Ovaltine Users' Guide

1. Introduction

Ovaltine is a portable performance analysis tool that we have developed in the Centre for Novel Computing (CNC) over the past two years. It is used to analyse OpenMP codes, specifically those written in Fortran77. It instruments a copy of the source code, which is then executed to produce data. Ovaltine uses this data to determine various overheads for the given parallel implementation with respect to the given sequential implementation, displaying the results in a simple table.

Overheads analysis is a methodology for determining where and how a parallel implementation is not achieving the expected performance. See [1] and [2] for a more detailed discussion on overheads analysis and its use, and [3] for more on Ovaltine's implementation of the methodology. Currently Ovaltine supports experiments to determine the following overheads:

- unparallelized overhead
- load imbalance
- synchronization (OpenMP BARRIER only)

More experiments, and the ability to chose which to perform, will be added in future releases. The user can either perform overhead analysis on the whole code or on a region of the program.

The user provides Ovaltine with a sequential and an OpenMP Fortran77 code which are then instrumented. The instrumented code can be run on any number of target machines on various numbers of processors. Ovaltine then analyzes the data collected from the instrumented runs and produces a table of overheads for each run. Ovaltine analyses a snapshot in the development of OpenMP programs. It does not suggest how to improve a given OpenMP code but rather points out the limitations of the current implementation using a table of overheads.

The code is currently a beta version and as such has a few limiting requirements and a limited subset of overheads. Before explaining how to use Ovaltine, we list the prerequisites.

Ovaltine was funded by the EPSRC.

2. Requirements

Ovaltine requires that each program unit (e.g. main routine, subroutine or function) is in a separate file (e.g. as obtained by using `fsplit`) and that the name of a file is the same as the program unit name. The source Fortran files must have a "`.f`" suffix (i.e. preprocessing or `.f77`/`.F` files are not allowed). The user has to provide one file containing a list of all the files (including pathnames relative to the current working directory) used for the sequential version, and another for the OpenMP version. As long as the main program is listed first, all other subprogram units can be listed in any order, one per line with no preceding white space. The "`.f`" suffix should be excluded from each file name.

For example, if the sequential program `main1` has calls to subroutines `sub1` and `sub2`, then a suitable input file, assuming all files reside in the current directory, would contain

```
main1
sub2
sub1
```

Ovaltine makes use of external software to help produce its internal representation of the source code. This version of Ovaltine requires the Polaris scanner (`p-scanner3`) to be on the users `PATH`, and requires Java2 (Java 1.3) or higher since some parts of Ovaltine require Java Swing.

Where possible the user should also have the Portable API (PAPI) installed for portable access to hardware counters and high resolution timers. If PAPI is not available, the user has to provide a routine for accessing a high resolution timer. More details are in the Section 4.

2.1. Restrictions

This initial release of Ovaltine has several, mostly minor, restrictions. The majority of these cases will not be seen in well written standard Fortran77 programs. Future releases of Ovaltine will enhance the tool to overcome these restrictions.

Fortran

- All files should adhere to the Fortran77 standard, in particular everything is uppercase and comments must have a "`C`" (i.e. a capital C) or "`*`" in the first column. (OV`convert` , from the download site, is an example script for converting "!" in the first column to "C" and making the program uppercase throughout.) However variables may have more than 6 characters and the use of `ENDDO` is encouraged.
- The main program unit must begin with `PROGRAM` *`name`*, where *`name`* is any valid program name and the file will be called *`name`*`.f`
- Fortran keywords can only be used as Fortran control statements – Ovaltine will incorrectly handle code such as:

```
CHARACTER FORMAT, GOTO
…
FORMAT = GOTO
```

- The following constructs are not currently handled by Ovaltine:
  - functions (allowed but not analysed)
  - function statements
  - `DO WHILE` statements
  - assigned `GOTO`, such as

```
ASSIGN 100 TO IFLAG
...
GOTO IFLAG
```

