_cpt
__kmpc_atomic_fixed1u_shr_cpt_rev
__kmpc_atomic_fixed1u_shr_rev
__kmpc_atomic_fixed2_add
```
5.1 Atomic Operations

- __kmpc_atomic_fixed4_shl_cpt_rev
- __kmpc_atomic_fixed4_shl_rev
- __kmpc_atomic_fixed4_shr
- __kmpc_atomic_fixed4_shr_cpt
- __kmpc_atomic_fixed4_shr_cpt_rev
- __kmpc_atomic_fixed4_shr_rev
- __kmpc_atomic_fixed4_sub
- __kmpc_atomic_fixed4_sub_cpt
- __kmpc_atomic_fixed4_sub_cpt_rev
- __kmpc_atomic_fixed4_sub_rev
- __kmpc_atomic_fixed4_swp
- __kmpc_atomic_fixed4_wr
- __kmpc_atomic_fixed4_xor
- __kmpc_atomic_fixed4_xor_cpt
- __kmpc_atomic_fixed4u_div
- __kmpc_atomic_fixed4u_div_cpt
- __kmpc_atomic_fixed4u_div_cpt_rev
- __kmpc_atomic_fixed4u_div_fp
- __kmpc_atomic_fixed4u_div_rev
- __kmpc_atomic_fixed4u_shr
- __kmpc_atomic_fixed4u_shr_cpt
- __kmpc_atomic_fixed4u_shr_cpt_rev
- __kmpc_atomic_fixed4u_shr_rev
- __kmpc_atomic_fixed8_add
- __kmpc_atomic_fixed8_add_cpt
- __kmpc_atomic_fixed8_add_fp
- __kmpc_atomic_fixed8_andb
- __kmpc_atomic_fixed8_andl
- __kmpc_atomic_fixed8_andl_cpt
- __kmpc_atomic_fixed8_div
- __kmpc_atomic_fixed8_div_cpt
- __kmpc_atomic_fixed8_div_cpt_rev
- __kmpc_atomic_fixed8_div_float8
- __kmpc_atomic_fixed8_div_fp
- __kmpc_atomic_fixed8_div_rev
- __kmpc_atomic_fixed8_eqv
- __kmpc_atomic_fixed8_eqv_cpt
- __kmpc_atomic_fixed8_max
- __kmpc_atomic_fixed8_max_cpt
- __kmpc_atomic_fixed8_min
- __kmpc_atomic_fixed8_min_cpt
- __kmpc_atomic_fixed8_mul
- __kmpc_atomic_fixed8_mul_cpt
- __kmpc_atomic_fixed8_mul_float8
- __kmpc_atomic_fixed8_mul_rev
- __kmpc_atomic_fixed8_neqv
- __kmpc_atomic_fixed8_neqv_cpt
- __kmpc_atomic_fixed8_orb
- __kmpc_atomic_fixed8_orb_cpt
- __kmpc_atomic_fixed8_orl
- __kmpc_atomic_fixed8_orl_cpt
- __kmpc_atomic_fixed8_rd
- __kmpc_atomic_fixed8_shl
- __kmpc_atomic_fixed8_shl_cpt
- __kmpc_atomic_fixed8_shl_cpt_rev
- __kmpc_atomic_fixed8_shl_rev
- __kmpc_atomic_fixed8_shr
- __kmpc_atomic_fixed8_shr_cpt
- __kmpc_atomic_fixed8_shr_cpt_rev
- __kmpc_atomic_fixed8_shr_rev
- __kmpc_atomic_fixed8_sub
- __kmpc_atomic_fixed8_sub_cpt
- __kmpc_atomic_fixed8_sub_cpt_rev
- __kmpc_atomic_fixed8_sub_rev
- __kmpc_atomic_fixed8_swp
- __kmpc_atomic_fixed8_wr
- __kmpc_atomic_fixed8_xor
- __kmpc_atomic_fixed8_xor_cpt
- __kmpc_atomic_fixed8u_div
- __kmpc_atomic_fixed8u_div_cpt
- __kmpc_atomic_fixed8u_div_cpt_rev
- __kmpc_atomic_fixed8u_div_fp
- __kmpc_atomic_fixed8u_div_rev
- __kmpc_atomic_fixed8u_shr
- __kmpc_atomic_fixed8u_shr_cpt
- __kmpc_atomic_fixed8u_shr_cpt_rev
- __kmpc_atomic_fixed8u_shr_rev
Functions for floating point

There are versions here for floating point numbers of size 4, 8, 10 and 16 bytes. (Ten byte floats are used by X87, but are now rare).

__kmpc_atomic_float4_add
__kmpc_atomic_float4_add_cpt
__kmpc_atomic_float4_add_float8
__kmpc_atomic_float4_add_fp
__kmpc_atomic_float4_div
__kmpc_atomic_float4_div_cpt
__kmpc_atomic_float4_div_cpt_rev
__kmpc_atomic_float4_div_float8
__kmpc_atomic_float4_div_fp
__kmpc_atomic_float4_div_rev
__kmpc_atomic_float4_max
__kmpc_atomic_float4_max_cpt
__kmpc_atomic_float4_min
__kmpc_atomic_float4_min_cpt
__kmpc_atomic_float4_mul
__kmpc_atomic_float4_mul_cpt
__kmpc_atomic_float4_mul_float8
__kmpc_atomic_float4_mul_fp
__kmpc_atomic_float4_rd
__kmpc_atomic_float4_sub
__kmpc_atomic_float4_sub_cpt
__kmpc_atomic_float4_sub_cpt_rev
__kmpc_atomic_float4_sub_float8
__kmpc_atomic_float4_sub_fp
__kmpc_atomic_float4_sub_rev
__kmpc_atomic_float4_swp
__kmpc_atomic_float4_wr
__kmpc_atomic_float8_add
__kmpc_atomic_float8_add_cpt
__kmpc_atomic_float8_add_fp
__kmpc_atomic_float8_div
__kmpc_atomic_float8_div_cpt
__kmpc_atomic_float8_div_cpt_rev
__kmpc_atomic_float8_div_float8
__kmpc_atomic_float8_div_fp
__kmpc_atomic_float8_max
__kmpc_atomic_float8_max_cpt
__kmpc_atomic_float8_min
__kmpc_atomic_float8_min_cpt
__kmpc_atomic_float8_mul
__kmpc_atomic_float8_mul_cpt
__kmpc_atomic_float8_mul_fp
__kmpc_atomic_float8_rd
__kmpc_atomic_float8_sub
__kmpc_atomic_float8_sub_cpt
__kmpc_atomic_float8_sub_cpt_rev
__kmpc_atomic_float8_sub_float8
__kmpc_atomic_float8_sub_rev
__kmpc_atomic_float8_swp
__kmpc_atomic_float8_wr
__kmpc_atomic_float10_add
__kmpc_atomic_float10_add_cpt
__kmpc_atomic_float10_add_fp
__kmpc_atomic_float10_div
__kmpc_atomic_float10_div_cpt
__kmpc_atomic_float10_div_cpt_rev
__kmpc_atomic_float10_div_float8
__kmpc_atomic_float10_div_fp
__kmpc_atomic_float10_max
__kmpc_atomic_float10_max_cpt
__kmpc_atomic_float10_mul
__kmpc_atomic_float10_mul_cpt
__kmpc_atomic_float10_mul_fp
__kmpc_atomic_float10_rd
__kmpc_atomic_float10_sub
__kmpc_atomic_float10_sub_cpt
__kmpc_atomic_float10_sub_cpt_rev
__kmpc_atomic_float10_sub_float8
__kmpc_atomic_float10_sub_rev
__kmpc_atomic_float10_swp
__kmpc_atomic_float10_wr
__kmpc_atomic_float16_add
__kmpc_atomic_float16_add_cpt
__kmpc_atomic_float16_div
__kmpc_atomic_float16_div_cpt
__kmpc_atomic_float16_div_cpt_rev
__kmpc_atomic_float16_div_float8
__kmpc_atomic_float16_div_fp
__kmpc_atomic_float16_max
__kmpc_atomic_float16_max_cpt
__kmpc_atomic_float16_min
__kmpc_atomic_float16_min_cpt
__kmpc_atomic_float16_mul
5.1 Atomic Operations

__kmpc_atomic_float16_mul_cpt
__kmpc_atomic_float16_rd
__kmpc_atomic_float16_sub
__kmpc_atomic_float16_sub_cpt
__kmpc_atomic_float16_sub_cpt_rev
__kmpc_atomic_float16_sub_rev
__kmpc_atomic_float16_swp
__kmpc_atomic_float16_wr

Functions for Complex types

Functions for complex types whose component floating point variables are of size 4, 8, 10 or 16 bytes. The names here are based on the size of the component float, not the size of the complex type. So __kmpc_atomic_cmplx8_add is an operation on a complex<double> or complex(kind=8), not complex<float>.

__kmpc_atomic_cmplx4_add
__kmpc_atomic_cmplx4_add_cpt
__kmpc_atomic_cmplx4_div
__kmpc_atomic_cmplx4_div_cpt
__kmpc_atomic_cmplx4_div_cpt_rev
__kmpc_atomic_cmplx4_div_rev
__kmpc_atomic_cmplx4_mul
__kmpc_atomic_cmplx4_mul_cpt
__kmpc_atomic_cmplx4_rd
__kmpc_atomic_cmplx4_sub
__kmpc_atomic_cmplx4_sub_cpt
__kmpc_atomic_cmplx4_sub_cpt_rev
__kmpc_atomic_cmplx4_sub_rev
__kmpc_atomic_cmplx4_swp
__kmpc_atomic_cmplx4_wr

__kmpc_atomic_cmplx8_add
__kmpc_atomic_cmplx8_add_cpt
__kmpc_atomic_cmplx8_div
__kmpc_atomic_cmplx8_div_cpt
__kmpc_atomic_cmplx8_div_cpt_rev
__kmpc_atomic_cmplx8_div_rev
__kmpc_atomic_cmplx8_mul
__kmpc_atomic_cmplx8_mul_cpt
__kmpc_atomic_cmplx8_rd
__kmpc_atomic_cmplx8_sub
__kmpc_atomic_cmplx8_sub_cpt
__kmpc_atomic_cmplx8_sub_cpt_rev
__kmpc_atomic_cmplx8_sub_cpt_rev
__kmpc_atomic_cmplx8_swp
__kmpc_atomic_cmplx8_wr

__kmpc_atomic_cmplx10_add
__kmpc_atomic_cmplx10_add_cpt
__kmpc_atomic_cmplx10_div
__kmpc_atomic_cmplx10_div_cpt
__kmpc_atomic_cmplx10_div_cpt_rev
__kmpc_atomic_cmplx10_div_rev
__kmpc_atomic_cmplx10_mul
__kmpc_atomic_cmplx10_mul_cpt
__kmpc_atomic_cmplx10_rd
__kmpc_atomic_cmplx10_sub
__kmpc_atomic_cmplx10_sub_cpt
__kmpc_atomic_cmplx10_sub_cpt_rev
__kmpc_atomic_cmplx10_sub_cpt_rev
__kmpc_atomic_cmplx10_swp
__kmpc_atomic_cmplx10_wr

__kmpc_atomic_cmplx16_add
__kmpc_atomic_cmplx16_add_cpt
__kmpc_atomic_cmplx16_div
__kmpc_atomic_cmplx16_div_cpt
__kmpc_atomic_cmplx16_div_cpt_rev
__kmpc_atomic_cmplx16_div_rev
__kmpc_atomic_cmplx16_mul
__kmpc_atomic_cmplx16_mul_cpt
__kmpc_atomic_cmplx16_sub
__kmpc_atomic_cmplx16_sub_cpt
__kmpc_atomic_cmplx16_sub_cpt_rev
__kmpc_atomic_cmplx16_sub_cpt_rev
__kmpc_atomic_cmplx16_swp
__kmpc_atomic_cmplx16_wr
5.2 Wait/Release operations

- enum flag_type { flag32, flag64, flag_oncore }

5.2.1 Detailed Description

The definitions and functions here implement the lowest level thread synchronizations of suspending a thread and awakening it. They are used to build higher level operations such as barriers and fork/join.

