Opari 2

1.0.2 (revision 716)

Generated by Doxygen 1.7.3

Wed Apr 4 2012 10:53:36

Contents

Chapter 1

Opari2

Opari2 is a tool to automatically instrument C, C++ and Fortran source code files in which OpenMP is used. Function calls to a POMP2 API are inserted around OpenMP directives. By implementing this API, detailed measurements regarding the runtime behavior of an OpenMP application can be made. A conforming POMP2 implementation needs to implement all POMP2 functions, see pomp2_lib.h for a list of those.

OpenMP 3.0 introduced tasking to OpenMP. To support this feature the POMP2 adapter needs to do some bookkeeping in regard to specific task IDs. The pomp2_lib.c provided with this package includes the necessary code so it is strongly advised to use it as a basis for writing an adapter to your own tool.

A detailed description of the first Opari version has been published by Mohr et al. in "Design and prototype of a performance tool interface for OpenMP" (Journal of supercomputing, 23, 2002).

1.1 INSTALLATION

Opari2 was developed with Autotools. After downloading and unpacking, change into your build directory and perform the following steps:

1. ./configure

   [--prefix=<*installation directory*>]

   [--with-compiler-suite=<*gcc*|*ibm*|*intel*|*pathscale*|*pgi*|*studio*>]

2. make

3. make install

See the file INSTALL for further information.

1.2 USAGE

To create an instrumented version of an OpenMP application, each file of interest is transformed by the OPARI2 tool. The application is then linked against the POMP2 runtime measurement library and optionally to a special initialization file (see section LINKING (startup initialization only) and SUMMARY for further details).

A call to Opari2 has the following syntax:

```
Usage: opari2 [OPTION] ... infile [outfile]

with following options and parameters:

[--f77|--f90|--c|--c++]  [OPTIONAL] Specifies the programming language
                         of the input source file. This option is only
                         necessary if the automatic language detection
                         based on the input file suffix fails.

[--nosrc]                [OPTIONAL] If specified, OPARI2 does not
                         generate #line constructs, which allow to
                         preserve the original source file and line
                         number information, in the transformation
                         process. This option might be necessary if
                         the OpenMP compiler does not understand #line
                         constructs.  The default is to generate #line
                         constructs.

[--nodecl]               [OPTIONAL] Disables the generation of
                         POMP2_DLISTXXXXX macros. These are used in the
                         parallel directives of the instrumentation to
                         make the region handles shared. By using this
                         option the shared clause is used directly on
                         the parallel directive with the respective
                         region handles.

[--tpd]                  [OPTIONAL] Adds the clause 'copyin(<pomp_tpd>)'
                         to any parallel construct. This allows to
                         pass data from the creating thread to its
                         children. The variable is declared externally
                         in all files, so it needs to be defined by
                         the pomp library.

[--disable=<constructs>] [OPTIONAL] Disable the instrumentation of
                         manually-annotated POMP regions or the
                         more fine-grained OpenMP constructs such as
                         !$OMP ATOMIC. <constructs> is a comma
                         separated list of the constructs for which
                         the instrumentation should be disabled.
                         Accepted tokens are atomic, critical, master,
                         flush, single, ordered or locks (as well as
                         sync to disable all of them) or regions.

[--task=                 Special treatment for the task directive
      abort|warn|remove] abort:  Stop instrumentation with an error
                                 message when encountering a task
                                 directive.
                         warn:   Resume but print a warning.
                         remove: Remove all task directives.

[--untied=               Special treatment for the untied task attribute.
     abort|keep|no-warn] The default beavior is to remove the untied
```

```
                          attribute, thus making all tasks tied, and print
                          out a warning.
                          abort:   Stop instrumentation with an error
                                   message when encountering a task
                                   directive with the untied attribute.
                          keep:    Do not remove the untied attribute.
                          no-warn: Do not print out a warning.

[--tpd-mangling=          [OPTIONAL] If programming languages are mixed
 gnu|intel|sun|pgi|       (C and Fortran), the <pomp_tpd> needs to use
 ibm|cray]                the Fortran mangled name also in C files.
                          This option specifies to use the mangling
                          scheme of the gnu, intel, sun, pgi or ibm
                          compiler. The default is to use the mangling
                          scheme of the compiler used to build opari2.

[--version]               [OPTIONAL] Prints version information.

[--help]                  [OPTIONAL] Prints this help text.

infile                    Input file name.

[outfile]                 [OPTIONAL] Output file name. If not
                          specified, opari2 uses the name
                           infile.mod.suffix if the input file is
                          called infile.suffix.

Report bugs to <scorep-bugs@groups.tu-dresden.de>.
```

If you run Opari2 on the input file `example.c` it will create two files:

- `example.mod.c` is the instrumented version of `example.c`, i.e. it contains the original code plus calls to the POMP2 API referencing handles to the OpenMP regions identified by Opari2.
- `example.c.opari.inc` contains the OpenMP region handle definitions accompanied with all the relevant data needed by the handles. This compile time context (CTC) information is encoded into a string for maximum portability. For each region, the tuple (region_handle, ctc_string) is passed to an initializing function (POMP2_Assign_handle()). All calls to these initializing functions are gathered in a function named POMP2_Init_regions_XXX_YY, where XXX_YY is unique for each compilation unit.

At some point during the runtime of the instrumented application, the region handles need to be initialized using the information stored in the CTC string. This can be done in one of of two ways:

- during *startup* of the measurement/POMP2 system, or
- during *runtime* when a region handle is accessed for the first time.

We *highly* recommend using the first option as it incurs much less runtime overhead than the second one (no locking, no lookup needed). In this case all POMP2_Init_regions_XXX_YY functions introduced by opari2 need to be called. See LINKING (startup initialization only) for further details. For runtime initialization the ctc string as argument to the relevant POMP2 function calls is provided as an argument.

1.3 CTC string decoding

As mentioned above, we pass ctc strings to different POMP2 functions. These functions need to parse the string in order to process the encoded information. With POMP2_Region_info and ctcString2RegionInfo() the opari2 package provides means of doing this, see pomp2_region_info.h.