  - arithmetic `IF` (e.g. `IF(expression) label1, label2, label3`)
  - `RETURN` *`int`* statements and the corresponding `CALL SUBNAME(…, *n, …)`
  - loop termination statements cannot be continued, have comments embedded nor consist of logical IF statements
  - nested `DO` loops with the same termination statement
  - `STOP` codes
  - unlabelled `CONTINUE`
- `GOTO` should be used instead of "`GO TO`" in logical IF or any labelled statement
- `STOP` cannot have white space (e.g. "`ST OP`") not be continued over different lines
- using a vendor's preprocessor directives (e.g. `CDIR$`) may cause problems, particularly if they are the first statements after the program declarations
- no more than 25 program units; each program unit to have no more than 100 "nodes" (e.g. each basic block or flow control statement is counted as a node) and no more than 100 outer iterations of inner parallel regions (see Section 7)
- Tabs, as opposed to white space, may cause problems under certain conditions

OpenMP

- Only OpenMP version 1.1 (November 1999) is supported.
- OpenMP directives must be fixed source form (i.e. starting in column one), are limited to the `C$OMP` sentinel and must be in uppercase. Furthermore, white space is required between directives and clauses. For example,

```
C$OMP PARALLELDOSHARED(A,B,C)
```

  is not supported by Ovaltine
- Clauses should be separated by white space not a comma.
- No white space is allowed "inside" clauses. Thus, for example, the following is not supported:

```
C$OMP END DO NO WAIT
```

- White space is not allowed between a clause and its arguments (e.g. `PRIVATE (X,Y,Z)` is not allowed)
- Conditional compilation (`C$`) is not fully supported. Nor is the C preprocessor macro, `_OPENMP`, supported.
- Only data scope attribute clauses can be on a continuation line. OpenMP keywords cannot be split with either white space nor over continuation lines. The following examples are thus illegal:

```
C$OMP PARALLEL DO DEF
C$OMP+AULT(NONE)

C$OMP PAR ALL EL SHA RED(A,B,C) PRI VA TE(I,J,K)
```

- Compound parallel work-sharing directives must have both keywords on the initial line of the directive. For example,

```
C$OMP PARALLEL DO DEFAULT(NONE)
C$OMP+SHARED(A,B,C) PRIVATE(I,J,K)
```

  is valid, whereas

```
C$OMP PARALLEL
C$OMP+DO DEFAULT(NONE) SHARED(A,B,C) PRIVATE(I,J,K)
```

  is not.
- `PARALLEL DO` and `PARALLEL SECTIONS` can not have repetition of `ORDERED`, `SCHEDULE()` or `LASTPRIVATE()`
- Nested parallelism is not supported by Ovaltine

Other

- Maximum number of threads you can run the instrumented codes on is 16. Limitations on the input codes are currently that you can have no more than 30 program units (per sequential or per parallel implementation) with no more than 250 nodes per program unit. (See Section 7 for how to change this)
- If you have calls to `OMP BARRIER` you will need to provide data on the costs of a barrier call on your system. For example, use EPCC's OpenMP microbenchmarks (`http://www.epcc.ed.ac.uk/research/openmpbench`) to measure these. The file should be of the form

```
OMP BARRIER;1;0.725E-06;+0.060E-06;-0.060E-06;
OMP BARRIER;2;2.509E-06;+0.111E-06;-0.111E-06;
OMP BARRIER;4;4.102E-06;+0.326E-06;-0.326E-06;
OMP BARRIER;6;5.522E-06;+0.628E-06;-0.628E-06;
OMP BARRIER;8;6.787E-06;+3.224E-06;-3.224E-06;
```

  where column 1 is the synchronization construct, column 2 is the number of processors and column 3 is the cost in seconds. The file should be called `barrier.costs` and be saved in a subdirectory `machine_values` of the directory from which the Analysis is performed (see Section 6).
- Where the source code for a subprogram unit is unavailable (e.g. to library calls) no analysis will occur. If such subprogram units contain OpenMP directives the output from Ovaltine may be misleading.

3. Defining a Region

Ovaltine offers the user the opportunity to determine the overheads for a particular region of code. (If this region is not defined, the whole code is considered.) The start of the "OV region" is set by including the following OV directive at the appropriate point

```
C$OV startRegion
```

The end of the OV region is set by including

```
C$OV finishRegion
```

at the appropriate point. Note that OV directives only have a single space between the `C$OV` and the directive, and case is significant. The following conditions must hold for a valid OV region:

(i) The `startRegion` and `finishRegion` directives must appear in the same program unit

(ii) There can only by one `startRegion` and only one `finishRegion` directive in the program unit

(iii) The `startRegion` directive must precede the `finishRegion` directive

(iv) The region directives must occur after any declarations and before any `RETURN` or `END` statements

(v) If a subprogram unit is called from with the OV region it should not also be called from outside that region.