5.2.2 Enumeration Type Documentation

5.2.2.1 enum flag_type

The flag_type describes the storage used for the flag.

Enumerator

- **flag32** 32 bit flags
- **flag64** 64 bit flags
- **flag_oncore** special 64-bit flag for on-core barrier (hierarchical)

Definition at line 58 of file kmp_wait_release.h.
5.3 Basic Types

• typedef struct ident ident_t
• #define KMP_IDENT_IMB 0x01
• #define KMP_IDENT_KMPC 0x02
• #define KMP_IDENT_AUTOPAR 0x08
• #define KMP_IDENT_ATOMIC_REDUCE 0x10
• #define KMP_IDENT_BARRIER_EXPL 0x20
• #define KMP_IDENT_BARRIER_IMPL 0x0040

5.3.1 Detailed Description
Types that are used throughout the runtime.

5.3.2 Macro Definition Documentation

5.3.2.1 #define KMP_IDENT_ATOMIC_REDUCE 0x10
Compiler generates atomic reduction option for kmpc_reduce*
Definition at line 196 of file kmp.h.

5.3.2.2 #define KMP_IDENT_AUTOPAR 0x08
Entry point generated by auto-parallelization
Definition at line 194 of file kmp.h.
Referenced by __kmpc_end_serialized_parallel().

5.3.2.3 #define KMP_IDENT_BARRIER_EXPL 0x20
To mark a ‘barrier’ directive in user code
Definition at line 198 of file kmp.h.

5.3.2.4 #define KMP_IDENT_BARRIER_IMPL 0x0040
To Mark implicit barriers.
Definition at line 200 of file kmp.h.

5.3.2.5 #define KMP_IDENT_JMB 0x01
Values for bit flags used in the ident_t to describe the fields.
Use trampoline for internal microtasks
Definition at line 189 of file kmp.h.

5.3.2.6 #define KMP_IDENT_KMPC 0x02
Use c-style ident structure
Definition at line 191 of file kmp.h.
5.3.3  Typedef Documentation

5.3.3.1 typedef struct ident ident_t

The ident structure that describes a source location.
5.4 Deprecated Functions

Functions

- `kmp_int32 __kmpc_ok_to_fork (ident_t *loc)`

5.4.1 Detailed Description

Functions in this group are for backwards compatibility only, and should not be used in new code.

5.4.2 Function Documentation

5.4.2.1 `kmp_int32 __kmpc_ok_to_fork (ident_t * loc )`

Parameters

| loc     | location description |

This function need not be called. It always returns TRUE.

Definition at line 181 of file kmp_csupport.c.
5.5  Startup and Shutdown

Functions

- void __kmpc_begin (ident_t *loc, kmp_int32 flags)
- void __kmpc_end (ident_t *loc)

5.5.1  Detailed Description

These functions are for library initialization and shutdown.

5.5.2  Function Documentation

5.5.2.1  void __kmpc_begin ( ident_t * loc, kmp_int32 flags )

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>loc</td>
<td>in source location information</td>
</tr>
<tr>
<td>flags</td>
<td>in for future use (currently ignored)</td>
</tr>
</tbody>
</table>

Initialize the runtime library. This call is optional; if it is not made then it will be implicitly called by attempts to use other library functions.

Definition at line 66 of file kmp_csupport.c.

5.5.2.2  void __kmpc_end ( ident_t * loc )

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>loc</td>
<td>source location information</td>
</tr>
</tbody>
</table>

Shutdown the runtime library. This is also optional, and even if called will not do anything unless the KMP_IGNORE_MPPEND environment variable is set to zero.

Definition at line 84 of file kmp_csupport.c.
5.6 Parallel (fork/join)

Typedefs

- typedef void(*kmpc_micro)(kmp_int32 *global_tid, kmp_int32 *bound_tid,...)

Functions

- void __kmpc_push_num_threads(ident_t *loc, kmp_int32 global_tid, kmp_int32 num_threads)
- void __kmpc_fork_call(ident_t *loc, kmp_int32 argc, kmpc_micro microtask,...)
- void __kmpc_push_num_teams(ident_t *loc, kmp_int32 global_tid, kmp_int32 num_teams, kmp_int32 num_threads)
- void __kmpc_fork_teams(ident_t *loc, kmp_int32 argc, kmpc_micro microtask,...)
- void __kmpc_serialized_parallel(ident_t *loc, kmp_int32 global_tid)
- void __kmpc_end_serialized_parallel(ident_t *loc, kmp_int32 global_tid)

5.6.1 Detailed Description

These functions are used for implementing #pragma omp parallel.

5.6.2 Typedef Documentation

5.6.2.1 typedef void(*kmpc_micro)(kmp_int32 *global_tid, kmp_int32 *bound_tid,...)

The type for a microtask which gets passed to __kmpc_fork_call(). The arguments to the outlined function are

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>global_tid</td>
<td>the global thread identity of the thread executing the function.</td>
</tr>
<tr>
<td>bound_tid</td>
<td>the local identity of the thread executing the function</td>
</tr>
<tr>
<td>...</td>
<td>pointers to shared variables accessed by the function.</td>
</tr>
</tbody>
</table>

Definition at line 1456 of file kmp.h.

5.6.3 Function Documentation

5.6.3.1 void __kmpc_end_serialized_parallel (ident_t *loc, kmp_int32 global_tid)

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>loc</td>
<td>source location information</td>
</tr>
<tr>
<td>global_tid</td>
<td>global thread number</td>
</tr>
</tbody>
</table>

Leave a serialized parallel construct.
Definition at line 504 of file kmp_csupport.c.

5.6.3.2 void __kmpc_fork_call (ident_t *loc, kmp_int32 argc, kmpc_micro microtask, ...)

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>loc</td>
<td>source location information</td>
</tr>
<tr>
<td>argc</td>
<td>total number of arguments in the ellipsis</td>
</tr>
<tr>
<td>microtask</td>
<td>pointer to callback routine consisting of outlined parallel construct</td>
</tr>
<tr>
<td>...</td>
<td>pointers to shared variables that aren’t global</td>
</tr>
</tbody>
</table>

Generated on Mon Aug 8 2016 17:18:34 for Intel® OpenMP® Runtime Library by Doxygen
Do the actual fork and call the microtask in the relevant number of threads.
Definition at line 301 of file kmp_csupport.c.

5.6.3.3

void _kmpc_fork_teams ( ident_t *loc, kmp_int32 argc, kmpc_micro microtask, ... )

Parameters

<table>
<thead>
<tr>
<th>loc</th>
<th>source location information</th>
</tr>
</thead>
<tbody>
<tr>
<td>argc</td>
<td>total number of arguments in the ellipsis</td>
</tr>
<tr>
<td>microtask</td>
<td>pointer to callback routine consisting of outlined teams construct</td>
</tr>
<tr>
<td>...</td>
<td>pointers to shared variables that aren’t global</td>
</tr>
</tbody>
</table>

Do the actual fork and call the microtask in the relevant number of threads.
Definition at line 400 of file kmp_csupport.c.

5.6.3.4

void _kmpc_push_num_teams ( ident_t *loc, kmp_int32 global_tid, kmp_int32 num_teams, kmp_int32 num_threads )

Parameters

<table>
<thead>
<tr>
<th>loc</th>
<th>source location information</th>
</tr>
</thead>
<tbody>
<tr>
<td>global_tid</td>
<td>global thread number</td>
</tr>
<tr>
<td>num_teams</td>
<td>number of teams requested for the teams construct</td>
</tr>
<tr>
<td>num_threads</td>
<td>number of threads per team requested for the teams construct</td>
</tr>
</tbody>
</table>

Set the number of teams to be used by the teams construct. This call is only required if the teams construct has a num_teams clause or a thread_limit clause (or both).
Definition at line 382 of file kmp_csupport.c.

5.6.3.5

void _kmpc_push_num_threads ( ident_t *loc, kmp_int32 global_tid, kmp_int32 num_threads )

Parameters

<table>
<thead>
<tr>
<th>loc</th>
<th>source location information</th>
</tr>
</thead>
<tbody>
<tr>
<td>global_tid</td>
<td>global thread number</td>
</tr>
<tr>
<td>num_threads</td>
<td>number of threads requested for this parallel construct</td>
</tr>
</tbody>
</table>

Set the number of threads to be used by the next fork spawned by this thread. This call is only required if the parallel construct has a num_threads clause.
Definition at line 260 of file kmp_csupport.c.

5.6.3.6

void _kmpc_serialized_parallel ( ident_t *loc, kmp_int32 global_tid )

Parameters

<table>
<thead>
<tr>
<th>loc</th>
<th>source location information</th>
</tr>
</thead>
<tbody>
<tr>
<td>global_tid</td>
<td>global thread number</td>
</tr>
</tbody>
</table>

Enter a serialized parallel construct. This interface is used to handle a conditional parallel region, like this,

#pragma omp parallel if (condition)

when the condition is false.
Definition at line 489 of file kmp_csupport.c.
5.7 Thread Information

Functions

- `kmp_int32 __kmpc_global_thread_num (ident_t *loc)`
- `kmp_int32 __kmpc_global_num_threads (ident_t *loc)`
- `kmp_int32 __kmpc_bound_thread_num (ident_t *loc)`
- `kmp_int32 __kmpc_bound_num_threads (ident_t *loc)`
- `kmp_int32 __kmpc_in_parallel (ident_t *loc)`

5.7.1 Detailed Description

These functions return information about the currently executing thread.

5.7.2 Function Documentation

5.7.2.1 `kmp_int32 __kmpc_bound_num_threads (ident_t *loc)`

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>loc</td>
<td>Source location information.</td>
</tr>
</tbody>
</table>

Returns

The number of threads in the innermost active parallel construct.

Definition at line 167 of file kmp_csupport.c.

5.7.2.2 `kmp_int32 __kmpc_bound_thread_num (ident_t *loc)`

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>loc</td>
<td>Source location information.</td>
</tr>
</tbody>
</table>

Returns

The thread number of the calling thread in the innermost active parallel construct.

Definition at line 155 of file kmp_csupport.c.

5.7.2.3 `kmp_int32 __kmpc_global_num_threads (ident_t *loc)`

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>loc</td>
<td>Source location information.</td>
</tr>
</tbody>
</table>

Returns

The number of threads under control of the OpenMP® runtime.

This function can be called in any context. It returns the total number of threads under the control of the OpenMP runtime. That is not a number that can be determined by any OpenMP standard calls, since the library may be called from more than one non-OpenMP thread, and this reflects the total over all such calls. Similarly the runtime maintains underlying threads even when they are not active (since the cost of creating and destroying OS threads is high), this call counts all such threads even if they are not waiting for work.
Definition at line 141 of file kmp_csupport.c.

5.7.2.4 kmp_int32 __kmpc_global_thread_num (ident_t * loc)

Parameters

| loc   | Source location information. |

Returns

The global thread index of the active thread.

This function can be called in any context.

If the runtime has only been entered at the outermost level from a single (necessarily non-OpenMP*) thread, then the thread number is that which would be returned by omp_get_thread_num() in the outermost active parallel construct. (Or zero if there is no active parallel construct, since the master thread is necessarily thread zero).

If multiple non-OpenMP threads all enter an OpenMP construct then this will be a unique thread identifier among all the threads created by the OpenMP runtime (but the value cannot be defined in terms of OpenMP thread ids returned by omp_get_thread_num()).

Definition at line 118 of file kmp_csupport.c.