The CTC string is a string in the format "length∗key=value∗key=value∗[key=value]∗∗, for example:

∗*82*∗*regionType=parallel*∗*sscl=xmpl*.c:61:61∗escl=xmpl.c:66:66∗hasIf=1∗∗

Mandatory keys are:

- *regionType* Type of the region (here parallel)
- *sscl* First line of the region (usually with full path to file)
- *escl* Last line of the region

Optional keys are

- *hasNumThreads* Set if a numThreads clause is used in the OpenMP directive
- *hasIf* Set if an if clause is used
- *hasOrdered* Set if an ordered clause is used
- *hasReduction* Set if a reduction clause is used
- *hasSchedule* Set if a schedule clause is used
- *hasCollapse* Set if a collapse clause is used

The optional values are set to 0 by default, i.e. the presence of the key denotes the presence of the respective clause.

You can use the function ctcString2RegionInfo() to decode CTC strings. It can be found in pomp2_region_info.c and pomp2_region_info.h, installed under <opari-prefix>/share/opari2/devel.

1.4 LINKING (startup initialization only)

For startup initialization all POMP2_Init_regions_XXX_YY functions that can be found in the object files and libraries of the application are called. This is done by creating an additional compilation unit that contains calls to following POMP2 functions:

- POMP2_Init_regions(),
- POMP2_Get_num_regions(), and
- POMP2_Get_opari2_version().

The resulting object file is linked to the application. During startup of the measurement system the only thing to be done is to call POMP2_Init_regions() which then calls all POMP2_Init_regions_XXX_YY functions.

In order to create the additional compilation unit (for example `pomp2_init_file.c`) the following command sequence can be used:

```
% `opari2-config --nm` <objs_and_libs> |            \
  `opari2-config --egrep` -i "pomp2_init_regions" | \
  `opari2-config --egrep` " [TN] " |                \
  `opari2-config --awk-cmd` -f                      \
  `opari2-config --awk-script` > pomp2_init_file.c
```

Here, <objs_and_libs> denotes the entire set of object files and libraries that were instrumented by opari2.

Due to portability reasons `nm`, `egrep` and `awk` are not called directly but via the provided `opari2-config` tool.

1.5 POMP user instrumentation

For manual user instrumentation the following pragmas are provided.

C/C++:

```
#pragma pomp inst init
#pragma pomp inst begin(region_name)
#pragma pomp inst altend(region_name)
#pragma pomp inst end(region_name)
```

Fortran:

```
!$POMP INST INIT
!$POMP INST BEGIN(region_name)
!$POMP INST ALTEND(region_name)
!$POMP INST END(region_name)
```

Users can specify code regions, like functions for example, with `INST BEGIN` and `INST END`. If a region contains several exit points like return/break/exit/... all but the last need to be marked with `INST ALTEND` pragmas. The `INST INIT` pragma should be used for initialization in the beginning of main, if no other initialization method is used. See the EXAMPLE section for an example on how to use user instrumentation.

1.6 EXAMPLE

The directory <prefix>/share/opari2/doc/example contains the following files:

```
example.c
example.f
Makefile
```

The Makefile contains all required information for building the instrumented and uninstrumented binaries. It demonstrates the compilation and linking steps as described above.

Additional examples which illustrate the use of user instrumentation can be found in <prefix>/share/opari2/doc/example_user_instrumentation. The folder contains the following files:

```
example_user_instrumentation.c
example_user_instrumentation.f
Makefile
```

1.7 News

1.7.1 LINK STEP

Opari2 uses a new mechanism to link files. The main advantage is, that no opari.rc file is needed anymore. Libraries can now be preinstrumented and parallel builds are supported. To achieve this, the handles for parallel regions are instrumented using a ctc_string.

1.7.2 POMP2

The POMP2 interface is not compatible with the original POMP interface. All functions of the new API begin with POMP2_. The declaration prototypes can be found in pomp2_lib.h.

1.7.3 POMP2_Parallel_fork

The POMP2_Parallel_fork() call has an additional argument to pass the requested number of threads to the POMP2 library. This allows the library to prepare data structures and allocate memory for the threads before they are created. The value passed to the library is determined as follows:

- If a `num_threads` clause is present, the expression inside this clause is evaluated into a local variable `pomp_num_threads`. This variable is afterwards passed in the call to POMP2_Parallel_fork() and in the num_threads clause itself.
- If no num_threads clause is present, omp_get_max_threads() is used to determine the requested value for the next parallel region. This value is stored in `pomp_num_threads` and passed to the POMP2_Parallel_fork() call.

In Fortran, instead of omp_get_max_threads(), a wrapper function pomp_get_max_threads_XXX_X is used. This function is needed to avoid multiple definitions of omp_get_max_threads() since we do not know whether it is defined in the user code or not. Removing all definitions in the user code would require much more Fortran parsing than is done with opari2, since function definitions cannot easily be distinguished from variable definitions.

1.7.4 pomp_tpd

If it is necessary for the POMP2 library to pass information from the master thread to its children, the option `--tpd` can be used. Opari2 uses the copyin clause to pass a threadprivate variable `pomp_tpd` to the newly spawned threads at the beginning of a parallel region. This is a 64 bit integer variable, since Fortran does not allow pointers. However a pointer can be stored in this variable, passed to child threads with the copyin clause (in C/C++ or Fortran) and later on be cast back to a pointer in the pomp library.

To support mixed programming (C/Fortran) the variable name depends on the name mangling of the Fortran compiler. This means, for GNU, Sun, Intel and PGI C compilers the variable is called pomp_tpd_ and for IBM it is called pomp_tpd in C. In Fortran it is of course always called pomp_tpd. The --tpd-mangling option can be used to change this. The variable is declared extern in all program units, so the pomp library contains the actual variable declaration of pomp_tpd as a 64 bit integer.

