Intel® OpenMP* Runtime Library

Generated by Doxygen 1.8.3.1

Wed Jul 1 2015 11:04:09

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.

Contents

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.

	icc/icl	**gcc**
Linux* OS	Yes(1,5)	Yes(2,4)
OS X*	Yes(1,3,4)	No
Windows* OS	Yes(1,4)	No

(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

```
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...)

```
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.

```
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.

```
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;
}

struct main_10_reduction_t_5 { float r_10_rpr; };

static kmp_critical_name lck = { 0 };
static ident_t loc10; // 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;
```

```
        case 2:
            __kmpc_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:

Chapter 3

Hierarchical Index

3.1 Class Hierarchy

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

Chapter 4

Class Index

4.1 Class List

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

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

```
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.

Data Type	Data type encoding
int8_t	`fixed1`
uint8_t	`fixed1u`
int16_t	`fixed2`
uint16_t	`fixed2u`
int32_t	`fixed4`
uint32_t	`fixed4u`
int32_t	`fixed8`
uint32_t	`fixed8u`
float	`float4`
double	`float8`
float 10 (8087 eighty bit float)	`float10`
complex<float>	`cmplx4`
complex<double>	`cmplx8`
complex<float10>	`cmplx10`

Operation	Operation encoding
+	add
-	sub
*	mul
/	div

&	andb
<<	shl
>>	shr
|	orb
^	xor
&&	andl
||	orl
maximum	max
minimum	min
.eqv.	eqv
.neqv.	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`)

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

Parameters

ident_t	a pointer to source location
gtid	the global thread id
lhs	a pointer to the left operand
rhs	the right operand

`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

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

Parameters

ident_t	a pointer to source location
gtid	the global thread id
lhs	a pointer to the left operand
rhs	the right operand
flag	one if the result is to be captured *after* the operation, zero if captured *before*.

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

```
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

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

For the write operations the form is

```
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.

Functons 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_swp
__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_shr
__kmpc_atomic_fixed1u_shr_cpt
__kmpc_atomic_fixed1u_shr_cpt_rev
__kmpc_atomic_fixed1u_shr_rev
__kmpc_atomic_fixed2_add
```