(vi) An OV region should have a single entry and a single exit point – no checking (e.g. for `GOTO`, `RETURN` or `STOP` within the region) is performed to test this. (NB We allow STOP within an OV region and try to handle this as well as we can but it is possible some data will be lost and a full analysis not possible.)

Furthermore a OV region cannot be inside an OpenMP parallel region and there can be a maximum of one OV region for the whole code. Future versions will allow multiple regions.

It is up to the user to ensure that comparative regions are defined in both the sequential and the parallel codes.

4. Installing Ovaltine

There are several steps to undertake before using Ovaltine. These are:

- download the Ovaltine files
- install the Ovaltine run time library; this will include installation of PAPI or the provision of a user routine to access a high resolution timer
- installation of the Polaris scanner

4.1. Downloading Ovaltine

The required files can be downloaded by anonymous ftp from `ftp.cs.man.ac.uk` in the directory `pub/cnc/ovaltine`. The files are

`OVinstrument.jar`	Java classes for instrumenting source code
`OVanalysis.jar`	Java classes for analysing data from running the instrumented codes
`OVroutines.tar.gz`	Compressed tar file of required Ovaltine run time library files and `Makefile` template
`OVconvert`	ksh script for converting "!" in first column to "C", converting all OpenMP directives to use the sentinel `C$OMP` and for making the code upper case throughout. Requires file name as parameter and outputs converted file with the same file name in directory `CONVERTED` in the current directory

Download all these files into a (new) directory which we henceforth refer to as `$OVDIR`

4.2. Installing the Ovaltine Run Time Library

The RTL contains routines relating to the timers and counters used by the Ovaltine instrumentation and routines for accessing a high resolution timer. Where possible the latter should be a call to PAPI.

1. Change directory to `$OVDIR` and extract the files using, for example,
   `gzcat OVroutines.tar.gz | tar -xvf -`
2. You should then have three new files: `OV_routines.F`, `ovtimerread.F` and `Makefile-template`.

4.3. Installing PAPI

Download PAPI from `http://icl.cs.utk.edu/projects/papi/software` and install as per their instructions. (NB: The PAPI source code for the SGI machines has a bug. In routine `_papi_hwd_get_real_usec` there is an implicit casting from int*long to long long:

```
retval = (t.tv_sec * 1000000) + t.tv_nsec / 1000;
```

Thus for various values of `t.tv_sec` negative numbers are returned giving nonsensical values for elapsed wall clock time. This is most noticeable if the code being timed takes over 2**31 seconds (c36mins). The following replacement is recommended:

```
retval = ( (long long) t.tv_sec * (long long) 1000000) + ( (long long) t.tv_nsec /
(long long) 1000);
```

We shall refer to the directory `$INCLUDEPAPI` as that containing PAPI's include files, and `$PAPILIB` as that where the PAPI libraries reside.

If you do not wish to use PAPI you must provide calls to a high resolution timer that returns the number of seconds since some arbitrary point and amend `$OVALTINE/ovtimerread.F` accordingly.

4.4. Installing Polaris Scanner (p-scanner3)

1. Please refer to the instructions provided by email for obtaining the `p-scanner3` binary for your platform.
2. Ensure `p-scanner3` is in your `PATH`

4.5. Configuring your Makefile

1. Copy the `Makefile-template` file to a new `Makefile` file in the directory of your source code, or other appropriate destination.
2. Amend the flags in the `Makefile` for your system (e.g. for the Fortran77 compiler and optimisations)
3. Ensure that OVDIR points to the `$OVDIR` directory you created earlier
4. If you do not have PAPI installed, ensure the PAPI flag is set to null and provide your own routine in `ovtimerread.F`, otherwise set `$PAPILIB` and `$INCLUDEPAPI` accordingly