1.7.5 Tasking construct

In *OpenMP 3.0* the new tasking construct was introduced. All parts of a program are now implicitly executed as tasks and the user gets the possibility of creating tasks that can be scheduled for asynchronous execution. Furthermore these tasks can be interrupted at certain scheduling points and resumed later on (see the OpenMP API 3.0 for more detailed information).

Opari2 instruments functions POMP2_Task_create_begin and POMP2_Task_create_-end to allow the recording of the task creation time. For the task execution time, the functions POMP2_Task_begin and POMP2_Task_end are instrumented in the code. To correctly record a profile or a trace of a program execution these different instances of tasks need to be differentiated. Since OpenMP does not provide Task ids, the performance measurement system needs to create and maintain own task ids. This cannot be done by code instrumentation as done by *Opari2* alone but requires some administration of task ids during runtime. To allow the measurement system to administrate these ids, additional task id parameters (pomp_old_task/pomp_new_task) were added to all functions belonging to OpenMP constructs which are task scheduling points. With this package there is a "dummy" library, which can be used as an adapter to your measurement system. This library contains all the relevant functionality to keep track of the different instances of tasks and it is highly recommended to use it as a template to implement your own adapter for your measurement system.

For more detailed information on this mechanism see:

"How to Reconcile Event-Based Performance Analysis with Tasking in OpenMP"

by Daniel Lorenz, Bernd Mohr, Christian Rössel, Dirk Schmidl, and Felix Wolf

In: Proc. of 6th Int. Workshop of OpenMP (IWOMP), LNCS, vol. 6132, pp. 109121

DOI: 10.1007/978-3-642-13217-9_9

1.8 SUMMARY

The typical usage of OPARI2 consists of the following steps:

1. Call OPARI2 for each input source file

```
% opari2 file1.f90
...
% opari2 fileN.f90
```

2. Compile all modified output files ∗.mod.∗ using the OpenMP compiler
3. Generate the initialization file

```
% `opari2-config --nm` file1.mod.o ... fileN.mod.o | \
  `opari2-config --egrep` -i "pomp2_init_regions" |  \
  `opari2-config --egrep` " [TD] " |                 \
  `opari2-config --awk-cmd` -f                       \
  `opari2-config --awk-script` > pomp2_init_file.c
```

4. Link the resulting object files against the pomp2 runtime measurement library.

Chapter 2

Data Structure Index

2.1 Data Structures

Here are the data structures with brief descriptions:

Chapter 3

File Index

3.1 File List

Here is a list of all documented files with brief descriptions:

Chapter 4

Data Structure Documentation

4.1 POMP2_Region_info Struct Reference

This struct stores all information on an OpenMP region, like the region type or corresponding source lines. The function ctcString2RegionInfo() can be used to fill this struct with data from a ctcString.

`#include <pomp2_region_info.h>`

Data Fields

Required attributes

- POMP2_Region_type mRegionType
- char * mStartFileName
- unsigned mStartLine1
- unsigned mStartLine2
- char * mEndFileName
- unsigned mEndLine1
- unsigned mEndLine2

Currently not provided by opari

- bool mHasCopyIn
- bool mHasCopyPrivate
- bool mHasIf
- bool mHasFirstPrivate
- bool mHasLastPrivate
- bool mHasNoWait
- bool mHasNumThreads
- bool mHasOrdered
- bool mHasReduction
- bool mHasCollapse
- bool mHasUntied
- POMP2_Schedule_type mScheduleType

- char * mUserGroupName

Attributes for specific region types

- unsigned mNumSections
- char * mCriticalName
- char * mUserRegionName

4.1.1 Detailed Description

This struct stores all information on an OpenMP region, like the region type or corresponding source lines. The function ctcString2RegionInfo() can be used to fill this struct with data from a ctcString.

4.1.2 Field Documentation

4.1.2.1 char* POMP2_Region_info::mCriticalName

name of a named critical region

4.1.2.2 char* POMP2_Region_info::mEndFileName

name of the corresponding source file from the closing pragma

4.1.2.3 unsigned POMP2_Region_info::mEndLine1

line number of the first line from the closing pragma

4.1.2.4 unsigned POMP2_Region_info::mEndLine2

line number of the last line from the closing pragma

4.1.2.5 bool POMP2_Region_info::mHasCollapse

true if a collapse clause is present

4.1.2.6 bool POMP2_Region_info::mHasCopyIn

true if a copyin clause is present

4.1.2.7 bool POMP2_Region_info::mHasCopyPrivate

true if a copyprivate clause is present

4.1.2.8 bool POMP2_Region_info::mHasFirstPrivate

true if a firstprivate clause is present

4.1.2.9 bool POMP2_Region_info::mHasIf

true if an if clause is present

4.1.2.10 bool POMP2_Region_info::mHasLastPrivate

true if a lastprivate clause is present

4.1.2.11 bool POMP2_Region_info::mHasNoWait

true if a nowait clause is present

4.1.2.12 bool POMP2_Region_info::mHasNumThreads

true if a numThreads clause is present

4.1.2.13 bool POMP2_Region_info::mHasOrdered

true if an ordered clause is present

4.1.2.14 bool POMP2_Region_info::mHasReduction

true if a reduction clause is present

4.1.2.15 bool POMP2_Region_info::mHasUntied

true if a untied clause was present, even if the task was changed to tied during instrumentation.

4.1.2.16 unsigned POMP2_Region_info::mNumSections

number of sections

4.1.2.17 POMP2_Region_type POMP2_Region_info::mRegionType

type of the OpenMP region

4.1.2.18 POMP2_Schedule_type POMP2_Region_info::mScheduleType

schedule type in the schedule clause

4.1.2.19 char* POMP2_Region_info::mStartFileName

name of the corresponding source file from the opening pragma

4.1.2.20 unsigned POMP2_Region_info::mStartLine1

line number of the first line from the opening pragma

4.1.2.21 unsigned POMP2_Region_info::mStartLine2

line number of the last line from the opening pragma

4.1.2.22 char* POMP2_Region_info::mUserGroupName

user group name

4.1.2.23 char* POMP2_Region_info::mUserRegionName

name of a user defined region

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

- pomp2_region_info.h

Chapter 5

File Documentation

5.1 pomp2_lib.h File Reference

This file contains the declarations of all POMP2 functions.