__kmpc_atomic_fixed2_add_cpt
__kmpc_atomic_fixed2_add_fp
__kmpc_atomic_fixed2_andb
__kmpc_atomic_fixed2_andb_cpt
__kmpc_atomic_fixed2_andl
__kmpc_atomic_fixed2_andl_cpt
__kmpc_atomic_fixed2_div
__kmpc_atomic_fixed2_div_cpt
__kmpc_atomic_fixed2_div_cpt_rev
__kmpc_atomic_fixed2_div_float8
__kmpc_atomic_fixed2_div_fp
__kmpc_atomic_fixed2_div_rev
__kmpc_atomic_fixed2_eqv
__kmpc_atomic_fixed2_eqv_cpt
__kmpc_atomic_fixed2_max
__kmpc_atomic_fixed2_max_cpt
__kmpc_atomic_fixed2_min
__kmpc_atomic_fixed2_min_cpt
__kmpc_atomic_fixed2_mul
__kmpc_atomic_fixed2_mul_cpt
__kmpc_atomic_fixed2_mul_float8
__kmpc_atomic_fixed2_mul_fp
__kmpc_atomic_fixed2_neqv
__kmpc_atomic_fixed2_neqv_cpt
__kmpc_atomic_fixed2_orb
__kmpc_atomic_fixed2_orb_cpt
__kmpc_atomic_fixed2_orl
__kmpc_atomic_fixed2_orl_cpt
__kmpc_atomic_fixed2_rd
__kmpc_atomic_fixed2_shl
__kmpc_atomic_fixed2_shl_cpt
__kmpc_atomic_fixed2_shl_cpt_rev
__kmpc_atomic_fixed2_shl_rev
__kmpc_atomic_fixed2_shr
__kmpc_atomic_fixed2_shr_cpt
__kmpc_atomic_fixed2_shr_cpt_rev
__kmpc_atomic_fixed2_shr_rev
__kmpc_atomic_fixed2_sub
__kmpc_atomic_fixed2_sub_cpt
__kmpc_atomic_fixed2_sub_cpt_rev
__kmpc_atomic_fixed2_sub_fp
__kmpc_atomic_fixed2_sub_rev
__kmpc_atomic_fixed2_swp
__kmpc_atomic_fixed2_wr
__kmpc_atomic_fixed2_xor
__kmpc_atomic_fixed2_xor_cpt
__kmpc_atomic_fixed2u_div
__kmpc_atomic_fixed2u_div_cpt
__kmpc_atomic_fixed2u_div_cpt_rev
__kmpc_atomic_fixed2u_div_fp
__kmpc_atomic_fixed2u_div_rev
__kmpc_atomic_fixed2u_shr
__kmpc_atomic_fixed2u_shr_cpt
__kmpc_atomic_fixed2u_shr_cpt_rev
__kmpc_atomic_fixed2u_shr_rev
__kmpc_atomic_fixed4_add
__kmpc_atomic_fixed4_add_cpt
__kmpc_atomic_fixed4_add_fp
__kmpc_atomic_fixed4_andb
__kmpc_atomic_fixed4_andb_cpt
__kmpc_atomic_fixed4_andl
__kmpc_atomic_fixed4_andl_cpt
__kmpc_atomic_fixed4_div
__kmpc_atomic_fixed4_div_cpt
__kmpc_atomic_fixed4_div_cpt_rev
__kmpc_atomic_fixed4_div_float8
__kmpc_atomic_fixed4_div_fp
__kmpc_atomic_fixed4_div_rev
__kmpc_atomic_fixed4_eqv
__kmpc_atomic_fixed4_eqv_cpt
__kmpc_atomic_fixed4_max
__kmpc_atomic_fixed4_max_cpt
__kmpc_atomic_fixed4_min
__kmpc_atomic_fixed4_min_cpt
__kmpc_atomic_fixed4_mul
__kmpc_atomic_fixed4_mul_cpt
__kmpc_atomic_fixed4_mul_float8
__kmpc_atomic_fixed4_mul_fp
__kmpc_atomic_fixed4_neqv
__kmpc_atomic_fixed4_neqv_cpt
__kmpc_atomic_fixed4_orb
__kmpc_atomic_fixed4_orb_cpt
__kmpc_atomic_fixed4_orl
__kmpc_atomic_fixed4_orl_cpt
__kmpc_atomic_fixed4_rd
__kmpc_atomic_fixed4_shl
__kmpc_atomic_fixed4_shl_cpt

__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_fp
__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_andb_cpt
__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_fp
__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_fp
__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_fp
__kmpc_atomic_float8_div_rev
__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_fp
__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_fp
__kmpc_atomic_float10_div_rev
__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_fp
__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_rev
__kmpc_atomic_float16_max
__kmpc_atomic_float16_max_cpt
__kmpc_atomic_float16_min
__kmpc_atomic_float16_min_cpt
__kmpc_atomic_float16_mul
```