Assuming you have PAPI installed your `Makefile` will look similar to the below, where the directories for PAPI and OVALTINE will reflect your local installation:

```
# Makefile template

#  - insert directory containing Ovaltine run time library files
OVDIR = ~/Ovaltine

# leave blank if PAPI not installed
PAPI=-DPAPI
# - insert relevant directory for PAPI include file
INCLUDEPAPI=-I/software/cnc/include
# - insert PAPI library directory and relevant libXXX names
PAPILIB = -L/software/cnc/lib64 -lpapi

# compiler flags - change for your system
F77 = f77
FFLAGS  = -mp
LFLAGS = -mp

# - insert your object file names
OBJS = MAIN.o SUB1.o SUB2.o

OVOBJS = OV_routines.o  ovtimerread.o

a.out:  ${OBJS}
        ${F77} ${LFLAGS} -o $@  ${OBJS}

instrumented.out:       ${OBJS} ${OVOBJS}
        ${F77} ${LFLAGS} -o $@  ${OBJS} ${OVOBJS} ${PAPILIB}

# OV files
OV_routines.o:  OVDECLS.f
        ${F77} -c ${FFLAGS} ${INCLUDEPAPI} -cpp ${PAPI} -I. ${OVDIR}/OV_routines.F

ovtimerread.o:  OVDECLS.f
        ${F77} -c ${FFLAGS} ${INCLUDEPAPI} -cpp ${PAPI} ${OVDIR}/ovtimerread.F
```

To build the original code you enter `make a.out` from the directory with the original source files, and to build the instrumented code enter `make instrumented.out` from the directory with the instrumented code. Note that you will need to either copy the Makefile to each directory or use the `-f` flag followed by the absolute/relative Makefile filename.

5. Using Ovaltine

Assuming all the above requirements are met, we now illustrate how to use Ovaltine. The instrumentation stage is initiated by entering

```
java -jar $OVDIR/OVInstrument
```

You will be prompted to supply the name of the sequential list file and then the OpenMP list file. You can either give the names of these files as an absolute filename or relative to the current working directory. Ovaltine will then take each code and convert it into a set of connected abstract syntax trees in its internal representation. If this is successful, you will be asked whether or not you wish to display the tree (either the contents of every node of the source programs, or whether you wish to have a graphical display of the trees). Ovaltine will then instrument the OV region with a set of overhead analysis experiments. Finally, you will be prompted for a destination directory for each of the instrumented sequential and instrumented parallel codes (again relative to the current working directory) – it is recommended you use a different directory for each, and not the same directories as the original source files. The instrumented codes will be written to these directories, with the filename prefix in capitals. The default is not to overwrite files so if you run Ovaltine several times for the same input codes you should delete all `.f` files in the destination directories before running Ovaltine again.

The user should then compile and link the instrumented codes with the Ovaltine library files including the Ovaltine-produced `OVDECLS.f` file which is `INCLUDE`d in each instrumented file and the Ovaltine library files. It is recommended that you use the `Makefile` for this stage in which case you can enter `make instrumented.out`. Otherwise your compilation, from each instrumented directory, will be of the form:

```
f77 -$OpenMPflag instrumentedFiles.f -I$INCLUDEPAPI -I. -cpp -DPAPI \
$OVDIR/OV_routines.F -cpp -DPAPI $OVDIR/ovtimerread.F -L$PAPILIB -lpapi
```

where

`$OpenMPflag`	is the required flag for OpenMP programs (e.g. –mp using MIPSpro compilers on an SGI Origin)
`instrumentedFiles.f`	is the collection of instrumented files
`$OVDIR`	is the directory containing the OV run time library files (see above)
`$INCLUDEPAPI`	is the directory of the PAPI include files (see above)
`$PAPILIB`	is the directory of the PAPI library/archive files (see above)

(NB The OpenMPflag is required in the compilation of the sequential implementation since there are some OpenMP directives in the Ovaltine libraries.)

The executables can then be run on the required number of processors, taking suitable measures to ensure that the same number of threads are used for every parallel region and that timings are reproducible. At the end of each run a file `fort.71`, containing performance data, is produced. This should be renamed to something more suitable (and must have the suffix `.results` before Ovaltine can perform its analysis).