Typedefs

- typedef void * POMP2_Region_handle

Functions

- void POMP2_Assign_handle (POMP2_Region_handle *pomp2_handle, const char ctc_string[ ])
- void POMP2_Atomic_enter (POMP2_Region_handle *pomp2_handle, const char ctc_string[ ])
- void POMP2_Atomic_exit (POMP2_Region_handle *pomp2_handle)
- void POMP2_Barrier_enter (POMP2_Region_handle *pomp2_handle, POMP2_-Task_handle *pomp2_old_task, const char ctc_string[ ])
- void POMP2_Barrier_exit (POMP2_Region_handle *pomp2_handle, POMP2_-Task_handle pomp2_old_task)
- void POMP2_Begin (POMP2_Region_handle *pomp2_handle)
- void POMP2_Critical_begin (POMP2_Region_handle *pomp2_handle)
- void POMP2_Critical_end (POMP2_Region_handle *pomp2_handle)
- void POMP2_Critical_enter (POMP2_Region_handle *pomp2_handle, const char ctc_string[ ])
- void POMP2_Critical_exit (POMP2_Region_handle *pomp2_handle)
- void POMP2_Destroy_lock (omp_lock_t *s)
- void POMP2_Destroy_nest_lock (omp_nest_lock_t *s)
- void POMP2_End (POMP2_Region_handle *pomp2_handle)
- void POMP2_Finalize ()

- void POMP2_Flush_enter (POMP2_Region_handle *pomp2_handle, const char ctc_string[])
- void POMP2_Flush_exit (POMP2_Region_handle *pomp2_handle)
- void POMP2_For_enter (POMP2_Region_handle *pomp2_handle, const char ctc_string[])
- void POMP2_For_exit (POMP2_Region_handle *pomp2_handle)
- POMP2_Task_handle POMP2_Get_new_task_handle ()
- void POMP2_Implicit_barrier_enter (POMP2_Region_handle *pomp2_handle, POMP2_Task_handle *pomp2_old_task)
- void POMP2_Implicit_barrier_exit (POMP2_Region_handle *pomp2_handle, POMP2_Task_handle pomp2_old_task)
- void POMP2_Init ()
- void POMP2_Init_lock (omp_lock_t *s)
- void POMP2_Init_nest_lock (omp_nest_lock_t *s)
- int POMP2_Lib_get_max_threads ()
- void POMP2_Master_begin (POMP2_Region_handle *pomp2_handle, const char ctc_string[])
- void POMP2_Master_end (POMP2_Region_handle *pomp2_handle)
- void POMP2_Off ()
- void POMP2_On ()
- void POMP2_Ordered_begin (POMP2_Region_handle *pomp2_handle)
- void POMP2_Ordered_end (POMP2_Region_handle *pomp2_handle)
- void POMP2_Ordered_enter (POMP2_Region_handle *pomp2_handle, const char ctc_string[])
- void POMP2_Ordered_exit (POMP2_Region_handle *pomp2_handle)
- void POMP2_Parallel_begin (POMP2_Region_handle *pomp2_handle)
- void POMP2_Parallel_end (POMP2_Region_handle *pomp2_handle)
- void POMP2_Parallel_fork (POMP2_Region_handle *pomp2_handle, int if_clause, int num_threads, POMP2_Task_handle *pomp2_old_task, const char ctc_string[])
- void POMP2_Parallel_join (POMP2_Region_handle *pomp2_handle, POMP2_Task_handle pomp2_old_task)
- void POMP2_Section_begin (POMP2_Region_handle *pomp2_handle, const char ctc_string[])
- void POMP2_Section_end (POMP2_Region_handle *pomp2_handle)
- void POMP2_Sections_enter (POMP2_Region_handle *pomp2_handle, const char ctc_string[])
- void POMP2_Sections_exit (POMP2_Region_handle *pomp2_handle)
- void POMP2_Set_lock (omp_lock_t *s)
- void POMP2_Set_nest_lock (omp_nest_lock_t *s)
- void POMP2_Single_begin (POMP2_Region_handle *pomp2_handle)
- void POMP2_Single_end (POMP2_Region_handle *pomp2_handle)
- void POMP2_Single_enter (POMP2_Region_handle *pomp2_handle, const char ctc_string[])
- void POMP2_Single_exit (POMP2_Region_handle *pomp2_handle)
- void POMP2_Task_begin (POMP2_Region_handle *pomp2_handle, POMP2_Task_handle pomp2_task)

- void POMP2_Task_create_begin (POMP2_Region_handle *pomp2_handle, POMP2_-Task_handle *pomp2_new_task, POMP2_Task_handle *pomp2_old_task, int pomp2_-if, const char ctc_string[ ])
- void POMP2_Task_create_end (POMP2_Region_handle *pomp2_handle, POMP2_-Task_handle pomp2_old_task)
- void POMP2_Task_end (POMP2_Region_handle *pomp2_handle)
- void POMP2_Taskwait_begin (POMP2_Region_handle *pomp2_handle, POMP2_-Task_handle *pomp2_old_task, const char ctc_string[ ])
- void POMP2_Taskwait_end (POMP2_Region_handle *pomp2_handle, POMP2_-Task_handle pomp2_old_task)
- int POMP2_Test_lock (omp_lock_t *s)
- int POMP2_Test_nest_lock (omp_nest_lock_t *s)
- void POMP2_Unset_lock (omp_lock_t *s)
- void POMP2_Unset_nest_lock (omp_nest_lock_t *s)
- void POMP2_Untied_task_begin (POMP2_Region_handle *pomp2_handle, POMP2_-Task_handle pomp2_task)
- void POMP2_Untied_task_create_begin (POMP2_Region_handle *pomp2_handle, POMP2_Task_handle *pomp2_new_task, POMP2_Task_handle *pomp2_old_-task, int pomp2_if, const char ctc_string[ ])
- void POMP2_Untied_task_create_end (POMP2_Region_handle *pomp2_handle, POMP2_Task_handle pomp2_old_task)
- void POMP2_Untied_task_end (POMP2_Region_handle *pomp2_handle)
- void POMP2_Workshare_enter (POMP2_Region_handle *pomp2_handle, const char ctc_string[ ])
- void POMP2_Workshare_exit (POMP2_Region_handle *pomp2_handle)