5.7.2.5 kmp_int32 __kmpc_in_parallel (ident_t * loc)

Parameters

| loc   | Source location information. |

Returns

1 if this thread is executing inside an active parallel region, zero if not.

Definition at line 245 of file kmp_csupport.c.

Referenced by __kmpc_fork_call().
5.8 Work Sharing

Enumerations

```c
enum sched_type {
    kmp_sch_lower = 32 , kmp_sch_static = 34 , kmp_sch_guided_chunked = 36 , kmp_sch_auto = 38 ,
    kmp_sch_static_steal = 44, kmp_sch_upper = 46, kmp_ord_lower = 64 , kmp_ord_static = 66 ,
    kmp_ord_auto = 70 , kmp_ord_upper = 72, kmp_distribute_static_chunked = 91, kmp_distribute_static = 92,
    kmp_nm_lower = 160 , kmp_nm_static = 162 , kmp_nm_guided_chunked = 164 , kmp_nm_auto = 166 ,
    kmp_nm_ord_static = 194 , kmp_nm_ord_auto = 198 , kmp_nm_upper = 200, kmp_sch_default = kmp_sch__static
};
```

Functions

```c
• kmp_int32 __kmpc_master (ident_t loc, kmp_int32 global_tid)
• void __kmpc_end_master (ident_t loc, kmp_int32 global_tid)
• void __kmpc_ordered (ident_t loc, kmp_int32 gtid)
• void __kmpc_end_ordered (ident_t loc, kmp_int32 gtid)
• void __kmpc_critical (ident_t loc, kmp_int32 global_tid, kmp_critical_name *crit)
• void __kmpc_end_critical (ident_t loc, kmp_int32 global_tid, kmp_critical_name *crit)
• kmp_int32 __kmpc_single (ident_t loc, kmp_int32 global_tid)
• void __kmpc_end_single (ident_t loc, kmp_int32 global_tid)
• void __kmpc_for_static_fini (ident_t loc, kmp_int32 global_tid)
• void __kmpc_dispatch_init_4 (ident_t loc, kmp_int32 gtid, enum sched_type schedule, kmp_int32 lb, kmp_int32 ub, kmp_int32 st, kmp_int32 chunk)
• void __kmpc_dispatch_init_4u (ident_t loc, kmp_int32 gtid, enum sched_type schedule, kmp_uint32 lb, kmp_uint32 ub, kmp_int32 st, kmp_int32 chunk)
• void __kmpc_dispatch_init_8 (ident_t loc, kmp_int32 gtid, enum sched_type schedule, kmp_int64 lb, kmp_int64 ub, kmp_int64 st, kmp_int64 chunk)
• void __kmpc_dispatch_init_8u (ident_t loc, kmp_int32 gtid, enum sched_type schedule, kmp_uint64 lb, kmp_uint64 ub, kmp_int64 st, kmp_int64 chunk)
• void __kmpc_for_static_fini (ident_t loc, kmp_int32 global_tid)
• void __kmpc_for_static_init_4 (ident_t loc, kmp_int32 gtid, kmp_int32 schedtype, kmp_int32 plastiter, kmp_int32 plower, kmp_int32 pupper, kmp_int32 pstride, kmp_int32 incr, kmp_int32 chunk)
• void __kmpc_dispatch_fini_4 (ident_t loc, kmp_int32 gtid)
• void __kmpc_dispatch_fini_8 (ident_t loc, kmp_int32 gtid)
• void __kmpc_dispatch_fini_4u (ident_t loc, kmp_int32 gtid)
• void __kmpc_dispatch_fini_8u (ident_t loc, kmp_int32 gtid)
```
5.8.1 Detailed Description

These functions are used for implementing #pragma omp for, #pragma omp sections, #pragma omp single and #pragma omp master constructs.

When handling loops, there are different functions for each of the signed and unsigned 32 and 64 bit integer types which have the name suffixes _4, _4u, _8 and _8u. The semantics of each of the functions is the same, so they are only described once.

Static loop scheduling is handled by __kmpc_for_static_init_4 and friends. Only a single call is needed, since the iterations to be executed by any give thread can be determined as soon as the loop parameters are known.

Dynamic scheduling is handled by the __kmpc_dispatch_init_4 and __kmpc_dispatch_next_4 functions. The init function is called once in each thread outside the loop, while the next function is called each time that the previous chunk of work has been exhausted.

5.8.2 Enumeration Type Documentation

5.8.2.1 enum sched_type

Describes the loop schedule to be used for a parallel for loop.

Enumerator

- kmp_sch_lower lower bound for unordered values
- kmp_sch_static static unspecialized
- kmp_sch_guided_chunked guided unspecialized
- kmp_sch_auto auto
- kmp_sch_static_steal accessible only through KMP_SCHEDULE environment variable
- kmp_sch_upper upper bound for unordered values
- kmp_ord_lower lower bound for ordered values, must be power of 2
- kmp_ord_static ordered static unspecialized
- kmp_ord_auto ordered auto
### Function Documentation

**5.8.3.1 void \_kmpe\_critical ( ident\_t * loc, kmp\_int32 global\_tid, kmp\_critical\_name * crit )**

**Parameters**

<table>
<thead>
<tr>
<th>Loc</th>
<th>Source location information.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Global_tid</td>
<td>Global thread number.</td>
</tr>
<tr>
<td>Crit</td>
<td>Identity of the critical section. This could be a pointer to a lock associated with the critical section, or some other suitably unique value.</td>
</tr>
</tbody>
</table>

Enter code protected by a critical construct. This function blocks until the executing thread can enter the critical section.

Definition at line 1099 of file kmp\_csupport\_c.c.

**5.8.3.2 void \_kmpe\_dispatch\_fini\_4 ( ident\_t * loc, kmp\_int32 gtid )**

**Parameters**

<table>
<thead>
<tr>
<th>Loc</th>
<th>Source code location</th>
</tr>
</thead>
<tbody>
<tr>
<td>Gtid</td>
<td>Global thread id</td>
</tr>
</tbody>
</table>

Mark the end of a dynamic loop.

Definition at line 2604 of file kmp\_dispatch\_cpp.

**5.8.3.3 void \_kmpe\_dispatch\_fini\_4u ( ident\_t * loc, kmp\_int32 gtid )**

See \_kmpe\_dispatch\_fini\_4

Definition at line 2622 of file kmp\_dispatch\_cpp.

**5.8.3.4 void \_kmpe\_dispatch\_fini\_8 ( ident\_t * loc, kmp\_int32 gtid )**

See \_kmpe\_dispatch\_fini\_4

Definition at line 2613 of file kmp\_dispatch\_cpp.
5.8.3.5 void _kmpc_dispatch_fini_8u (ident_t * loc, kmp_int32 gtid)

See __kmpc_dispatch_fini_4

Definition at line 2631 of file kmp_dispatch.cpp.

5.8.3.6 void _kmpc_dispatch_init_4 (ident_t * loc, kmp_int32 gtid, enum sched_type schedule, kmp_int32 lb, kmp_int32 ub, kmp_int32 st, kmp_int32 chunk)

Parameters

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>loc</td>
<td>Source location</td>
</tr>
<tr>
<td>gtid</td>
<td>Global thread id</td>
</tr>
<tr>
<td>schedule</td>
<td>Schedule type</td>
</tr>
<tr>
<td>lb</td>
<td>Lower bound</td>
</tr>
<tr>
<td>ub</td>
<td>Upper bound</td>
</tr>
<tr>
<td>st</td>
<td>Step (or increment if you prefer)</td>
</tr>
<tr>
<td>chunk</td>
<td>The chunk size to block with</td>
</tr>
</tbody>
</table>

This function prepares the runtime to start a dynamically scheduled for loop, saving the loop arguments. These functions are all identical apart from the types of the arguments.

Definition at line 2462 of file kmp_dispatch.cpp.

5.8.3.7 void _kmpc_dispatch_init_4u (ident_t * loc, kmp_int32 gtid, enum sched_type schedule, kmp_uint32 lb, kmp_uint32 ub, kmp_int32 st, kmp_int32 chunk)

See __kmpc_dispatch_init_4

Definition at line 2472 of file kmp_dispatch.cpp.

5.8.3.8 void _kmpc_dispatch_init_8 (ident_t * loc, kmp_int32 gtid, enum sched_type schedule, kmp_int64 lb, kmp_int64 ub, kmp_int64 st, kmp_int64 chunk)

See __kmpc_dispatch_init_4

Definition at line 2483 of file kmp_dispatch.cpp.

5.8.3.9 void _kmpc_dispatch_init_8u (ident_t * loc, kmp_int32 gtid, enum sched_type schedule, kmp_uint64 lb, kmp_uint64 ub, kmp_int64 st, kmp_int64 chunk)

See __kmpc_dispatch_init_4

Definition at line 2495 of file kmp_dispatch.cpp.

5.8.3.10 int _kmpc_dispatch_next_4 (ident_t * loc, kmp_int32 gtid, kmp_int32 * p_last, kmp_int32 * p_lb, kmp_int32 * p_ub, kmp_int32 * p_st)

Parameters

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>loc</td>
<td>Source code location</td>
</tr>
<tr>
<td>gtid</td>
<td>Global thread id</td>
</tr>
<tr>
<td>p_last</td>
<td>Pointer to a flag set to one if this is the last chunk or zero otherwise</td>
</tr>
<tr>
<td>p_lb</td>
<td>Pointer to the lower bound for the next chunk of work</td>
</tr>
<tr>
<td>p_ub</td>
<td>Pointer to the upper bound for the next chunk of work</td>
</tr>
<tr>
<td>p_st</td>
<td>Pointer to the stride for the next chunk of work</td>
</tr>
</tbody>
</table>
5.8 Work Sharing

Returns

one if there is work to be done, zero otherwise

Get the next dynamically allocated chunk of work for this thread. If there is no more work, then the lb, ub and stride need not be modified.

Definition at line 2561 of file kmp_dispatch.cpp.

5.8.3.11 int _kmpec_dispatch_next_4u (ident_t *loc, kmp_int32 gtid, kmp_int32 *p_last, kmp_uint32 *p_lb, kmp_uint32 *p_ub, kmp_int32 *p_st )

See __kmpec_dispatch_next_4

Definition at line 2571 of file kmp_dispatch.cpp.

5.8.3.12 int _kmpec_dispatch_next_8 (ident_t *loc, kmp_int32 gtid, kmp_int32 *p_last, kmp_int64 *p_lb, kmp_int64 *p_ub, kmp_int64 *p_st )

See __kmpec_dispatch_next_4

Definition at line 2581 of file kmp_dispatch.cpp.

5.8.3.13 int _kmpec_dispatch_next_8u (ident_t *loc, kmp_int32 gtid, kmp_int32 *p_last, kmp_uint64 *p_lb, kmp_uint64 *p_ub, kmp_int64 *p_st )

See __kmpec_dispatch_next_4

Definition at line 2591 of file kmp_dispatch.cpp.

5.8.3.14 void _kmpec_dist_dispatch_init_4 (ident_t *loc, kmp_int32 gtid, enum sched_type schedule, kmp_int32 *p_last, kmp_int32 lb, kmp_int32 ub, kmp_int32 st, kmp_int32 chunk )

See __kmpec_dispatch_init_4

Difference from __kmpec_dispatch_init set of functions is these functions are called for composite distribute parallel for construct. Thus before regular iterations dispatching we need to calc per-team iteration space.

These functions are all identical apart from the types of the arguments.

Definition at line 2513 of file kmp_dispatch.cpp.