```
__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_atomc_cmplx8_add` is an operation on a `complex<double>` or `complex(kind=8)`, *not* `complex<float>`.

```
__kmpc_atomic_cmplx4_add
__kmpc_atomic_cmplx4_add_cmplx8
__kmpc_atomic_cmplx4_add_cpt
__kmpc_atomic_cmplx4_div
__kmpc_atomic_cmplx4_div_cmplx8
__kmpc_atomic_cmplx4_div_cpt
__kmpc_atomic_cmplx4_div_cpt_rev
__kmpc_atomic_cmplx4_div_rev
__kmpc_atomic_cmplx4_mul
__kmpc_atomic_cmplx4_mul_cmplx8
__kmpc_atomic_cmplx4_mul_cpt
__kmpc_atomic_cmplx4_rd
__kmpc_atomic_cmplx4_sub
__kmpc_atomic_cmplx4_sub_cmplx8
__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_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_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_rd
__kmpc_atomic_cmplx16_sub
__kmpc_atomic_cmplx16_sub_cpt
__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 awaking 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 57 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 203 of file kmp.h.

5.3.2.2 #define KMP_IDENT_AUTOPAR 0x08

Entry point generated by auto-parallelization

Definition at line 201 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 205 of file kmp.h.

5.3.2.4 #define KMP_IDENT_BARRIER_IMPL 0x0040

To Mark implicit barriers.

Definition at line 207 of file kmp.h.

5.3.2.5 #define KMP_IDENT_IMB 0x01

Values for bit flags used in the ident_t to describe the fields.

Use trampoline for internal microtasks

Definition at line 196 of file kmp.h.

5.3.2.6 #define KMP_IDENT_KMPC 0x02

Use c-style ident structure

Definition at line 198 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 180 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

loc	in source location information
flags	in for future use (currently ignored)

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 65 of file kmp_csupport.c.

5.5.2.2 void __kmpc_end (ident_t ∗ *loc*)

Parameters

loc	source location information

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 83 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

Parameters

global_tid	the global thread identity of the thread executing the function.
bound_tid	the local identitiy of the thread executing the function
...	pointers to shared variables accessed by the function.

Definition at line 1307 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

loc	source location information
global_tid	global thread number

Leave a serialized parallel construct.

Definition at line 464 of file kmp_csupport.c.

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

Parameters

loc	source location information
argc	total number of arguments in the ellipsis
microtask	pointer to callback routine consisting of outlined parallel construct
...	pointers to shared variables that aren't global

Do the actual fork and call the microtask in the relevant number of threads.

Definition at line 300 of file kmp_csupport.c.

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

Parameters

loc	source location information
argc	total number of arguments in the ellipsis
microtask	pointer to callback routine consisting of outlined teams construct
...	pointers to shared variables that aren't global

Do the actual fork and call the microtask in the relevant number of threads.

Definition at line 383 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

loc	source location information
global_tid	global thread number
num_teams	number of teams requested for the teams construct
num_threads	number of threads per team requested for the teams construct

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 365 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

loc	source location information
global_tid	global thread number
num_threads	number of threads requested for this parallel construct

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 259 of file kmp_csupport.c.

5.6.3.6 void __kmpc_serialized_parallel (ident_t * *loc*, kmp_int32 *global_tid*)

Parameters

loc	source location information
global_tid	global thread number

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 449 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

loc	Source location information.

Returns

The number of threads in the innermost active parallel construct.

Definition at line 166 of file kmp_csupport.c.

5.7.2.2 kmp_int32 __kmpc_bound_thread_num (**ident_t** ∗ *loc*)

Parameters

loc	Source location information.

Returns

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

Definition at line 154 of file kmp_csupport.c.

5.7.2.3 kmp_int32 __kmpc_global_num_threads (**ident_t** ∗ *loc*)

Parameters

loc	Source location information.

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 140 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 ony 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 cannote be defined in terms of OpenMP thread ids returned by omp_get_thread_num()).

Definition at line 117 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 244 of file kmp_csupport.c.

5.8 Work Sharing

Enumerations

- 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 = 45, 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

- 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_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)
- 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)
- int __kmpc_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)
- int __kmpc_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)
- int __kmpc_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)
- 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)

- 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_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)
- 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)
- 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)

- void __kmpc_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)
- 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_uint32 ∗pupperD, kmp_int32 ∗pstride, kmp_int32 incr, kmp_int32 chunk)