Functions generated by the instrumenter

- size_t POMP2_Get_num_regions ()
- void POMP2_Init_regions ()
- const char * POMP2_Get_opari2_version ()

5.1.1 Detailed Description

This file contains the declarations of all POMP2 functions. alpha

Authors

Daniel Lorenz <d.lorenz@fz-juelich.de> Dirk Schmidl <schmidl@rz.rwth-aachen.de> Peter Philippen <p.philippen@fz-juelich.de>

5.1.2 Typedef Documentation

5.1.2.1 typedef void* POMP2_Region_handle

Handles to identify OpenMP regions.

5.1.3 Function Documentation

5.1.3.1 void POMP2_Assign_handle (POMP2_Region_handle * *pomp2_handle*, const char *ctc_string[]*)

Registers a POMP2 region and returns a region handle.

Parameters

pomp2_handle	Returns the handle for the newly registered region.
ctc_string	A string containing the region data.

5.1.3.2 void POMP2_Atomic_enter (POMP2_Region_handle * *pomp2_handle*, const char *ctc_string[]*)

Called before an atomic statement.

Parameters

pomp2_handle	The handle of the started region.
ctc_string	Initialization string. May be ignored if <pomp2_handle> is already initialized.

5.1.3.3 void POMP2_Atomic_exit (POMP2_Region_handle * *pomp2_handle*)

Called after an atomic statement.

Parameters

pomp2_handle	The handle of the ended region.

5.1.3.4 void POMP2_Barrier_enter (POMP2_Region_handle * *pomp2_handle*, POMP2_Task_handle * *pomp2_old_task*, const char *ctc_string[]*)

Called before a barrier.

OpenMP 3.0: Barriers can be used as scheduling points for tasks. When entering a barrier the task id of the currently executing task (*pomp2_current_task*) is saved in *pomp2_old_task*, which is defined inside the instrumented user code.

Parameters

pomp2_handle	The handle of the started region.

pomp2_old_task	Pointer to a "taskprivate" variable where the current task id is stored.
ctc_string	Initialization string. May be ignored if <pomp2_handle> is already initialized.

5.1.3.5 void POMP2_Barrier_exit (POMP2_Region_handle * *pomp2_handle*, POMP2_Task_handle *pomp2_old_task*)

Called after a barrier.

OpenMP 3.0: When a task exits a barrier the variable *pomp2_old_task* (defined in the instrumented user code) holds the id of the task that entered the barrier. The value is stored in the adapter (in *pomp2_current_task*) to be made available for the measurement system for the following regions.

Parameters

pomp2_handle	The handle of the ended region.
pomp2_old_task	"Taskprivate" variable storing the id of the task the barrier is located in.

5.1.3.6 void POMP2_Begin (POMP2_Region_handle * *pomp2_handle*)

Called at the begin of a user defined POMP2 region.

Parameters

pomp2_handle	The handle of the started region.

5.1.3.7 void POMP2_Critical_begin (POMP2_Region_handle * *pomp2_handle*)

Called at the start of a critical region.

Parameters

pomp2_handle	The handle of the started region.

5.1.3.8 void POMP2_Critical_end (POMP2_Region_handle * *pomp2_handle*)

Called at the end of a critical region.

Parameters

pomp2_handle	The handle of the ended region.

5.1.3.9 void POMP2_Critical_enter (POMP2_Region_handle * *pomp2_handle*, const char *ctc_string[]*)

Called before a critical region.

Parameters

pomp2_handle	The handle of the started region.
ctc_string	Initialization string. May be ignored if <pomp2_handle> is already initialized.

5.1.3.10 void POMP2_Critical_exit (POMP2_Region_handle * *pomp2_handle*)

Called after a critical region.

Parameters

pomp2_handle	The handle of the region.

5.1.3.11 void POMP2_Destroy_lock (omp_lock_t * *s*)

Wraps the omp_destroy_lock function.

Parameters

s	The OpenMP lock to destroy.

5.1.3.12 void POMP2_Destroy_nest_lock (omp_nest_lock_t * *s*)

Wraps the omp_destroy_nest_lock function.

Parameters

s	The nested OpenMP lock to destroy.

5.1.3.13 void POMP2_End (POMP2_Region_handle * *pomp2_handle*)

Called at the begin of a user defined POMP2 region.

Parameters

pomp2_handle	The handle of the started region.

5.1.3.14 void POMP2_Finalize ( )

Finalizes the POMP2 adapter. It is inserted at the #pragma pomp inst end.

5.1.3.15 void POMP2_Flush_enter (POMP2_Region_handle * *pomp2_handle*, const char *ctc_string[]*)

Called before an flush.

Parameters

pomp2_handle	The handle of the started region.
ctc_string	Initialization string. May be ignored if <pomp2_handle> is already initialized.

5.1.3.16 void POMP2_Flush_exit (POMP2_Region_handle * *pomp2_handle*)

Called after an flush.

Parameters

pomp2_handle	The handle of the ended region.

5.1.3.17 void POMP2_For_enter (POMP2_Region_handle * *pomp2_handle*, const char *ctc_string[]*)

Called before a for loop.

Parameters

pomp2_handle	The handle of the region.
ctc_string	Initialization string. May be ignored if <pomp2_handle> is already initialized.

5.1.3.18 void POMP2_For_exit (POMP2_Region_handle * *pomp2_handle*)

Called after a for loop.

Parameters

pomp2_-handle	The handle of the region.

5.1.3.19 POMP2_Task_handle POMP2_Get_new_task_handle ( )

Function that returns a new task handle.