5.8.3.15 void _kmpec_dist_for_static_init_4 (ident_t *loc, kmp_int32 gtid, kmp_int32 schedule, kmp_int32 plastiter, kmp_int32 plower, kmp_int32 *pupper, kmp_int32 *pupperD, kmp_int32 *pstride, kmp_int32 incr, kmp_int32 chunk )

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>loc</td>
<td>Source code location</td>
</tr>
<tr>
<td>gtid</td>
<td>Global thread id of this thread</td>
</tr>
<tr>
<td>schedule</td>
<td>Scheduling type for the parallel loop</td>
</tr>
<tr>
<td>plastiter</td>
<td>Pointer to the &quot;last iteration&quot; flag</td>
</tr>
<tr>
<td>plower</td>
<td>Pointer to the lower bound</td>
</tr>
<tr>
<td>pupper</td>
<td>Pointer to the upper bound of loop chunk</td>
</tr>
<tr>
<td>pupperD</td>
<td>Pointer to the upper bound of dist_chunk</td>
</tr>
<tr>
<td>pstride</td>
<td>Pointer to the stride for parallel loop</td>
</tr>
<tr>
<td>incr</td>
<td>Loop increment</td>
</tr>
<tr>
<td>chunk</td>
<td>The chunk size for the parallel loop</td>
</tr>
</tbody>
</table>
Each of the four functions here are identical apart from the argument types.

The functions compute the upper and lower bounds and strides to be used for the set of iterations to be executed by the current thread from the statically scheduled loop that is described by the initial values of the bounds, strides, increment and chunks for parallel loop and distribute constructs.

Definition at line 868 of file kmp_sched.cpp.

5.8.3.16 void __kmpc_dist_for_static_init_4u ( ident_t * loc, kmp_int32 gtid, kmp_int32 schedule, kmp_int32 * plastiter, kmp_uint32 * plower, kmp_uint32 * pupper, kmp_int32 * pupperD, kmp_int32 * pstride, kmp_int32 incr, kmp_int32 chunk )

See __kmpc_dist_for_static_init_4
Definition at line 881 of file kmp_sched.cpp.

5.8.3.17 void __kmpc_dist_for_static_init_8 ( ident_t * loc, kmp_int32 gtid, kmp_int32 schedule, kmp_int32 * plastiter, kmp_int64 * plower, kmp_int64 * pupper, kmp_int64 * pupperD, kmp_int64 * pstride, kmp_int64 incr, kmp_int64 chunk )

See __kmpc_dist_for_static_init_4
Definition at line 894 of file kmp_sched.cpp.

5.8.3.18 void __kmpc_dist_for_static_init_8u ( ident_t * loc, kmp_int32 gtid, kmp_int32 schedule, kmp_int32 * plastiter, kmp_uint64 * plower, kmp_uint64 * pupper, kmp_uint64 * pupperD, kmp_int64 * pstride, kmp_int64 incr, kmp_int64 chunk )

See __kmpc_dist_for_static_init_4
Definition at line 907 of file kmp_sched.cpp.

5.8.3.19 void __kmpc_end_critical ( ident_t * loc, kmp_int32 global_tid, kmp_critical_name * crit )

Parameters

<table>
<thead>
<tr>
<th>loc</th>
<th>source location information.</th>
</tr>
</thead>
<tbody>
<tr>
<td>global_tid</td>
<td>global thread number.</td>
</tr>
<tr>
<td>crit</td>
<td>identity of the critical section. This could be a pointer to a lock associated with the critical section, or some other suitably unique value.</td>
</tr>
</tbody>
</table>

Leave a critical section, releasing any lock that was held during its execution.
Definition at line 1283 of file kmp_csupport.c.

5.8.3.20 void __kmpc_end_master ( ident_t * loc, kmp_int32 global_tid )

Parameters

<table>
<thead>
<tr>
<th>loc</th>
<th>source location information.</th>
</tr>
</thead>
<tbody>
<tr>
<td>globaltid</td>
<td>global thread number.</td>
</tr>
</tbody>
</table>

Mark the end of a master region. This should only be called by the thread that executes the master region.
Definition at line 775 of file kmp_csupport.c.

5.8.3.21 void __kmpc_end_ordered ( ident_t * loc, kmp_int32 gtid )
5.8 Work Sharing

Parameters

<table>
<thead>
<tr>
<th>loc</th>
<th>source location information.</th>
</tr>
</thead>
<tbody>
<tr>
<td>gtid</td>
<td>global thread number.</td>
</tr>
</tbody>
</table>

End execution of an ordered construct.
Definition at line 875 of file kmp_csupport.c.

5.8.3.22 void _kmpc_end_single ( ident_t * loc, kmp_int32 global_tid )

Parameters

<table>
<thead>
<tr>
<th>loc</th>
<th>source location information</th>
</tr>
</thead>
<tbody>
<tr>
<td>global_tid</td>
<td>global thread number</td>
</tr>
</tbody>
</table>

Mark the end of a single construct. This function should only be called by the thread that executed the block of code protected by the single construct.
Definition at line 1522 of file kmp_csupport.c.

5.8.3.23 void _kmpc_for_static_fini ( ident_t * loc, kmp_int32 global_tid )

Parameters

<table>
<thead>
<tr>
<th>loc</th>
<th>Source location</th>
</tr>
</thead>
<tbody>
<tr>
<td>global_tid</td>
<td>Global thread id</td>
</tr>
</tbody>
</table>

Mark the end of a statically scheduled loop.
Definition at line 1549 of file kmp_csupport.c.

5.8.3.24 void _kmpc_for_static_init_4 ( ident_t * loc, kmp_int32 gtid, kmp_int32 schedtype, kmp_int32 plastiter, kmp_int32 * plower, kmp_int32 * pupper, kmp_int32 * pstride, kmp_int32 incr, kmp_int32 chunk )

Parameters

<table>
<thead>
<tr>
<th>loc</th>
<th>Source code location</th>
</tr>
</thead>
<tbody>
<tr>
<td>gtid</td>
<td>Global thread id of this thread</td>
</tr>
<tr>
<td>schedtype</td>
<td>Scheduling type</td>
</tr>
<tr>
<td>plastiter</td>
<td>Pointer to the &quot;last iteration&quot; flag</td>
</tr>
<tr>
<td>plower</td>
<td>Pointer to the lower bound</td>
</tr>
<tr>
<td>pupper</td>
<td>Pointer to the upper bound</td>
</tr>
<tr>
<td>pstride</td>
<td>Pointer to the stride</td>
</tr>
<tr>
<td>incr</td>
<td>Loop increment</td>
</tr>
<tr>
<td>chunk</td>
<td>The chunk size</td>
</tr>
</tbody>
</table>

Each of the four functions here are identical apart from the argument types.
The functions compute the upper and lower bounds and stride to be used for the set of iterations to be executed by the current thread from the statically scheduled loop that is described by the initial values of the bounds, stride, increment and chunk size.
Definition at line 798 of file kmp_sched.cpp.

5.8.3.25 void _kmpc_for_static_init_4u ( ident_t * loc, kmp_int32 gtid, kmp_int32 schedtype, kmp_int32 * plastiter, kmp_uint32 * plower, kmp_uint32 * pupper, kmp_int32 * pstride, kmp_int32 incr, kmp_int32 chunk )

See __kmpc_for_static_init_4
5.8.3.26 void __kmpc_for_static_init_8 (ident_t *loc, kmp_int32 gtid, kmp_int32 schedtype, kmp_int32 *plastiter, kmp_int64 *plower, kmp_int64 *pupper, kmp_int64 *pstride, kmp_int64 incr, kmp_int64 chunk)

See __kmpc_for_static_init_4
Definition at line 822 of file kmp_sched.cpp.

5.8.3.27 void __kmpc_for_static_init_8u (ident_t *loc, kmp_int32 gtid, kmp_int32 schedtype, kmp_int32 *plastiter, kmp_uint64 *plower, kmp_uint64 *pupper, kmp_int64 *pstride, kmp_int64 incr, kmp_int64 chunk)

See __kmpc_for_static_init_4
Definition at line 834 of file kmp_sched.cpp.

5.8.3.28 kmp_int32 __kmpc_master (ident_t *loc, kmp_int32 global_tid)

Parameters

loc | source location information.
global_tid | global thread number.

Returns

1 if this thread should execute the master block, 0 otherwise.

Definition at line 719 of file kmp_csupport.c.
Referenced by __kmpc_barrier_master_nowait().

5.8.3.29 void __kmpc_ordered (ident_t *loc, kmp_int32 gtid)

Parameters

loc | source location information.
gtid | global thread number.

Start execution of an ordered construct.
Definition at line 811 of file kmp_csupport.c.

5.8.3.30 kmp_int32 __kmpc_single (ident_t *loc, kmp_int32 global_tid)

Parameters

loc | source location information
global_tid | global thread number

Returns

One if this thread should execute the single construct, zero otherwise.

Test whether to execute a single construct. There are no implicit barriers in the two "single" calls, rather the compiler should introduce an explicit barrier if it is required.

Definition at line 1475 of file kmp_csupport.c.
5.8 Work Sharing

5.8.3.31  void __kmpc_team_static_init_4 ( ident_t * loc, kmp_int32 gtid, kmp_int32 * p_last, kmp_int32 * p_lb, kmp_int32 * p_ub, kmp_int32 * p_st, kmp_int32 incr, kmp_int32 chunk )

Parameters

<table>
<thead>
<tr>
<th></th>
<th>Source location</th>
</tr>
</thead>
<tbody>
<tr>
<td>loc</td>
<td>Global thread id</td>
</tr>
<tr>
<td>gtid</td>
<td>pointer to last iteration flag</td>
</tr>
<tr>
<td>p_last</td>
<td>pointer to Lower bound</td>
</tr>
<tr>
<td>p_lb</td>
<td>pointer to Upper bound</td>
</tr>
<tr>
<td>p_ub</td>
<td>Step (or increment if you prefer)</td>
</tr>
<tr>
<td>p_st</td>
<td>Loop increment</td>
</tr>
<tr>
<td>incr</td>
<td>The chunk size to block with</td>
</tr>
</tbody>
</table>

The functions compute the upper and lower bounds and stride to be used for the set of iterations to be executed by the current team from the statically scheduled loop that is described by the initial values of the bounds, stride, increment and chunk for the distribute construct as part of composite distribute parallel loop construct. These functions are all identical apart from the types of the arguments.

Definition at line 945 of file kmp_sched.cpp.

5.8.3.32  void __kmpc_team_static_init_4u ( ident_t * loc, kmp_int32 gtid, kmp_int32 * p_last, kmp_uint32 * p_lb, kmp_uint32 * p_ub, kmp_int32 * p_st, kmp_int32 incr, kmp_int32 chunk )

See __kmpc_team_static_init_4

Definition at line 957 of file kmp_sched.cpp.

5.8.3.33  void __kmpc_team_static_init_8 ( ident_t * loc, kmp_int32 gtid, kmp_int32 * p_last, kmp_int64 * p_lb, kmp_int64 * p_ub, kmp_int32 * p_st, kmp_int32 incr, kmp_int32 chunk )

See __kmpc_team_static_init_4

Definition at line 969 of file kmp_sched.cpp.

5.8.3.34  void __kmpc_team_static_init_8u ( ident_t * loc, kmp_int32 gtid, kmp_int32 * p_last, kmp_uint64 * p_lb, kmp_uint64 * p_ub, kmp_int64 * p_st, kmp_int64 incr, kmp_int64 chunk )

See __kmpc_team_static_init_4

Definition at line 981 of file kmp_sched.cpp.