- 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)
- 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)

- 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)
- 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)
- 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_int64 ∗p_st, kmp_int64 incr, kmp_int64 chunk)
- 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)

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

kmp_ord_upper upper bound for ordered values

kmp_distribute_static_chunked distribute static chunked

kmp_distribute_static distribute static unspecialized

kmp_nm_lower lower bound for nomerge values

kmp_nm_static static unspecialized

kmp_nm_guided_chunked guided unspecialized

kmp_nm_auto auto

kmp_nm_ord_static ordered static unspecialized

kmp_nm_ord_auto auto

kmp_nm_upper upper bound for nomerge values

kmp_sch_default default scheduling algorithm

Definition at line 320 of file kmp.h.

5.8.3 Function Documentation

5.8.3.1 void __kmpc_critical (ident_t * *loc,* kmp_int32 *global_tid,* kmp_critical_name * *crit*)

Parameters

loc	source location information.
global_tid	global thread number .
crit	identity of the critical section. This could be a pointer to a lock associated with the critical section, or some other suitably unique value.

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

Definition at line 1093 of file kmp_csupport.c.

5.8.3.2 void __kmpc_dispatch_fini_4 (ident_t * *loc,* kmp_int32 *gtid*)

Parameters

loc	Source code location
gtid	Global thread id

Mark the end of a dynamic loop.

Definition at line 2464 of file kmp_dispatch.cpp.

5.8.3.3 void __kmpc_dispatch_fini_4u (ident_t * *loc,* kmp_int32 *gtid*)

See __kmpc_dispatch_fini_4

Definition at line 2482 of file kmp_dispatch.cpp.

5.8.3.4 void __kmpc_dispatch_fini_8 (ident_t * *loc,* kmp_int32 *gtid*)

See __kmpc_dispatch_fini_4

Definition at line 2473 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 2491 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

loc	Source location
gtid	Global thread id
schedule	Schedule type
lb	Lower bound
ub	Upper bound
st	Step (or increment if you prefer)
chunk	The chunk size to block with

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 2314 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 2325 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 2337 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 2350 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

loc	Source code location
gtid	Global thread id
p_last	Pointer to a flag set to one if this is the last chunk or zero otherwise
p_lb	Pointer to the lower bound for the next chunk of work
p_ub	Pointer to the upper bound for the next chunk of work
p_st	Pointer to the stride for the next chunk of work

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 2421 of file kmp_dispatch.cpp.

5.8.3.11 int __kmpc_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 __kmpc_dispatch_next_4

Definition at line 2431 of file kmp_dispatch.cpp.

5.8.3.12 int __kmpc_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 __kmpc_dispatch_next_4

Definition at line 2441 of file kmp_dispatch.cpp.

5.8.3.13 int __kmpc_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 __kmpc_dispatch_next_4

Definition at line 2451 of file kmp_dispatch.cpp.

5.8.3.14 void __kmpc_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 __kmpc_dispatch_init_4

Difference from __kmpc_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 2369 of file kmp_dispatch.cpp.

5.8.3.15 void __kmpc_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

loc	Source code location
gtid	Global thread id of this thread
schedule	Scheduling type for the parallel loop
plastiter	Pointer to the "last iteration" flag
plower	Pointer to the lower bound
pupper	Pointer to the upper bound of loop chunk
pupperD	Pointer to the upper bound of dist_chunk
pstride	Pointer to the stride for parallel loop
incr	Loop increment
chunk	The chunk size for the parallel loop

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 823 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_uint32 ∗ *pupperD*, kmp_int32 ∗ *pstride*, kmp_int32 *incr*, kmp_int32 *chunk*)

See __kmpc_dist_for_static_init_4

Definition at line 836 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 849 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 862 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

loc	source location information.
global_tid	global thread number .
crit	identity of the critical section. This could be a pointer to a lock associated with the critical section, or some other suitably unique value.

Leave a critical section, releasing any lock that was held during its execution.

Definition at line 1192 of file kmp_csupport.c.

5.8.3.20 void __kmpc_end_master (ident_t ∗ *loc*, kmp_int32 *global_tid*)

Parameters

loc	source location information.
global_tid	global thread number .

Mark the end of a `master` region. This should only be called by the thread that executes the `master` region.

Definition at line 762 of file kmp_csupport.c.

5.8.3.21 void __kmpc_end_ordered (ident_t ∗ *loc*, kmp_int32 *gtid*)

Parameters

loc	source location information.
gtid	global thread number.

End execution of an `ordered` construct.

Definition at line 863 of file kmp_csupport.c.

5.8.3.22 void __kmpc_end_single (ident_t * *loc*, kmp_int32 *global_tid*)

Parameters

loc	source location information
global_tid	global thread number

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 1426 of file kmp_csupport.c.

5.8.3.23 void __kmpc_for_static_fini (ident_t * *loc*, kmp_int32 *global_tid*)

Parameters

loc	Source location
global_tid	Global thread id

Mark the end of a statically scheduled loop.

Definition at line 1452 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

loc	Source code location
gtid	Global thread id of this thread
schedtype	Scheduling type