6. Analysis

To obtain the overhead analysis, rename the relevant `fort.71` file with any suitable prefix (in any suitable directory) but with the suffix "`.results`". Now run the analysis stage of Ovaltine by entering

```
java -jar $OVDIR/OVanalysis
```

As before, enter the "list file" pointing to the *original, uninstrumented* codes. You will then be prompted to enter the names (minus the `.results` suffix) of the data files. Ovaltine will re-construct the abstract syntax tree for each code, and use the performance data to produce a table of overheads, similar to:

```
OVALTINE:           OVERHEADS          ANALYSIS
 ------------------------------------------------------------

serial numThreads=0.0
parallel numThreads=4.0

(p-1)/p = 0.75
alpha = 0.9995521184795916 (fraction of parallelized code)


                 Times/seconds     (%age of actual parallel time)

serial time:       6.028E-01
----------------
parallel time:     6.164E-01
----------------
ideal par time:    1.507E-01         24.448%
non parallel   OH: 2.025E-04         0.033%
load imbal     OH: 2.819E-01         45.726%
sync (barrier) OH: 0E00         0%
unaccounted    OH: 1.837E-01         29.793%
----------------
Efficiency: 24.448%
```

where `alpha` is the fraction of parallelized code (from Amdahl's Law). Depending upon the structure of your code you may also see output relating to the load imbalance per iteration of a DO loop. These are summed to give the total load imbalance for the OV region in question.

6.1. Understanding the Output

To determine the main overhead for your parallel implementation, look at the percentages for the categorised overheads (i.e. those with `OH` in the table). For the above example, the high value for the load imbalance overhead indicates that there are areas of the code where the work is not evenly spread between the threads. The full output from Ovaltine will give the load imbalance between each relevant pair of synchronization points and should be studied in detail to see whether there are particular `DO` loops, for example, that need further attention. A high figure in the `non parallel OH` indicates significant sequential areas of the code. A high figure for the `sync OH` results from copious calls to `OMP BARRIER`. The `unaccounted OH` relates to other differences between the sequential and parallel implements, such as different memory access patterns, overheads due to the parallelization (e.g. the cost of setting up `OMP PARALLEL` regions) and unquantifiable changes in the run time environment.

7. Bugs, Handy Hints and Known Issues

1. If you have any problems, we'd like to know. Send a copy of your source code (preferably the smallest example that illustrates your problem), together with the `Makefile`/compilation options and system information to `ovaltine@cs.man.ac.uk`. There is an email list for users to get the latest news and bug fixes. To subscribe send an email to `ovaltine-request@lists.man.ac.uk` with the word `subscribe` as the only text.
2. Ensure you set `OMP_NUM_THREADS` to one before running the instrumented serial code.
3. If during compilation of the instrumented codes you get an error regarding a subscript out of range for one or more of the OV_ arrays, then you have more program units or nodes that the default maximum. You can increase these by manually increasing the values of `OV_MAX_THREADS`, `OV_MAX_PUS` or `OV_MAX_NODES` in the `OVDECLS.f` file in all directories containing instrumented code; do *not* rerun Ovaltine since this will attempt tp overwrite your changes in `OVDECLS.f`. Note that increasing such values will result in a larger memory footprint and greater intrusion by Ovaltine. (The next version of Ovaltine will compute these maximum values automatically.)
4. If you display the abstract syntax trees via a GUI, Ovaltine will not terminate until all the displayed frames are closed.
5. If your program uses files not in the input `.list` files (e.g. header files or codes you don't want instrumented) then you will need to manually copy these to the instrumented directories.
6. When you run the instrumented code, if you notice repetition of the "Welcome to Ovaltine" message a likely cause is that the first file name in the input `.list` file is not the program entry point.

8. Future Work

Some or all of: choice of OH expts; more than a single region that is instrumented at a time; automatic calculation of all lower/upper bounds for all arrays used by Ovaltine instrumentations.

9. References

[1] J.M. Bull. *A Hierarchical Classification of Overheads in Parallel Programs*, Proceedings of First IFIP TC10 International Workshop on Software Engineering for Parallel and Distributed Systems, I. Jelly, I. Gorton and P. Croll (eds), Chapman Hall, pp208-219, Mach 1996.

[2] G.D. Riley, J.M. Bull and J.R. Gurd. *Performance Improvement Through Overhead Analysis: A Case Study in Molecular Dynamics*, Proc. 11th ACM International Conference on Supercomputing, ACM Press, pp36-43, July 1997.

[3] M.K. Bane and G.D. Riley, *Automatic Overheads Profiler for OpenMP Codes*, Proceedings of 2nd European Workshop on OpenMP (EWOMP), http://www.epcc.ed.ac.uk/ewomp2000/proceedings.html

© Michael Bane, Centre for Novel Computing, University of Manchester (2002)