5.9 Synchronization

Functions

- void __kmpc_flush (ident_t *loc)
- void __kmpc_barrier (ident_t *loc, kmp_int32 global_tid)
- kmp_int32 __kmpc_barrier_master (ident_t *loc, kmp_int32 global_tid)
- void __kmpc_end_barrier_master (ident_t *loc, kmp_int32 global_tid)
- kmp_int32 __kmpc_barrier_master_nowait (ident_t *loc, kmp_int32 global_tid)
- kmp_int32 __kmpc_reduce_nowait (ident_t *loc, kmp_int32 global_tid, kmp_int32 num_vars, size_t reduce_size, void *reduce_data, void (*)(void *)reduce_func)(void *lhs_data, void *rhs_data), kmp_critical_name *lck)
- void __kmpc_end_reduce_nowait (ident_t *loc, kmp_int32 global_tid, kmp_int32 global_tid)
- kmp_int32 __kmpc_reduce (ident_t *loc, kmp_int32 global_tid, kmp_int32 num_vars, size_t reduce_size, void *reduce_data, void (*)(void *)reduce_func)(void *lhs_data, void *rhs_data), kmp_critical_name *lck)
- void __kmpc_end_reduce (ident_t *loc, kmp_int32 global_tid, kmp_critical_name *lck)

5.9.1 Detailed Description

These functions are used for implementing barriers.

5.9.2 Function Documentation

5.9.2.1 void __kmpc_barrier (ident_t *loc, kmp_int32 global_tid)

Parameters

<table>
<thead>
<tr>
<th>loc</th>
<th>source location information</th>
</tr>
</thead>
<tbody>
<tr>
<td>global_tid</td>
<td>thread id.</td>
</tr>
</tbody>
</table>

Execute a barrier.

Definition at line 684 of file kmp_csupport.c.

5.9.2.2 kmp_int32 __kmpc_barrier_master (ident_t *loc, kmp_int32 global_tid)

Parameters

<table>
<thead>
<tr>
<th>loc</th>
<th>source location information</th>
</tr>
</thead>
<tbody>
<tr>
<td>global_tid</td>
<td>thread id.</td>
</tr>
</tbody>
</table>

Returns

one if the thread should execute the master block, zero otherwise

Start execution of a combined barrier and master. The barrier is executed inside this function.

Definition at line 1373 of file kmp_csupport.c.

5.9.2.3 kmp_int32 __kmpc_barrier_master_nowait (ident_t *loc, kmp_int32 global_tid)

Parameters

<table>
<thead>
<tr>
<th>loc</th>
<th>source location information</th>
</tr>
</thead>
<tbody>
<tr>
<td>global_tid</td>
<td>thread id.</td>
</tr>
</tbody>
</table>


5.9 Synchronization

Returns

one if the thread should execute the master block, zero otherwise

Start execution of a combined barrier and master(nowait) construct. The barrier is executed inside this function.
There is no equivalent "end" function, since the
Definition at line 1421 of file kmp_csupport.c.

5.9.2.4 void _kmpec_end_barrier_master ( ident_t ∗ loc, kmp_int32 global_tid )

Parameters

<table>
<thead>
<tr>
<th>loc</th>
<th>source location information</th>
</tr>
</thead>
<tbody>
<tr>
<td>global_tid</td>
<td>thread id.</td>
</tr>
</tbody>
</table>

Complete the execution of a combined barrier and master. This function should only be called at the completion of
the master code. Other threads will still be waiting at the barrier and this call releases them.
Definition at line 1403 of file kmp_csupport.c.

5.9.2.5 void _kmpec_end_reduce ( ident_t ∗ loc, kmp_int32 global_tid, kmp_critical_name ∗ lck )

Parameters

<table>
<thead>
<tr>
<th>loc</th>
<th>source location information</th>
</tr>
</thead>
<tbody>
<tr>
<td>global_tid</td>
<td>global thread id.</td>
</tr>
<tr>
<td>lck</td>
<td>pointer to the unique lock data structure</td>
</tr>
</tbody>
</table>

Finish the execution of a blocking reduce. The lck pointer must be the same as that used in the corresponding
start function.
Definition at line 2949 of file kmp_csupport.c.

5.9.2.6 void _kmpec_end_reduce_nowait ( ident_t ∗ loc, kmp_int32 global_tid, kmp_critical_name ∗ lck )

Parameters

<table>
<thead>
<tr>
<th>loc</th>
<th>source location information</th>
</tr>
</thead>
<tbody>
<tr>
<td>global_tid</td>
<td>global thread id.</td>
</tr>
<tr>
<td>lck</td>
<td>pointer to the unique lock data structure</td>
</tr>
</tbody>
</table>

Finish the execution of a reduce nowait.
Definition at line 2804 of file kmp_csupport.c.

5.9.2.7 void _kmpec_flush ( ident_t ∗ loc )

Parameters

| loc  | source location information |

Execute flush. This is implemented as a full memory fence. (Though depending on the memory ordering con-
vention obeyed by the compiler even that may not be necessary).
Definition at line 616 of file kmp_csupport.c.

5.9.2.8 kmp_int32 _kmpec_reduce ( ident_t ∗ loc, kmp_int32 global_tid, kmp_int32 num_vars, size_t reduce_size, void * reduce_data, void(*) (void *lhs_data, void *rhs_data) reduce_func, kmp_critical_name ∗ lck )
### Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>loc</code></td>
<td>source location information</td>
</tr>
<tr>
<td><code>global_tid</code></td>
<td>global thread number</td>
</tr>
<tr>
<td><code>num_vars</code></td>
<td>number of items (variables) to be reduced</td>
</tr>
<tr>
<td><code>reduce_size</code></td>
<td>size of data in bytes to be reduced</td>
</tr>
<tr>
<td><code>reduce_data</code></td>
<td>pointer to data to be reduced</td>
</tr>
<tr>
<td><code>reduce_func</code></td>
<td>callback function providing reduction operation on two operands and returning result of reduction in <code>lhs_data</code></td>
</tr>
<tr>
<td><code>lck</code></td>
<td>pointer to the unique lock data structure</td>
</tr>
</tbody>
</table>

### Returns

1 for the master thread, 0 for all other team threads, 2 for all team threads if atomic reduction needed.

A blocking reduce that includes an implicit barrier.

Definition at line 2862 of file `kmp_csupport.c`.

```c
kmp_int32 _kmpc_reduce_nowait ( ident_t * loc, kmp_int32 global_tid, kmp_int32 num_vars, size_t reduce_size, void * reduce_data, void(void (*)(void *, void *))reduce_func, kmp_critical_name * lck )
```

### Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>loc</code></td>
<td>source location information</td>
</tr>
<tr>
<td><code>global_tid</code></td>
<td>global thread number</td>
</tr>
<tr>
<td><code>num_vars</code></td>
<td>number of items (variables) to be reduced</td>
</tr>
<tr>
<td><code>reduce_size</code></td>
<td>size of data in bytes to be reduced</td>
</tr>
<tr>
<td><code>reduce_data</code></td>
<td>pointer to data to be reduced</td>
</tr>
<tr>
<td><code>reduce_func</code></td>
<td>callback function providing reduction operation on two operands and returning result of reduction in <code>lhs_data</code></td>
</tr>
<tr>
<td><code>lck</code></td>
<td>pointer to the unique lock data structure</td>
</tr>
</tbody>
</table>

### Returns

1 for the master thread, 0 for all other team threads, 2 for all team threads if atomic reduction needed.

The nowait version is used for a reduce clause with the nowait argument.

Definition at line 2665 of file `kmp_csupport.c`.
5.10 Thread private data support

Functions

- void __kmpc_copyprivate (ident_t *loc, kmp_int32 gtid, size_t cpy_size, void *cpy_data, void(*)(void *), kmp_int32 didit)
- void __kmpc_threadprivate_register (ident_t *loc, void *data, kmpc_ctor ctor, kmpc_cctorcctor, kmpc_dtor dctor)
- void *__kmpc_threadprivate_cached (ident_t *loc, kmp_int32 global_tid, void *data, size_t size, void ***cache)
- void __kmpc_threadprivate_register_vec (ident_t *loc, void *data, kmpc_ctor_vec ctor, kmpc_cctor_veccctor, kmpc_dtor_vec dctor, size_t vector_length)

- typedef void (*)(kmpc_ctor)(void *)
- typedef void(*)(kmpc_dtor)(void *)
- typedef void (*)(kmpc_ctor)(void *, void *)
- typedef void (*)(kmpc_ctor_vec)(void *, size_t)
- typedef void (*)(kmpc_dtor_vec)(void *, size_t)
- typedef void (*)(kmpc_ctor_vec)(void *, void *, size_t)

5.10.1 Detailed Description

These functions support copyin/out and thread private data.

5.10.2 Typedef Documentation

5.10.2.1 typedef void (*)(kmpc_ctor)(void *)

Pointer to an alternate constructor. The first argument is the this pointer.
Definition at line 1483 of file kmp.h.

5.10.2.2 typedef void (*)(kmpc_ctor_vec)(void *, size_t)

Array constructor. First argument is the this pointer Third argument the number of array elements.
Definition at line 1505 of file kmp.h.

5.10.2.3 typedef void (*)(kmpc_ctor)(void *)

Pointer to the constructor function. The first argument is the this pointer
Definition at line 1472 of file kmp.h.

5.10.2.4 typedef void (*)(kmpc_ctor_vec)(void *, size_t)

Array constructor. First argument is the this pointer Second argument the number of array elements.
Definition at line 1493 of file kmp.h.

5.10.2.5 typedef void (*)(kmpc_dtor)(void *)

Pointer to the destructor function. The first argument is the this pointer
Definition at line 1478 of file kmp.h.
5.10.2.6 typedef void( kmpc_dtor_vec)(void *, size_t)

Pointer to the array destructor function. The first argument is the this pointer Second argument the number of array elements.

Definition at line 1499 of file kmp.h.

5.10.3 Function Documentation

5.10.3.1 void _kmpc_copyprivate ( ident_t * loc, kmp_int32 gtid, size_t cpy_size, void * cpy_data, void(*)(void *, void *))

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>loc</td>
<td>source location information</td>
</tr>
<tr>
<td>gtid</td>
<td>global thread number</td>
</tr>
<tr>
<td>cpy_size</td>
<td>size of the cpy_data buffer</td>
</tr>
<tr>
<td>cpy_data</td>
<td>pointer to data to be copied</td>
</tr>
<tr>
<td>cpy_func</td>
<td>helper function to call for copying data</td>
</tr>
<tr>
<td>didit</td>
<td>flag variable: 1=single thread; 0=not single thread</td>
</tr>
</tbody>
</table>

__kmpc_copyprivate implements the interface for the private data broadcast needed for the copyprivate clause associated with a single region in an OpenMP * program (both C and Fortran). All threads participating in the parallel region call this routine. One of the threads (called the single thread) should have the didit variable set to 1 and all other threads should have that variable set to 0. All threads pass a pointer to a data buffer (cpy_data) that they have built.

The OpenMP specification forbids the use of nowait on the single region when a copyprivate clause is present. However, __kmpc_copyprivate implements a barrier internally to avoid race conditions, so the code generation for the single region should avoid generating a barrier after the call to __kmpc_copyprivate.

The gtid parameter is the global thread id for the current thread. The loc parameter is a pointer to source location information.

Internal implementation: The single thread will first copy its descriptor address (cpy_data) to a team-private location, then the other threads will each call the function pointed to by the parameter cpy_func, which carries out the copy by copying the data using the cpy_data buffer.

The cpy_func routine used for the copy and the contents of the data area defined by cpy_data and cpy_size may be built in any fashion that will allow the copy to be done. For instance, the cpy_data buffer can hold the actual data to be copied or it may hold a list of pointers to the data. The cpy_func routine must interpret the cpy_data buffer appropriately.

The interface to cpy_func is as follows:

```c
void cpy_func( void *destination, void *source )
```

where void *destination is the cpy_data pointer for the thread being copied to and void *source is the cpy_data pointer for the thread being copied from.

Definition at line 1766 of file kmp_csupport.c.

