

Opari 2
1.0-beta (revision 492)

Generated by Doxygen 1.7.3

Fri Nov 4 2011 21:02:16

Contents

1	Opari2	1
1.1	INSTALLATION	1
1.2	USAGE	2
1.3	CTC string decoding	3
1.4	LINKING (startup initialization only)	4
1.5	POMP user instrumentation	5
1.6	EXAMPLE	5
1.7	News	6
1.7.1	LINK STEP	6
1.7.2	POMP2	6
1.7.3	POMP2_Parallel_fork	6
1.7.4	pomp_tpd	6
1.7.5	Tasking construct	7
1.8	SUMMARY	7
2	Data Structure Index	9
2.1	Data Structures	9
3	Data Structure Documentation	11
3.1	POMP2_Region_info Struct Reference	11
3.1.1	Detailed Description	12
3.1.2	Field Documentation	12
3.1.2.1	mCriticalName	12
3.1.2.2	mEndFileName	12
3.1.2.3	mEndLine1	12
3.1.2.4	mEndLine2	12
3.1.2.5	mHasCopyIn	12
3.1.2.6	mHasCopyPrivate	12
3.1.2.7	mHasFirstPrivate	12
3.1.2.8	mHasIf	13
3.1.2.9	mHasLastPrivate	13
3.1.2.10	mHasNoWait	13
3.1.2.11	mHasNumThreads	13
3.1.2.12	mHasOrdered	13
3.1.2.13	mHasReduction	13
3.1.2.14	mNumSections	13
3.1.2.15	mRegionType	13
3.1.2.16	mScheduleType	13
3.1.2.17	mStartFileName	13

3.1.2.18	mStartLine1	13
3.1.2.19	mStartLine2	14
3.1.2.20	mUserGroupName	14
3.1.2.21	mUserRegionName	14

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:

<code>[--f77 --f90 --c --c++]</code>	[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.
<code>[--nosrc]</code>	[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.
<code>[--tpd]</code>	[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.
<code>[--disable constructs]</code>	[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 or locks (as well as sync to disable all of them) or regions.
<code>[--task= abort warn remove]</code>	Special treatment for the task directive abort: Stop instrumentation with an error message when encountering a task directive. warn: Resume but print a warning. remove: Remove all task directives.
<code>[--untied= abort keep no-warn]</code>	Special treatment for the untied task attribute. The default behavior 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 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.

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` " T " | \
`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/opari/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/opari/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` " T " | \  
`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:

[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`) [11](#)

Chapter 3

Data Structure Documentation

3.1 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](#)
- POMP2_Schedule_type [mScheduleType](#)
- char * [mUserGroupName](#)

Attributes for specific region types

- unsigned [mNumSections](#)
- char * [mCriticalName](#)
- char * [mUserRegionName](#)

3.1.1 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`.

3.1.2 Field Documentation

3.1.2.1 char* POMP2_Region_info::mCriticalName

name of a named critical region

3.1.2.2 char* POMP2_Region_info::mEndFileName

name of the corresponding source file from the closing pragma

3.1.2.3 unsigned POMP2_Region_info::mEndLine1

line number of the first line from the closing pragma

3.1.2.4 unsigned POMP2_Region_info::mEndLine2

line number of the last line from the closing pragma

3.1.2.5 bool POMP2_Region_info::mHasCopyIn

true if a copyin clause is present

3.1.2.6 bool POMP2_Region_info::mHasCopyPrivate

true if a copyprivate clause is present

3.1.2.7 bool POMP2_Region_info::mHasFirstPrivate

true if a firstprivate clause is present

3.1.2.8 bool POMP2_Region_info::mHasIf

true if an if clause is present

3.1.2.9 bool POMP2_Region_info::mHasLastPrivate

true if a lastprivate clause is present

3.1.2.10 bool POMP2_Region_info::mHasNoWait

true if a nowait clause is present

3.1.2.11 bool POMP2_Region_info::mHasNumThreads

true if a numThreads clause is present

3.1.2.12 bool POMP2_Region_info::mHasOrdered

true if an ordered clause is present

3.1.2.13 bool POMP2_Region_info::mHasReduction

true if a reduction clause is present

3.1.2.14 unsigned POMP2_Region_info::mNumSections

number of sections

3.1.2.15 POMP2_Region_type POMP2_Region_info::mRegionType

type of the OpenMP region

3.1.2.16 POMP2_Schedule_type POMP2_Region_info::mScheduleType

schedule type in the schedule clause

3.1.2.17 char* POMP2_Region_info::mStartFileName

name of the corresponding source file from the opening pragma

3.1.2.18 unsigned POMP2_Region_info::mStartLine1

line number of the first line from the opening pragma

3.1.2.19 unsigned POMP2_Region_info::mStartLine2

line number of the last line from the opening pragma

3.1.2.20 char* POMP2_Region_info::mUserGroupName

user group name

3.1.2.21 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

Index

mCriticalName
 POMP2_Region_info, [12](#)
mEndFileName
 POMP2_Region_info, [12](#)
mEndLine1
 POMP2_Region_info, [12](#)
mEndLine2
 POMP2_Region_info, [12](#)
mHasCopyIn
 POMP2_Region_info, [12](#)
mHasCopyPrivate
 POMP2_Region_info, [12](#)
mHasFirstPrivate
 POMP2_Region_info, [12](#)
mHasIf
 POMP2_Region_info, [12](#)
mHasLastPrivate
 POMP2_Region_info, [13](#)
mHasNoWait
 POMP2_Region_info, [13](#)
mHasNumThreads
 POMP2_Region_info, [13](#)
mHasOrdered
 POMP2_Region_info, [13](#)
mHasReduction
 POMP2_Region_info, [13](#)
mNumSections
 POMP2_Region_info, [13](#)
mRegionType
 POMP2_Region_info, [13](#)
mScheduleType
 POMP2_Region_info, [13](#)
mStartFileName
 POMP2_Region_info, [13](#)
mStartLine1
 POMP2_Region_info, [13](#)
mStartLine2
 POMP2_Region_info, [13](#)
mUserGroupName
 POMP2_Region_info, [14](#)
mUserRegionName
 POMP2_Region_info, [14](#)

POMP2_Region_info, [14](#)
POMP2_Region_info, [11](#)
 mCriticalName, [12](#)
 mEndFileName, [12](#)
 mEndLine1, [12](#)
 mEndLine2, [12](#)
 mHasCopyIn, [12](#)
 mHasCopyPrivate, [12](#)
 mHasFirstPrivate, [12](#)
 mHasIf, [12](#)
 mHasLastPrivate, [13](#)
 mHasNoWait, [13](#)
 mHasNumThreads, [13](#)
 mHasOrdered, [13](#)
 mHasReduction, [13](#)
 mNumSections, [13](#)
 mRegionType, [13](#)
 mScheduleType, [13](#)
 mStartFileName, [13](#)
 mStartLine1, [13](#)
 mStartLine2, [13](#)
 mUserGroupName, [14](#)
 mUserRegionName, [14](#)