Returns

new task handle

5.1.3.20 size_t POMP2_Get_num_regions ( )

Returns the number of instrumented regions.

The instrumenter scans all opari-created include files with nm and greps the POMP2_-INIT_uuid_numRegions() function calls. Here we return the sum of all numRegions.

Returns

number of instrumented regions

5.1.3.21 const char* POMP2_Get_opari2_version ( )

Returns the opari version.

Returns

version string

5.1.3.22 void POMP2_Implicit_barrier_enter (POMP2_Region_handle * *pomp2_handle*, POMP2_Task_handle * *pomp2_old_task*)

Called before an implicit barrier.

OpenMP 3.0: Barriers can be used as scheduling points for tasks. When entering a barrier the task id of the currently executing task (*pomp2_current_task*) is saved in *pomp2_old_task*, which is defined inside the instrumented user code.

Parameters

pomp2_-handle	The handle of the started region.
pomp2_-old_task	Pointer to a "taskprivate" variable where the current task id is stored.

5.1.3.23 void POMP2_Implicit_barrier_exit (POMP2_Region_handle * *pomp2_handle*, POMP2_Task_handle *pomp2_old_task*)

Called after an implicit barrier.

OpenMP 3.0: When a task exits a barrier the variable *pomp2_old_task* (defined in the instrumented user code) holds the id of the task that entered the barrier. The value is stored in the adapter (in *pomp2_current_task*) to be made available for the measurement system for the following regions.

Parameters

pomp2_handle	The handle of the started region.
pomp2_old_task	"Taskprivate" variable storing the id the task the implicit barrier is used in.

5.1.3.24 void POMP2_Init ( )

Initializes the POMP2 adapter. It is inserted at the #pragma pomp inst begin.

5.1.3.25 void POMP2_Init_lock (omp_lock_t * *s*)

Wraps the omp_init_lock function.

Parameters

s	The OpenMP lock to initialize.

5.1.3.26 void POMP2_Init_nest_lock (omp_nest_lock_t * *s*)

Wraps the omp_init_nest_lock function.

Parameters

s	The nested OpenMP lock to initialize.

5.1.3.27 void POMP2_Init_regions ( )

Init all opari-created regions.

The instrumentor scans all opari-created include files with nm and greps the POMP2_INIT_uuid_numRegions() function calls. The instrumentor then defines these functions by calling all grepped functions.

5.1.3.28 int POMP2_Lib_get_max_threads ( )

Wraps the omp_get_max_threads function.

Needed for the instrumentation of parallel regions where the num_threads clause is used with the return value of the omp_get_max_threads function. This can't be used directly because the user may have declared it himself. Double declarations are not allowed.

5.1.3.29 void POMP2_Master_begin (POMP2_Region_handle * *pomp2_handle*, const char *ctc_string[]*)

Called at the start of a master region.

Parameters

pomp2_handle	The handle of the region.
ctc_string	Initialization string. May be ignored if <pomp2_handle> is already initialized.

5.1.3.30 void POMP2_Master_end (POMP2_Region_handle * *pomp2_handle*)

Called at the end of a master region.

Parameters

pomp2_handle	The handle of the ended region.

5.1.3.31 void POMP2_Off ( )

Disables the POMP2 adapter.

5.1.3.32 void POMP2_On ( )

Enables the POMP2 adapter.

5.1.3.33 void POMP2_Ordered_begin (POMP2_Region_handle * *pomp2_handle*)

Called at the start of an ordered region.

Parameters

pomp2_handle	The handle of the region.

5.1.3.34 void POMP2_Ordered_end (POMP2_Region_handle * pomp2_handle)

Called at the end of an ordered region.

Parameters

pomp2_handle	The handle of the region.

5.1.3.35 void POMP2_Ordered_enter (POMP2_Region_handle * pomp2_handle, const char ctc_string[])

Called before an ordered region.

Parameters

pomp2_handle	The handle of the region.
ctc_string	Initialization string. Ignored.

5.1.3.36 void POMP2_Ordered_exit (POMP2_Region_handle * pomp2_handle)

Called after an ordered region.

Parameters

pomp2_handle	The handle of the region.

5.1.3.37 void POMP2_Parallel_begin (POMP2_Region_handle * pomp2_handle)

Called at the start of a parallel region.

OpenMP 3.0: When a new parallel region is entered, each thread taking part in that region is executed as a task. These tasks are assigned a new unique task id which is stored in *pomp2_current_task*.

Parameters

pomp2_handle	The handle of the region.

5.1.3.38 void POMP2_Parallel_end (POMP2_Region_handle * pomp2_handle)

Called at the end of a parallel region.

Parameters

pomp2_handle	The handle of the region.

5.1.3.39 void POMP2_Parallel_fork (POMP2_Region_handle * *pomp2_handle*, int *if_clause*, int *num_threads*, POMP2_Task_handle * *pomp2_old_task*, const char *ctc_string[]*)

Called before a parallel region.

OpenMP 3.0: The task id of the currently executing task (*pomp2_current_task*) is saved in *pomp2_old_task*, which is defined inside the instrumented user code. In each of the threads taking part in the following parallel region a newly defined unique task id is assigned (see POMP2_Parallel_begin)

Parameters

pomp2_handle	The handle of the region.
if_clause	Value of the argument of an if clause (if present).
num_threads	Upper bound for number of child threads.
pomp2_old_task	The task id of the region from which the parallel region was started.
ctc_string	Initialization string. May be ignored if <pomp2_handle> is already initialized.

5.1.3.40 void POMP2_Parallel_join (POMP2_Region_handle * *pomp2_handle*, POMP2_Task_handle *pomp2_old_task*)

Called after a parallel region.

OpenMP 3.0: When a task exits a parallel region the variable *pomp2_old_task* (defined in the instrumented user code) holds the id of the task that entered the region. The value is stored in the adapter (in *pomp2_current_task*) to be made available for the measurement system for the following regions.

Parameters

pomp2_handle	The handle of the region.
pomp2_old_task	The task id of the region inside of which the parallel region was executed.