5.10.3.2 void _kmpc_threadprivate_cached ( ident_t * loc, kmp_int32 global_tid, void * data, size_t size, void *** cache )

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>loc</td>
<td>source location information</td>
</tr>
<tr>
<td>global_tid</td>
<td>global thread number</td>
</tr>
<tr>
<td>data</td>
<td>pointer to data to privatize</td>
</tr>
<tr>
<td>size</td>
<td>size of data to privatize</td>
</tr>
<tr>
<td>cache</td>
<td>pointer to cache</td>
</tr>
</tbody>
</table>
5.10 Thread private data support

Returns

pointer to private storage

Allocate private storage for threadprivate data.

Definition at line 649 of file kmp_threadprivate.c.

5.10.3.3 void _kmpc_threadprivate_register ( ident_t *loc, void *data, kmpc_ctor ctor, kmpc_cctor cctor, kmpc_dtor dtor )

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>loc</td>
<td>source location information</td>
</tr>
<tr>
<td>data</td>
<td>pointer to data being privatized</td>
</tr>
<tr>
<td>ctor</td>
<td>pointer to constructor function for data</td>
</tr>
<tr>
<td>cctor</td>
<td>pointer to copy constructor function for data</td>
</tr>
<tr>
<td>dtor</td>
<td>pointer to destructor function for data</td>
</tr>
</tbody>
</table>

Register constructors and destructors for thread private data. This function is called when executing in parallel, when we know the thread id.

Definition at line 551 of file kmp_threadprivate.c.

5.10.3.4 void _kmpc_threadprivate_register_vec ( ident_t *loc, void *data, kmpc_ctor_vec ctor, kmpc_cctor_vec cctor, kmpc_dtor_vec dtor, size_t vector_length )

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>loc</td>
<td>source location information</td>
</tr>
<tr>
<td>data</td>
<td>pointer to data being privatized</td>
</tr>
<tr>
<td>ctor</td>
<td>pointer to constructor function for data</td>
</tr>
<tr>
<td>cctor</td>
<td>pointer to copy constructor function for data</td>
</tr>
<tr>
<td>dtor</td>
<td>pointer to destructor function for data</td>
</tr>
<tr>
<td>vector_length</td>
<td>length of the vector (bytes or elements?)</td>
</tr>
</tbody>
</table>

Register vector constructors and destructors for thread private data.

Definition at line 718 of file kmp_threadprivate.c.
5.11 Statistics Gathering from OMPTB

Macros

• `#define KMP_FOREACH_COUNTER(macro, arg)`
  Add new counters under `KMP_FOREACH_COUNTER()` macro in `kmp_stats.h`.

• `#define KMP_FOREACH_EXPLICIT_TIMER(macro, arg)`
  Add new explicit timers under `KMP_FOREACH_EXPLICIT_TIMER()` macro.

• `#define KMP_TIME_BLOCK(name) blockTimer __BLOCKTIME__(__kmp_stats_thread_ptr->getTimer(TIMER_##name), TIMER_##name)`
  Uses specified timer (name) to time code block.

• `#define KMP_COUNT_VALUE(name, value) __kmp_stats_thread_ptr->getTimer(TIMER_##name)->addSample(value)`
  Adds value to specified timer (name).

• `#define KMP_COUNT_BLOCK(name) __kmp_stats_thread_ptr->getCounter(COUNTER_##name)->increment()`
  Increments specified counter (name).

• `#define KMP_START_EXPLICIT_TIMER(name) __kmp_stats_thread_ptr->getExplicitTimer(EXPLICIT_TIMER_##name)->start(TIMER_##name)`
  "Starts" an explicit timer which will need a corresponding `KMP_STOP_EXPLICIT_TIMER()` macro.

• `#define KMP_STOP_EXPLICIT_TIMER(name) __kmp_stats_thread_ptr->getExplicitTimer(EXPLICIT_TIMER_##name)->stop(TIMER_##name)`
  "Stops" an explicit timer.

• `#define KMP_OUTPUT_STATS(heading_string) __kmp_output_stats(heading_string)`
  Outputs the current thread statistics and reset them.

• `#define KMP_INIT_PARTITIONED_TIMERS(name) __kmp_stats_thread_ptr->getPartitionedTimers()->init(timerPair(EXPLICIT_TIMER_##name, TIMER_##name))`
  Initializes the partitioned timers to begin with name.

• `#define KMP_RESET_STATS() __kmp_reset_stats()`
  Resets all stats (counters to 0, timers to 0 elapsed ticks)

Enumerations

• `enum stats_flags_e {
    noTotal = 1 << 0, onlyInMaster = 1 << 1, noUnits = 1 << 2, notInMaster = 1 << 3,
    logEvent = 1 << 4
} `
  Flags to describe the statistic (timer or counter)

• `enum stats_state_e`
  The states which a thread can be in

5.11.1 Detailed Description

These macros support profiling the libomp5 library. Use `-stats=on` when building with build.pl to enable and then use the KMP_ macros to profile (through counts or clock ticks) libomp5 during execution of an OpenMP program.

5.11.2 Environment Variables

This section describes the environment variables relevant to stats-gathering in libomp5
5.11 Statistics Gathering from OMPTB

This environment variable is set to an output filename that will be appended NOT OVERWRITTEN if it exists. If this environment variable is undefined, the statistics will be output to stderr.

KMP_STATS_THREADS

This environment variable indicates to print thread-specific statistics as well as aggregate statistics. Each thread’s statistics will be shown as well as the collective sum of all threads. The values "true", "on", "1", "yes" will all indicate to print per thread statistics.

5.11.3 Macro Definition Documentation

5.11.3.1 #define KMP_COUNT_BLOCK( name ) _kmp_stats_thread_ptr->getCounter(COUNTER_##name)->increment()

Increments specified counter (name).

Parameters

| name   | counter name as specified under the KMP_FOREACH_COUNTER() macro |

Use KMP_COUNT_BLOCK(name, value) macro to increment a statistics counter for the executing thread.

Definition at line 795 of file kmp_stats.h.

Referenced by __kmpc_barrier(), __kmpc_critical(), __kmpc_fork_call(), __kmpc_fork_teams(), __kmpc_master(), __kmpc_reduce(), __kmpc_reduce_nowait(), and __kmpc_single().

5.11.3.2 #define KMP_COUNT_VALUE( name, value ) _kmp_stats_thread_ptr->getTimer(TIMER_##name)->addSample(value)

Adds value to specified timer (name).

Parameters

| name   | timer name as specified under the KMP_FOREACH_TIMER() macro |
| value  | double precision sample value to add to statistics for the timer |

Use KMP_COUNT_VALUE(name, value) macro to add a particular value to a timer statistics.

Definition at line 783 of file kmp_stats.h.

5.11.3.3 #define KMP_FOREACH_COUNTER( macro, arg )

Value:

macro (OMP_PARALLEL, stats_flags_e::onlyInMaster | stats_flags_e::noTotal, arg) |
  macro (OMP_NESTED_PARALLEL, 0, arg) |
  macro (OMP_FOR_static, 0, arg) |
  macro (OMP_FOR_dynamic, 0, arg) |
  macro (OMP_DISTRIBUTE, 0, arg) |
  macro (OMP_BARRIER, 0, arg) |
  macro (OMP_CRITICAL, 0, arg) |
  macro (OMP_SINGLE, 0, arg) |
  macro (OMP_MASTER, 0, arg) |
  macro (OMP_TEAMS, 0, arg) |
  macro (OMP_set_lock, 0, arg) |
  macro (OMP茚 test_lock, 0, arg) |
  macro (REDUCE_wait, 0, arg) |
  macro (REDUCE_nowait, 0, arg) |
  macro (OMP茚 TASKYIELD, 0, arg) |
  macro (OMP茚 TASKLOOP, 0, arg) |
  macro (TASK_executed, 0, arg) |
  macro (TASK_cancelled, 0, arg) |
  macro (TASK_stolen, 0, arg) |
  macro (LAST,0, arg)
Add new counters under `KMP_FOREACH_COUNTER()` macro in `kmp_stats.h`.

**Parameters**

<table>
<thead>
<tr>
<th>macro</th>
<th>a user defined macro that takes three arguments - macro(COUNTER_NAME, flags, arg)</th>
</tr>
</thead>
<tbody>
<tr>
<td>arg</td>
<td>a user defined argument to send to the user defined macro</td>
</tr>
</tbody>
</table>

A counter counts the occurrence of some event. Each thread accumulates its own count, at the end of execution the counts are aggregated treating each thread as a separate measurement. (Unless onlyInMaster is set, in which case there’s only a single measurement). The min, mean, max are therefore the values for the threads. Adding the counter here and then putting a `KMP_BLOCK_COUNTER(name)` at the point you want to count is all you need to do. All of the tables and printing is generated from this macro. Format is “macro(name, flags, arg)”

Definition at line 106 of file `kmp_stats.h`.

5.11.3.4 `#define KMP_FOREACH_EXPLICIT_TIMER( macro, arg )`

**Value:**

```c
macro(OMP_worker_thread_life, 0, arg)  
macro(OMP_static_scheduling, 0, arg)  
macro(OMP_dynamic_scheduling, 0, arg)  
macro(OMP_critical, 0, arg)  
macro(OMP_critical_wait, 0, arg)  
macro(OMP_single, 0, arg)  
macro(OMP_master, 0, arg)  
macro(OMP_idle, 0, arg)  
macro(OMP_plain_barrier, 0, arg)  
macro(OMP_fork_join_barrier, 0, arg)  
macro(OMP_parallel, 0, arg)  
macro(OMP_task_immediate, 0, arg)  
macro(OMP_task_taskwait, 0, arg)  
macro(OMP_task_taskyield, 0, arg)  
macro(OMP_task_taskgroup, 0, arg)  
macro(OMP_task_join_bar, 0, arg)  
macro(OMP_task_plain_bar, 0, arg)  
macro(OMP_serial, 0, arg)  
macro(OMP_taskloop_scheduling, 0, arg)  
KMP_FOREACH_EXPLICIT_DEVELOPER_TIMER(macro,arg)  
macro(LAST, 0, arg)
```

Add new explicit timers under `KMP_FOREACH_EXPLICIT_TIMER()` macro.

**Parameters**

<table>
<thead>
<tr>
<th>macro</th>
<th>a user defined macro that takes three arguments - macro(TIMER_NAME, flags, arg)</th>
</tr>
</thead>
<tbody>
<tr>
<td>arg</td>
<td>a user defined argument to send to the user defined macro</td>
</tr>
</tbody>
</table>

**Warning**

YOU MUST HAVE THE SAME NAMED TIMER UNDER `KMP_FOREACH_TIMER()` OR ELSE BAD THINGS WILL HAPPEN!

Explicit timers are ones where we need to allocate a timer itself (as well as the accumulated timing statistics). We allocate these on a per-thread basis, and explicitly start and stop them. Block timers just allocate the timer itself on the stack, and use the destructor to notice block exit; they don’t need to be defined here. The name here should be the same as that of a timer above.

Definition at line 254 of file `kmp_stats.h`.

5.11.3.5 `#define KMP_INIT_PARTITIONED_TIMERS( name ) _kmp_stats_thread_ptr->getPartitionedTimers()->init(timerPair(E-XPLICIT_TIMER, #name, TIMER_##name))`

Initializes the partitioned timers to begin with name.
Parameters

| name | timer which you want this thread to begin with |

Definition at line 850 of file kmp_stats.h.

5.11.3.6  

```c
#define KMP_OUTPUT_STATS( heading_string ) .kmp_output_stats(heading_string)
```

Outputs the current thread statistics and reset them.

Parameters

| heading_string | heading put above the final stats output |

Explicitly stops all timers and outputs all stats. Environment variable, `OMPTB_STATSFILE=filename`, can be used to output the stats to a filename instead of stderr. Environment variable, `OMPTB_STATSTHREADS=true|undefined`, can be used to output thread specific stats. For now the `OMPTB_STATSTHREADS` environment variable can either be defined with any value, which will print out thread specific stats, or it can be undefined (not specified in the environment) and thread specific stats won’t be printed. It should be noted that all statistics are reset when this macro is called.