plastiter	Pointer to the "last iteration" flag
plower	Pointer to the lower bound
pupper	Pointer to the upper bound
pstride	Pointer to the stride
incr	Loop increment
chunk	The chunk size

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 753 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

Definition at line 765 of file kmp_sched.cpp.

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 777 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 789 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 709 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 797 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 1384 of file kmp_csupport.c.

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

loc	Source location
gtid	Global thread id
p_last	pointer to last iteration flag
p_lb	pointer to Lower bound
p_ub	pointer to Upper bound
p_st	Step (or increment if you prefer)
incr	Loop increment
chunk	The chunk size to block with

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 900 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 912 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_int64 * *p_st*, kmp_int64 *incr*, kmp_int64 *chunk*)

See __kmpc_team_static_init_4

Definition at line 924 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 936 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(∗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_critical_name ∗lck)
- kmp_int32 __kmpc_reduce (ident_t ∗loc, kmp_int32 global_tid, kmp_int32 num_vars, size_t reduce_size, void ∗reduce_data, 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

loc	source location information
global_tid	thread id.

Execute a barrier.

Definition at line 672 of file kmp_csupport.c.

5.9.2.2 kmp_int32 __kmpc_barrier_master (ident_t ∗ *loc*, kmp_int32 *global_tid*)

Parameters

loc	source location information
global_tid	thread id.

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 1282 of file kmp_csupport.c.

5.9.2.3 kmp_int32 __kmpc_barrier_master_nowait (ident_t ∗ *loc*, kmp_int32 *global_tid*)

Parameters

loc	source location information
global_tid	thread id.

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 1330 of file kmp_csupport.c.

5.9.2.4 void __kmpc_end_barrier_master (ident_t * *loc*, kmp_int32 *global_tid*)

Parameters

loc	source location information
global_tid	thread id.

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 1312 of file kmp_csupport.c.

5.9.2.5 void __kmpc_end_reduce (ident_t * *loc*, kmp_int32 *global_tid*, kmp_critical_name * *lck*)

Parameters

loc	source location information
global_tid	global thread id.
lck	pointer to the unique lock data structure

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 2729 of file kmp_csupport.c.

5.9.2.6 void __kmpc_end_reduce_nowait (ident_t * *loc*, kmp_int32 *global_tid*, kmp_critical_name * *lck*)

Parameters

loc	source location information
global_tid	global thread id.
lck	pointer to the unique lock data structure

Finish the execution of a reduce nowait.

Definition at line 2584 of file kmp_csupport.c.

5.9.2.7 void __kmpc_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 convention obeyed by the compiler even that may not be necessary).

Definition at line 606 of file kmp_csupport.c.

5.9.2.8 kmp_int32 __kmpc_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

loc	source location information
global_tid	global thread number
num_vars	number of items (variables) to be reduced
reduce_size	size of data in bytes to be reduced
reduce_data	pointer to data to be reduced
reduce_func	callback function providing reduction operation on two operands and returning result of reduction in lhs_data
lck	pointer to the unique lock data structure

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 2642 of file kmp_csupport.c.

5.9.2.9 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 *lhs_data, void *rhs_data) *reduce_func*, kmp_critical_name * *lck*)

Parameters

loc	source location information
global_tid	global thread number
num_vars	number of items (variables) to be reduced
reduce_size	size of data in bytes to be reduced
reduce_data	pointer to data to be reduced
reduce_func	callback function providing reduction operation on two operands and returning result of reduction in lhs_data
lck	pointer to the unique lock data structure

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 2445 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(∗cpy_func)(void ∗, void ∗), kmp_int32 didit)
- void __kmpc_threadprivate_register (ident_t ∗loc, void ∗data, kmpc_ctor ctor, kmpc_cctor cctor, kmpc_dtor dtor)
- 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_vec cctor, kmpc_dtor_vec dtor, size_t vector_length)

- typedef void ∗(∗ kmpc_ctor)(void ∗)
- typedef void(∗ kmpc_dtor)(void ∗)
- typedef void ∗(∗ kmpc_cctor)(void ∗, void ∗)
- typedef void ∗(∗ kmpc_ctor_vec)(void ∗, size_t)
- typedef void(∗ kmpc_dtor_vec)(void ∗, size_t)
- typedef void ∗(∗ kmpc_cctor_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_cctor)(void ∗, void ∗)

Pointer to an alternate constructor. The first argument is the `this` pointer.

Definition at line 1334 of file kmp.h.

5.10.2.2 typedef void∗(∗ kmpc_cctor_vec)(void ∗, void ∗, size_t)

Array constructor. First argument is the `this` pointer Third argument the number of array elements.

Definition at line 1356 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 1323 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 1344 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 1329 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 1350 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 ∗) *cpy_func,* kmp_int32 *didit*)

Parameters

loc	source location information
gtid	global thread number
cpy_size	size of the cpy_data buffer
cpy_data	pointer to data to be copied
cpy_func	helper function to call for copying data
didit	flag variable: 1=single thread; 0=not single thread

__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:

```
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 1663 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

loc	source location information
global_tid	global thread number
data	pointer to data to privatize
size	size of data to privatize
cache	pointer to cache

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

loc	source location information
data	pointer to data being privatized
ctor	pointer to constructor function for data
cctor	pointer to copy constructor function for data
dtor	pointer to destructor function for data

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

loc	source location information
data	pointer to data being privatized
ctor	pointer to constructor function for data
cctor	pointer to copy constructor function for data
dtor	pointer to destructor function for data
vector_length	length of the vector (bytes or elements?) Register vector constructors and destructors for thread private data.

Definition at line 718 of file kmp_threadprivate.c.

5.11 Statistics Gathering from OMPTB

Classes

- class stats_flags_e

    flags to describe the statistic (timers or counter)

Macros

- #define KMP_FOREACH_COUNTER(macro, arg)

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

    Add new timers under KMP_FOREACH_TIMER() 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_RESET_STATS() __kmp_reset_stats()

    resets all stats (counters to 0, timers to 0 elapsed ticks)

5.11.1 Detailed Description

These macros support profiling the libiomp5 library. Use –stats=on when building with build.pl to enable and then use the KMP_∗ macros to profile (through counts or clock ticks) libiomp5 during execution of an OpenMP program.

5.11.2 Environment Variables

This section describes the environment variables relevant to stats-gathering in libiomp5

`KMP_STATS_FILE`

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 654 of file kmp_stats.h.

Referenced by __kmpc_barrier(), __kmpc_critical(), __kmpc_dispatch_init_4(), __kmpc_dispatch_init_4u(), __kmpc_dispatch_init_8(), __kmpc_dispatch_init_8u(), __kmpc_dist_dispatch_init_4(), __kmpc_fork_call(), __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 642 of file kmp_stats.h.

5.11.3.3 #define KMP_FOREACH_COUNTER(*macro,* *arg*)

Value:

```
macro (OMP_PARALLEL, stats_flags_e::onlyInMaster, arg)      \
    macro (OMP_FOR_static, 0, arg)                           \
    macro (OMP_FOR_dynamic, 0, arg)                          \
    macro (OMP_DISTR_FOR_static, 0, arg)                     \
    macro (OMP_DISTR_FOR_dynamic, 0, arg)                    \
    macro (OMP_BARRIER, 0, arg)                              \
    macro (OMP_CRITICAL,0, arg)                              \
    macro (OMP_SINGLE, 0, arg)                               \
    macro (OMP_MASTER, 0, arg)                               \
    macro (OMP_set_lock, 0, arg)                             \
    macro (OMP_test_lock, 0, arg)                            \
    macro (OMP_test_lock_failure, 0, arg)                    \
    macro (REDUCE_wait, 0, arg)                              \
    macro (REDUCE_nowait, 0, arg)                            \
    macro (LAST,0,arg)
```

Add new counters under KMP_FOREACH_COUNTER() macro in kmp_stats.h.

Parameters

macro	a user defined macro that takes three arguments - macro(COUNTER_NAME, flags, arg)
arg	a user defined argument to send to the user defined macro

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 in a KMP_BLOCK_COUNTER(name) 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 84 of file kmp_stats.h.