5.1.3.41 void POMP2_Section_begin (POMP2_Region_handle * *pomp2_handle*, const char *ctc_string[]*)

Called at the start of a section.

Parameters

pomp2_handle	The handle of the region.
ctc_string	Initialization string. May be ignored if <pomp2_handle> is already initialized.

5.1.3.42 void POMP2_Section_end (POMP2_Region_handle * *pomp2_handle*)

Called at the end of a section.

Parameters

pomp2_handle	The handle of the region.

5.1.3.43 void POMP2_Sections_enter (POMP2_Region_handle * *pomp2_handle*, const char *ctc_string[]*)

Called before a set of sections.

Parameters

pomp2_handle	The handle of the region.
ctc_string	Initialization string. May be ignored if <pomp2_handle> is already initialized.

5.1.3.44 void POMP2_Sections_exit (POMP2_Region_handle * *pomp2_handle*)

Called after a set of sections.

Parameters

pomp2_handle	The handle of the region.

5.1.3.45 void POMP2_Set_lock (omp_lock_t * *s*)

Wraps the omp_set_lock function.

Parameters

s	The OpenMP lock to set.

5.1.3.46 void POMP2_Set_nest_lock (omp_nest_lock_t * *s*)

Wraps the omp_set_nest_lock function

Parameters

s	The nested OpenMP lock to set.

5.1.3.47 void POMP2_Single_begin (POMP2_Region_handle * *pomp2_handle*)

Called at the start of a single region.

Parameters

pomp2_handle	The handle of the region.

5.1.3.48 void POMP2_Single_end (POMP2_Region_handle * *pomp2_handle*)

Called at the end of a single region.

Parameters

pomp2_handle	The handle of the region.

5.1.3.49 void POMP2_Single_enter (POMP2_Region_handle * *pomp2_handle*, const char *ctc_string[]*)

Called before a single region.

Parameters

pomp2_handle	The handle of the region.
ctc_string	Initialization string. May be ignored if <pomp2_handle> is already initialized.

5.1.3.50 void POMP2_Single_exit (POMP2_Region_handle * *pomp2_handle*)

Called after a single region.

Parameters

pomp2_handle	The handle of the region.

5.1.3.51 void POMP2_Task_begin (POMP2_Region_handle * *pomp2_handle*, POMP2_Task_handle *pomp2_task*)

OpenMP 3.0: Marks the beginning of the execution of a task.

Parameters

pomp2_handle	The region handle.
pomp2_task	handle of task.

5.1.3.52 void POMP2_Task_create_begin (POMP2_Region_handle * *pomp2_handle*, POMP2_Task_handle * *pomp2_new_task*, POMP2_Task_handle * *pomp2_old_task*, int *pomp2_if*, const char *ctc_string[]*)

OpenMP 3.0: When a task encounters a task construct it creates a new task. The task may be scheduled for later execution or executed immediately. In both cases the pomp-adapter assigns the id of the currently active task to *pomp2_old_task* which is defined in the instrumented user code.

Parameters

pomp2_handle	The handle of the region.
pomp2_old_task	Pointer to the task id in the instrumented user code
pomp2_if	If an if clause is present on the task directive this variable holds the evaluated result of the argument of the if clause. Else it is 1.
ctc_string	The initialization string.

5.1.3.53 void POMP2_Task_create_end (POMP2_Region_handle * *pomp2_handle*, POMP2_Task_handle *pomp2_old_task*)

OpenMP 3.0: After the code region which is executed as a separate task, the task id of the creating task is restored from *pomp2_old_task* (defined in the instrumented user code) and stored in *pomp2_current_task* inside the adapter.

Parameters

pomp2_handle	The region handle.
pomp2_old_task	The task id of the task inside of which the task was created. This is stored inside the instrumented user code.

5.1.3.54 void POMP2_Task_end (POMP2_Region_handle * pomp2_handle)

OpenMP 3.0: Marks the end of the execution of a task.

Parameters

pomp2_handle	The region handle.

5.1.3.55 void POMP2_Taskwait_begin (POMP2_Region_handle * pomp2_handle, POMP2_Task_handle * pomp2_old_task, const char ctc_string[])

Called before a taskwait.

OpenMP 3.0: Taskwait directives can be used as scheduling points for tasks. When entering a taskwait region the task id of the currently executing task (*pomp2_current_task*) is saved in *pomp2_old_task*, which is defined inside the instrumented user code.

Parameters

pomp2_handle	The handle of the started region.
pomp2_old_task	Pointer to a "taskprivate" variable where the current task id is stored.
ctc_string	Initialization string. May be ignored if <pomp2_handle> is already initialized.

5.1.3.56 void POMP2_Taskwait_end (POMP2_Region_handle * pomp2_handle, POMP2_Task_handle pomp2_old_task)

Called after a barrier.

OpenMP 3.0: When a task exits a taskwait region the variable *pomp2_old_task* (defined in the instrumented user code) holds the id of the task that entered the taskwait. The value is stored in the adapter (in *pomp2_current_task*) to be made available for the measurement system for the following regions.

Parameters

pomp2_handle	The handle of the ended region.
pomp2_old_task	"Taskprivate" variable storing the id of the task the barrier is located in.

5.1.3.57 int POMP2_Test_lock (omp_lock_t * s)

Wraps the omp_test_lock function

Parameters

s	the OpenMP lock to test for.

Returns

result of omp_test_lock

5.1.3.58 int POMP2_Test_nest_lock (omp_nest_lock_t * s)

Wraps the omp_test_nest_lock function

Parameters

s	The nested OpenMP lock to test for.

Returns

result of omp_test_nest_lock

5.1.3.59 void POMP2_Unset_lock (omp_lock_t * s)

Wraps the omp_unset_lock function.

Parameters

s	the OpenMP lock to unset.

5.1.3.60 void POMP2_Unset_nest_lock (omp_nest_lock_t * s)

Wraps the omp_unset_nest_lock function

Parameters

s	The nested OpenMP lock to unset.