Definition at line 840 of file kmp_stats.h.

5.11.3.7  

```c
#define KMP_RESET_STATS() .kmp_reset_stats()
```

resets all stats (counters to 0, timers to 0 elapsed ticks)
Reset all stats for all threads.

Definition at line 879 of file kmp_stats.h.

5.11.3.8  

```c
#define KMP_STOP_EXPLICIT_TIMER( name ) .kmp_stats_thread_ptr->getExplicitTimer(EXPLICIT_TIMER##name)->stop(TIMER##name)
```

"Starts" an explicit timer which will need a corresponding `KMP_STOP_EXPLICIT_TIMER()` macro.

Parameters

| name | explicit timer name as specified under the `KMP_FOREACH_EXPLICIT_TIMER()` macro |

Use to start a timer. This will need a corresponding `KMP_STOP_EXPLICIT_TIMER()` macro to stop the timer unlike the `KMP_TIME_BLOCK(name)` macro which has an implicit stopping macro at the end of the code block. All explicit timers are stopped at library exit time before the final statistics are outputted.

Definition at line 809 of file kmp_stats.h.

Referenced by `__kmpc_critical()`.

5.11.3.9  

```c
#define KMP_STOP_EXPLICIT_TIMER( name ) .kmp_stats_thread_ptr->getExplicitTimer(EXPLICIT_TIMER##name)->stop(TIMER##name)
```

"Stop"s an explicit timer.

Parameters

| name | explicit timer name as specified under the `KMP_FOREACH_EXPLICIT_TIMER()` macro |

Use `KMP_STOP_EXPLICIT_TIMER(name)` to stop a timer. When this is done, the time between the last `KMP_-START_EXPLICIT_TIMER(name)` and this `KMP_STOP_EXPLICIT_TIMER(name)` will be added to the timer’s stat
The timer will then be reset. After the `KMP_STOP_EXPLICIT_TIMER(name)` macro is called, another call to `KMP_START_EXPLICIT_TIMER(name)` will start the timer once again.

Definition at line 823 of file kmp_stats.h.

```c
#define KMP_TIME_BLOCK(name) blockTimer_BLOCKTIME((kmp_stats_thread_ptr->getTimer(TIMER##name), TIMER##name))
```

Use specified timer (name) to time code block.

**Parameters**

| name | timer name as specified under the KMP_FOREACH_TIMER() macro |

Use `KMP_TIME_BLOCK(name)` macro to time a code block. This will record the time taken in the block and use the destructor to stop the timer. Convenient! With this definition you can't have more than one `KMP_TIME_BLOCK` in the same code block. I don't think that's a problem.

Definition at line 770 of file kmp_stats.h.

### Enumeration Type Documentation

#### 5.11.4.1 `enum stats_flags_e`

flags to describe the statistic (timer or counter)

**Enumerator**

- `noTotal` do not show a TOTAL_aggregation for this statistic
- `onlyInMaster` statistic is valid only for master
- `noUnits` statistic doesn’t need units printed next to it in output
- `notInMaster` statistic is valid only for non-master threads
- `logEvent` statistic can be logged on the event timeline when KMP_STATS_EVENTS is on (valid only for timers)

Definition at line 65 of file kmp_stats.h.
5.12 Tasking support

Functions

- `kmp_int32 __kmpc_omp_task_with_deps (ident_t *loc_ref, kmp_int32 gtid, kmp_task_t *new_task, kmp_int32 ndeps, kmp_depend_info_t *dep_list, kmp_int32 ndeps_noalias, kmp_depend_info_t *noalias_dep_list)`
- `void __kmpc_omp_wait_deps (ident_t *loc_ref, kmp_int32 gtid, kmp_int32 ndeps, kmp_depend_info_t *dep_list, kmp_int32 ndeps_noalias, kmp_depend_info_t *noalias_dep_list)`

5.12.1 Detailed Description

These functions support tasking constructs.

5.12.2 Function Documentation

5.12.2.1 kmp_int32 __kmpc_omp_task_with_deps (ident_t *loc_ref, kmp_int32 gtid, kmp_task_t *new_task, kmp_int32 ndeps, kmp_depend_info_t *dep_list, kmp_int32 ndeps_noalias, kmp_depend_info_t *noalias_dep_list)

**Parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>loc_ref</code></td>
<td>location of the original task directive</td>
</tr>
<tr>
<td><code>gtid</code></td>
<td>Global Thread ID of encountering thread</td>
</tr>
<tr>
<td><code>new_task</code></td>
<td>task thunk allocated by <code>__kmp_omp_task_alloc()</code> for the &quot;new task&quot;</td>
</tr>
<tr>
<td><code>ndeps</code></td>
<td>Number of depend items with possible aliasing</td>
</tr>
<tr>
<td><code>dep_list</code></td>
<td>List of depend items with possible aliasing</td>
</tr>
<tr>
<td><code>ndeps_noalias</code></td>
<td>Number of depend items with no aliasing</td>
</tr>
<tr>
<td><code>noalias_dep_list</code></td>
<td>List of depend items with no aliasing</td>
</tr>
</tbody>
</table>

**Returns**

Returns either TASK_CURRENT_NOT_QUEUED if the current task was not suspendend and queued, or TASK_CURRENT_QUEUED if it was suspended and queued

Schedule a non-thread-switchable task with dependences for execution

Definition at line 443 of file kmp_taskdeps.cpp.

5.12.2.2 void __kmpc_omp_wait_deps (ident_t *loc_ref, kmp_int32 gtid, kmp_int32 ndeps, kmp_depend_info_t *dep_list, kmp_int32 ndeps_noalias, kmp_depend_info_t *noalias_dep_list)

**Parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>loc_ref</code></td>
<td>location of the original task directive</td>
</tr>
<tr>
<td><code>gtid</code></td>
<td>Global Thread ID of encountering thread</td>
</tr>
<tr>
<td><code>ndeps</code></td>
<td>Number of depend items with possible aliasing</td>
</tr>
<tr>
<td><code>dep_list</code></td>
<td>List of depend items with possible aliasing</td>
</tr>
<tr>
<td><code>ndeps_noalias</code></td>
<td>Number of depend items with no aliasing</td>
</tr>
<tr>
<td><code>noalias_dep_list</code></td>
<td>List of depend items with no aliasing</td>
</tr>
</tbody>
</table>

Blocks the current task until all specifies dependencies have been fulfilled.

Definition at line 556 of file kmp_taskdeps.cpp.

Referenced by `__kmpc_omp_task_with_deps()`.
5.13 User visible functions

These functions can be called directly by the user, but are runtime library specific, rather than being OpenMP interfaces.
Chapter 6

Class Documentation

6.1 hierarchy_info Class Reference

#include <kmp_affinity.h>

Public Attributes

• kmp_uint32 maxLevels
• kmp_uint32 depth
• kmp_uint32 *numPerLevel

Static Public Attributes

• static const kmp_uint32 maxLeaves = 4

6.1.1 Detailed Description

A structure for holding machine-specific hierarchy info to be computed once at init. This structure represents a mapping of threads to the actual machine hierarchy, or to our best guess at what the hierarchy might be, for the purpose of performing an efficient barrier. In the worst case, when there is no machine hierarchy information, it produces a tree suitable for a barrier, similar to the tree used in the hyper barrier.

Definition at line 166 of file kmp_affinity.h.

6.1.2 Member Data Documentation

6.1.2.1 kmp_uint32 hierarchy_info::depth

This is specifically the depth of the machine configuration hierarchy, in terms of the number of levels along the longest path from root to any leaf. It corresponds to the number of entries in numPerLevel if we exclude all but one trailing 1.

Definition at line 181 of file kmp_affinity.h.

6.1.2.2 const kmp_uint32 hierarchy_info::maxLeaves = 4 [static]

Good default values for number of leaves and branching factor, given no affinity information. Behaves a bit like hyper barrier.

Definition at line 170 of file kmp_affinity.h.
6.1.2.3 kmp_uint32 hierarchy_info::maxLevels

Number of levels in the hierarchy. Typical levels are threads/core, cores/package or socket, packages/node, nodes/machine, etc. We don’t want to get specific with nomenclature. When the machine is oversubscribed we add levels to duplicate the hierarchy, doubling the thread capacity of the hierarchy each time we add a level.

Definition at line 176 of file kmp_affinity.h.

6.1.2.4 kmp_uint32* hierarchy_info::numPerLevel

Level 0 corresponds to leaves. numPerLevel[i] is the number of children the parent of a node at level i has. For example, if we have a machine with 4 packages, 4 cores/package and 2 HT per core, then numPerLevel = {2, 4, 4, 1, 1}. All empty levels are set to 1.

Definition at line 190 of file kmp_affinity.h.

The documentation for this class was generated from the following file:

- kmp_affinity.h

6.2 ident Struct Reference

#include <kmp.h>

Public Attributes

- kmp_int32 reserved_1
- kmp_int32 flags
- kmp_int32 reserved_2
- kmp_int32 reserved_3
- char const* psource

6.2.1 Detailed Description

The ident structure that describes a source location.

Definition at line 211 of file kmp.h.

6.2.2 Member Data Documentation

6.2.2.1 kmp_int32 ident::flags

also f.flags; KMP_IDENT_xxx flags; KMP_IDENT_KMPC identifies this union member

Definition at line 213 of file kmp.h.

Referenced by __kmpc_end_serialized_parallel().

6.2.2.2 char const* ident::psource

String describing the source location.

The string is composed of semi-colon separated fields which describe the source file, the function and a pair of line numbers that delimit the construct.
Definition at line 220 of file kmp.h.
Referenced by __kmpc_ok_to_fork().

6.2.2.3  kmp_int32 ident::reserved_1
might be used in Fortran; see above
Definition at line 212 of file kmp.h.

6.2.2.4  kmp_int32 ident::reserved_2
not really used in Fortran any more; see above
Definition at line 214 of file kmp.h.

6.2.2.5  kmp_int32 ident::reserved_3
source[4] in Fortran, do not use for C++
Definition at line 219 of file kmp.h.

The documentation for this struct was generated from the following file:

- kmp.h

6.3  kmp_flag<P>  Class Template Reference

#include <kmp_wait_release.h>

Public Member Functions

- volatile P * get ()
- void set (volatile P *new_loc)
- flag_type get_type ()

Private Attributes

- volatile P * loc
- flag_type t

6.3.1  Detailed Description

template<
typename P>
class kmp_flag<P>

Base class for wait/release volatile flag
Definition at line 68 of file kmp_wait_release.h.

6.3.2  Member Function Documentation

6.3.2.1  template<
typename P>
volatile P * kmp_flag<P>::get () [inline]
Returns
the pointer to the actual flag

Definition at line 77 of file kmp_wait_release.h.

6.3.2.2 template<typename P> flag_type kmp_flag<P>::get() [inline]

Returns
the flag_type

Definition at line 85 of file kmp_wait_release.h.

6.3.2.3 template<typename P> void kmp_flag<P>::set(volatile P*p new_loc) [inline]

Parameters
new_loc in set loc to point at new_loc

Definition at line 81 of file kmp_wait_release.h.

6.3.3 Member Data Documentation

6.3.3.1 template<typename P> volatile P*kmp_flag<P>::loc [private]

Pointer to the flag storage that is modified by another thread
Definition at line 69 of file kmp_wait_release.h.
Referenced by kmp_flag<kmp_uint32>::get(), and kmp_flag<kmp_uint32>::set().

6.3.3.2 template<typename P> flag_type kmp_flag<P>::t [private]

"Type" of the flag in loc
Definition at line 70 of file kmp_wait_release.h.
Referenced by kmp_flag<kmp_uint32>::get_type().