5.11.3.4 #define KMP_FOREACH_EXPLICIT_TIMER(*macro, arg*)

Value:

```
macro(OMP_serial, 0, arg)                         \
    macro(OMP_start_end, 0, arg)                      \
    macro(USER_icv_copy, 0, arg) \
    macro(USER_launch_thread_loop, stats_flags_e::logEvent, arg) \
    macro(LAST, 0, arg)
```

Add new explicit timers under KMP_FOREACH_EXPLICIT_TIMER() macro.

Parameters

macro	a user defined macro that takes three arguments - macro(TIMER_NAME, flags, arg)
arg	a user defined argument to send to the user defined macro

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 204 of file kmp_stats.h.

5.11.3.5 #define KMP_FOREACH_TIMER(*macro, arg*)

Add new timers under KMP_FOREACH_TIMER() macro in kmp_stats.h.

Parameters

macro	a user defined macro that takes three arguments - macro(TIMER_NAME, flags, arg)
arg	a user defined argument to send to the user defined macro

A timer collects multiple samples of some count in each thread and then finally aggregates over all the threads. The count is normally a time (in ticks), hence the name "timer". (But can be any value, so we use this for "number of arguments passed to fork" as well, or we could collect "loop iteration count" if we wanted to). For timers the threads are not significant, it's the individual observations that count, so the statistics are at that level. Format is "macro(name, flags, arg)"

Definition at line 115 of file kmp_stats.h.

5.11.3.6 #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_STATSTHREAD-`

S=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 699 of file kmp_stats.h.

5.11.3.7 #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 709 of file kmp_stats.h.

5.11.3.8 #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.

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 668 of file kmp_stats.h.

Referenced by __kmpc_fork_call().

5.11.3.9 #define KMP_STOP_EXPLICIT_TIMER(*name*) __kmp_stats_thread_ptr->getExplicitTimer(EXPLICIT_TIMER_##name)->stop(TIMER_##name)

"Stops" 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 value. 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 682 of file kmp_stats.h.

Referenced by __kmpc_fork_call().

5.11.3.10 #define KMP_TIME_BLOCK(*name*) blockTimer __BLOCKTIME__(__kmp_stats_thread_ptr->getTimer(TIMER_##name), TIMER_##name)

Uses 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 629 of file kmp_stats.h.

Referenced by __kmpc_barrier().

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

loc_ref	location of the original task directive
gtid	Global Thread ID of encountering thread
new_task	task thunk allocated by __kmp_omp_task_alloc() for the ”new task”
ndeps	Number of depend items with possible aliasing
dep_list	List of depend items with possible aliasing
ndeps_noalias	Number of depend items with no aliasing
noalias_dep_list	List of depend items with no aliasing

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 412 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

loc_ref	location of the original task directive
gtid	Global Thread ID of encountering thread
ndeps	Number of depend items with possible aliasing
dep_list	List of depend items with possible aliasing
ndeps_noalias	Number of depend items with no aliasing
noalias_dep_list	List of depend items with no aliasing

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

Definition at line 480 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 147 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 162 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 151 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 157 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 171 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 218 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 220 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 227 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 219 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 221 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 226 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 67 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 76 of file kmp_wait_release.h.

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

Returns

the flag_type

Definition at line 84 of file kmp_wait_release.h.

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

Parameters

new_loc	in set loc to point at new_loc

Definition at line 80 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 68 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 69 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

6.4 stats_flags_e Class Reference

flags to describe the statistic (timers or counter)

`#include <kmp_stats.h>`

Static Public Attributes

- static const int onlyInMaster = 1<<0
  statistic is valid only for master
- static const int noUnits = 1<<1
  statistic doesn't need units printed next to it in output

- static const int synthesized = 1<<2

    statistic's value is created atexit time in the __kmp_output_stats function
- static const int notInMaster = 1<<3

    statistic is valid for non-master threads
- static const int logEvent = 1<<4

    statistic can be logged when KMP_STATS_EVENTS is on (valid only for timers)

6.4.1 Detailed Description

flags to describe the statistic (timers or counter)

Definition at line 59 of file kmp_stats.h.

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

- kmp_stats.h

Index