5.1.3.61 void POMP2_Untied_task_begin (POMP2_Region_handle * pomp2_handle, POMP2_Task_handle pomp2_task)

OpenMp 3.0: Marks the beginning of the execution of an untied task.

Parameters

pomp2_handle	The region handle.
pomp2_task	Handle of this task.

5.1.3.62 void POMP2_Untied_task_create_begin (POMP2_Region_handle * *pomp2_handle*, POMP2_Task_handle * *pomp2_new_task*, POMP2_Task_handle * *pomp2_old_task*, int *pomp2_if*, const char *ctc_string[]*)

OpenMP 3.0: When a task encounters a task construct it creates a new task. If the untied clause is specified the task is executed as an untied task. The task may be scheduled for later execution or executed immediately. In both cases the pomp-adapter assigns the id of the currently active task to *pomp2_old_task* which is defined in the instrumented user code.

Parameters

pomp2_handle	The handle of the region.
pomp2_old_task	Pointer to the task id in the instrumented user code.
pomp2_if	If an if clause is present on the task directive this variable holds the evaluated result of the argument of the if clause. Else it is 1.
ctc_string	The initialization string.

5.1.3.63 void POMP2_Untied_task_create_end (POMP2_Region_handle * *pomp2_handle*, POMP2_Task_handle *pomp2_old_task*)

OpenMP 3.0: After the code region which is executed as a separate untied task, the task id of the creating task is restored from *pomp2_old_task* (defined in the instrumented user code) and stored in *pomp2_current_task* inside the adapter.

Parameters

pomp2_handle	The region handle.
pomp2_old_task	The id of the task from which the untied task was created. This is stored in the instrumented user code.

5.1.3.64 void POMP2_Untied_task_end (POMP2_Region_handle * *pomp2_handle*)

OpenMP 3.0: Marks the end of the execution of a task.

Parameters

pomp2_handle	The region handle.

5.1.3.65 void POMP2_Workshare_enter (POMP2_Region_handle * *pomp2_handle*, const char *ctc_string[]*)

Called before a workshare region.

Parameters

pomp2_handle	The handle of the region.
ctc_string	Initialization string. May be ignored if <pomp2_handle> is already initialized.

5.1.3.66 void POMP2_Workshare_exit (POMP2_Region_handle * *pomp2_handle*)

Called after a workshare region.

Parameters

pomp2_handle	The handle of the region.

5.2 pomp2_region_info.h File Reference

This file contains function declarations and structs which handle informations on OpenMP regions. POMP2_Region_info is used to store these informations. It can be filled with a ctcString by ctcString2RegionInfo().

Data Structures

- struct POMP2_Region_info

  This struct stores all information on an OpenMP region, like the region type or corresponding source lines. The function ctcString2RegionInfo() can be used to fill this struct with data from a ctcString.

Enumerations

- enum POMP2_Region_type
- enum POMP2_Schedule_type

Functions

- void ctcString2RegionInfo (const char ctcString[ ], POMP2_Region_info *regionInfo)
- void freePOMP2RegionInfoMembers (POMP2_Region_info *regionInfo)
- const char * pomp2RegionType2String (POMP2_Region_type regionType)
- const char * pomp2ScheduleType2String (POMP2_Schedule_type scheduleType)

5.2.1 Detailed Description

This file contains function declarations and structs which handle informations on OpenMP regions. POMP2_Region_info is used to store these informations. It can be filled with a ctcString by ctcString2RegionInfo().

Author

Christian Rössel <c.roessel@fz-juelich.de> alpha

Date

Started Fri Mar 20 16:30:45 2009

5.2.2 Enumeration Type Documentation

5.2.2.1 enum POMP2_Region_type

POMP2_Region_type

5.2.2.2 enum POMP2_Schedule_type

type to store the scheduling type of a for worksharing constuct

5.2.3 Function Documentation

5.2.3.1 void ctcString2RegionInfo (const char *ctcString[]*, POMP2_Region_info * *regionInfo*)

ctcString2RegionInfo() fills the POMP2_Region_info object with data read from the ctcString. If the ctcString does not comply with the specification, the program aborts with exit code 1.

Rationale: ctcString2RegionInfo() is used during initialization of the measurement system. If an error occurs, it is better to abort than to struggle with undefined behaviour or *guessing* the meaning of the broken string.

Note

Can be called from multiple threads concurrently, assuming malloc is thread-safe. ctcString2RegionInfo() will assign memory to the members of *regionInfo*. You are supposed to to release this memory by calling freePOMP2RegionInfoMembers().

Parameters

ctcString	A string in the format "length∗key=value∗[key=value]∗". The length field is parsed but not used by this implementation. Possible values for key are listed in ctcTokenMap. The string must at least contain values for the keys `regionType`, `sscl` and `escl`. Possible values for the key `regionType` are listed in regionTypesMap. The format for `sscl` resp. `escl` values is "`filename:lineNo1:lineNo2`".

regionInfo	must be a valid object

Postcondition

At least the required attributes (see POMP2_Region_info) are set.
All other members of *regionInfo* are set to 0 resp. false resp. POMP2_No_schedule.
If `regionType=sections` than POMP2_Region_info::mNumSections has a value > 0.
If `regionType=region` than POMP2_Region_info::mUserRegionName has a value != 0.
If `regionType=critical` than POMP2_Region_info::mCriticalName may have a value != 0.

5.2.3.2 void freePOMP2RegionInfoMembers (POMP2_Region_info * *regionInfo*)

Free the memory of the regionInfo members.

Parameters

regionInfo	The regioninfo to be freed.

5.2.3.3 const char* pomp2RegionType2String (POMP2_Region_type *regionType*)

converts regionType into a string

Parameters

regionType	The regionType to be converted.

Returns

string representation of the region type

5.2.3.4 const char* pomp2ScheduleType2String (POMP2_Schedule_type *scheduleType*)

converts scheduleType into a string

Parameters

schedule-Type	The scheduleType to be converted.

Returns

string representation of the scheduleType

Index