The documentation for this class was generated from the following file:

- kmp_wait_release.h
Index

__kmpc_barrier
  Synchronization, 38
__kmpc_barrier_master
  Synchronization, 38
__kmpc_barrier_master_nowait
  Synchronization, 38
__kmpc_begin
  Startup and Shutdown, 24
__kmpc_bound_num_threads
  Thread Information, 27
__kmpc_bound_thread_num
  Thread Information, 27
__kmpc_copyprivate
  Thread private data support, 42
__kmpc_critical
  Work Sharing, 31
__kmpc_dispatch_fini_4
  Work Sharing, 31
__kmpc_dispatch_fini_4u
  Work Sharing, 31
__kmpc_dispatch_fini_8
  Work Sharing, 31
__kmpc_dispatch_fini_8u
  Work Sharing, 31
__kmpc_dispatch_init_4
  Work Sharing, 32
__kmpc_dispatch_init_4u
  Work Sharing, 32
__kmpc_dispatch_init_8
  Work Sharing, 32
__kmpc_dispatch_init_8u
  Work Sharing, 32
__kmpc_dispatch_next_4
  Work Sharing, 32
__kmpc_dispatch_next_4u
  Work Sharing, 33
__kmpc_dispatch_next_8
  Work Sharing, 33
__kmpc_dispatch_next_8u
  Work Sharing, 33
__kmpc_dist_dispatch_init_4
  Work Sharing, 33
__kmpc_dist_for_static_init_4
  Work Sharing, 33
__kmpc_dist_for_static_init_4u
  Work Sharing, 34
__kmpc_dist_for_static_init_8
  Work Sharing, 34
__kmpc_dist_for_static_init_8u
  Work Sharing, 34
__kmpc_dist_for_static_init_8u
  Work Sharing, 34
__kmpc_end
  Startup and Shutdown, 24
__kmpc_end_barrier_master
  Synchronization, 38
__kmpc_end_critical
  Synchronization, 39
__kmpc_end_master
  Work Sharing, 34
__kmpc_end_ordered
  Work Sharing, 34
__kmpc_end_reduce
  Work Sharing, 34
__kmpc_end_reduce_nowait
  Synchronization, 39
__kmpc_end_serialized_parallel
  Parallel (fork/join), 25
__kmpc_end_single
  Work Sharing, 35
__kmpc_flush
  Synchronization, 39
__kmpc_for_static_fini
  Work Sharing, 35
__kmpc_for_static_init_4
  Work Sharing, 35
__kmpc_for_static_init_4u
  Work Sharing, 35
__kmpc_for_static_init_8
  Work Sharing, 36
__kmpc_for_static_init_8u
  Work Sharing, 36
__kmpc_fork_call
  Parallel (fork/join), 25
__kmpc_fork_teams
  Parallel (fork/join), 26
__kmpc_global_num_threads
  Thread Information, 27
__kmpc_global_thread_num
  Thread Information, 28
__kmpc_in_parallel
  Thread Information, 28
__kmpc_master
  Work Sharing, 36
__kmpc_ok_to_fork
  Deprecated Functions, 23
__kmpc_omp_task_with_deps
  Tasking support, 49
__kmpc_omp_wait_deps
  Tasking support, 49
__kmpc_ordered
   Work Sharing, 36
__kmpc_push_num_teams
   Parallel (fork/join), 26
__kmpc_push_num_threads
   Parallel (fork/join), 26
__kmpc_reduce
   Synchronization, 39
__kmpc_reduce_nowait
   Synchronization, 40
__kmpc_serialized_parallel
   Parallel (fork/join), 26
__kmpc_single
   Work Sharing, 36
__kmpc_team_static_init_4
   Work Sharing, 36
__kmpc_team_static_init_4u
   Work Sharing, 37
__kmpc_team_static_init_8
   Work Sharing, 37
__kmpc_team_static_init_8u
   Work Sharing, 37
__kmpc_threadprivate_cached
   Thread private data support, 42
__kmpc_threadprivate_register
   Thread private data support, 43
__kmpc_threadprivate_register_vec
   Thread private data support, 43

Atomic Operations, 13

Basic Types, 21
   ident_t, 22
   KMP_IDENT_AUTOPAR, 21
   KMP_IDENT_IMB, 21
   KMP_IDENT_KMPC, 21

Deprecated Functions, 23
   __kmpc_ok_to_fork, 23

depth
   hierarchy_info, 51

flag32
   Wait/Release operations, 20
flag64
   Wait/Release operations, 20
flag_oncore
   Wait/Release operations, 20
flag_type
   Wait/Release operations, 20
flags
   ident, 52
   get
   kmp_flag, 53
   get_type
   kmp_flag, 54
   hierarchy_info, 51
   depth, 51

maxLeaves, 51
maxLevels, 51
numPerLevel, 52

ident, 52
   flags, 52
   psource, 52
   reserved_1, 53
   reserved_2, 53
   reserved_3, 53

KMP_COUNT_BLOCK
   Statistics Gathering from OMPTB, 45
KMP_COUNT_VALUE
   Statistics Gathering from OMPTB, 45
KMP_IDENT_AUTOPAR
   Basic Types, 21
KMP_IDENT_IMB
   Basic Types, 21
KMP_IDENT_KMPC
   Basic Types, 21
KMP_OUTPUT_STATS
   Statistics Gathering from OMPTB, 47
KMP_RESET_STATS
   Statistics Gathering from OMPTB, 47
KMP_TIME_BLOCK
   Statistics Gathering from OMPTB, 48
kmp_distribute_static
   Work Sharing, 31
kmp_distribute_static_chunked
   Work Sharing, 31
kmp_nm_auto
   Work Sharing, 31
kmp_nm_guided_chunked
   Work Sharing, 31
kmp_nm_lower
   Work Sharing, 31
kmp_nm_ord_auto
   Work Sharing, 31
kmp_nm_ord_static
   Work Sharing, 31
kmp_nm_static
   Work Sharing, 31
kmp_nm_upper
   Work Sharing, 31
kmp_ord_auto
   Work Sharing, 30
kmp_ord_lower
   Work Sharing, 30
kmp_ord_static
   Work Sharing, 30
kmp_ord_upper
   Work Sharing, 30
kmp_sch_auto
   Work Sharing, 30
kmp_sch_default
   Work Sharing, 31
INDEX

kmp_sch_guided_chunked
Work Sharing, 30

kmp_sch_lower
Work Sharing, 30

kmp_sch_static
Work Sharing, 30

kmp_sch_static_steal
Work Sharing, 30

kmp_sch_upper
Work Sharing, 30

kmp_flag
get, 53
get_type, 54
loc, 54
set, 54
t, 54

kmp_flag< P >, 53

kmpc_cctor
Thread private data support, 41

kmpc_cctor_vec
Thread private data support, 41

kmpc_ctor
Thread private data support, 41

kmpc_ctor_vec
Thread private data support, 41

kmpc_dtor
Thread private data support, 41

kmpc_dtor_vec
Thread private data support, 41

kmpc_micro
Thread private data support, 41

Parallel (fork/join), 25

loc
kmp_flag, 54

logEvent
Statistics Gathering from OMPTB, 48

maxLeaves
hierarchy_info, 51

maxLevels
hierarchy_info, 51

noTotal
Statistics Gathering from OMPTB, 48

noUnits
Statistics Gathering from OMPTB, 48

notInMaster
Statistics Gathering from OMPTB, 48

numPerLevel
hierarchy_info, 52

onlyInMaster
Statistics Gathering from OMPTB, 48

Parallel (fork/join), 25

__kmpc_push_num_teams, 26
__kmpc_push_num_threads, 26
__kmpc_serialized_parallel, 26
kmpc_micro, 25

psource
ident, 52

reserved_1
ident, 53

reserved_2
ident, 53

reserved_3
ident, 53

sched_type
Work Sharing, 30

set
kmp_flag, 54

Startup and Shutdown, 24

__kmpc_begin, 24

__kmpc_end, 24

Statistics Gathering from OMPTB
logEvent, 48

noTotal, 48

noUnits, 48

notInMaster, 48

onlyInMaster, 48

Statistics Gathering from OMPTB, 44

KMP_COUNT_BLOCK, 45

KMP_COUNT_VALUE, 45

KMP_RESET_STATS, 47

KMP_TIME_BLOCK, 48

stats_flags_e, 48

stats_flags_e
Statistics Gathering from OMPTB, 48

Synchronization, 38

__kmpc_barrier, 38

__kmpc_barrier_master, 38

__kmpc_barrier_master_nowait, 38

__kmpc_end_barrier_master, 39

__kmpc_end_reduce, 39

__kmpc_end_reduce_nowait, 39

__kmpc_flush, 39

__kmpc_reduce, 39

__kmpc_reduce_nowait, 40


t
kmp_flag, 54

Tasking support, 49

__kmpc_omp_task_with_deps, 49

__kmpc_omp_wait_deps, 49

Thread Information, 27

__kmpc_bound_num_threads, 27

__kmpc_bound_thread_num, 27

__kmpc_global_num_threads, 27

__kmpc_global_thread_num, 28

__kmpc_in_parallel, 28

Thread private data support, 41

__kmpc_copyprivate, 42

__kmpc_threadprivate_cached, 42
INDEX

__kmpc_threadprivate_register, 43
__kmpc_threadprivate_register_vec, 43
kmpc_ctor, 41
kmpc_ctor_vec, 41
kmpc_dtor, 41
kmpc_dtor_vec, 41

User visible functions, 50

Wait/Release operations
flag32, 20
flag64, 20
flag_oncore, 20

Wait/Release operations, 20
flag_type, 20

Work Sharing, 29
__kmpc_critical, 31
__kmpc_dispatch_fini_4, 31
__kmpc_dispatch_fini_4u, 31
__kmpc_dispatch_fini_8, 31
__kmpc_dispatch_fini_8u, 31
__kmpc_dispatch_init_4, 32
__kmpc_dispatch_init_4u, 32
__kmpc_dispatch_init_8, 32
__kmpc_dispatch_init_8u, 32
__kmpc_dispatch_next_4, 32
__kmpc_dispatch_next_4u, 33
__kmpc_dispatch_next_8, 33
__kmpc_dispatch_next_8u, 33
__kmpc_dist_dispatch_init_4, 33
__kmpc_dist_for_static_init_4, 33
__kmpc_dist_for_static_init_4u, 34
__kmpc_dist_for_static_init_8, 34
__kmpc_dist_for_static_init_8u, 34
__kmpc_end_critical, 34
__kmpc_end_master, 34
__kmpc_end_ordered, 34
__kmpc_end_single, 35
__kmpc_for_static_fini, 35
__kmpc_for_static_init_4, 35
__kmpc_for_static_init_4u, 35
__kmpc_for_static_init_8, 36
__kmpc_for_static_init_8u, 36
__kmpc_master, 36
__kmpc_ordered, 36
__kmpc_single, 36
__kmpc_team_static_init_4, 36
__kmpc_team_static_init_4u, 37
__kmpc_team_static_init_8, 37
__kmpc_team_static_init_8u, 37

kmp_distribute_static, 31
kmp_distribute_static_chunked, 31
kmp_nm_auto, 31
kmp_nm_guided_chunked, 31
kmp_nm_ord_auto, 31
kmp_nm_ord_static, 31

kmp_nm_static, 31
kmp_nm_upper, 31
kmp_ord_auto, 30
kmp_ord_lower, 30
kmp_ord_static, 30
kmp_ord_upper, 30
kmp_sch_auto, 30
kmp_sch_default, 31
kmp_sch_guided_chunked, 30
kmp_sch_lower, 30
kmp_sch_static, 30
kmp_sch_static_steal, 30
kmp_sch_upper, 30
sched_type, 30

Generated on Mon Aug 8 2016 17:18:34 for Intel® OpenMP® Runtime Library by Doxygen