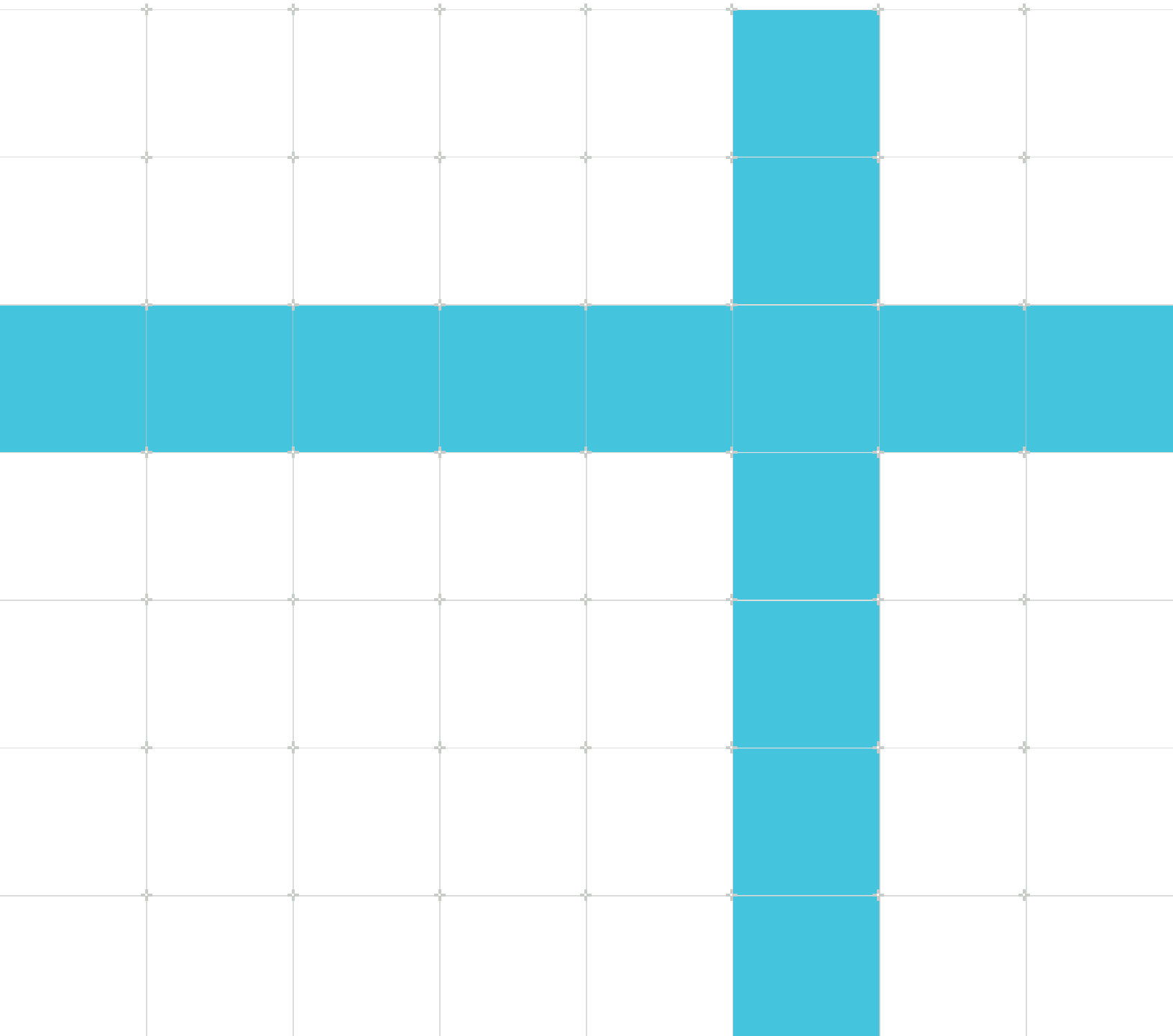
Arm Compiler for Linux OpenMP settings

Version 1.0

Issue 03
102580_0100_03_en



Arm Compiler for Linux OpenMP settings

Release information

Document history

Issue	Date	Confidentiality	Change
0100-03	4 April 2024	Non-Confidential	Image update
0100-02	4 March 2022	Non-Confidential	Initial release

Proprietary Notice

makes no representation with respect to, and has undertaken no analysis to identify or understand the scope and content of, any patents, copyrights, trade secrets, trademarks, or other rights.

TO THE EXTENT NOT PROHIBITED BY LAW, IN NO EVENT WILL ARM BE LIABLE FOR ANY DAMAGES, INCLUDING WITHOUT LIMITATION ANY DIRECT, INDIRECT, SPECIAL, INCIDENTAL, PUNITIVE, OR CONSEQUENTIAL DAMAGES, HOWEVER CAUSED AND REGARDLESS OF THE THEORY OF LIABILITY, ARISING OUT OF ANY USE OF THIS DOCUMENT, EVEN IF ARM HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

Reference by Arm to any third party's products or services within this document is not an express or implied approval or endorsement of the use thereof.

This document consists solely of commercial items. You shall be responsible for ensuring that any permitted use, duplication, or disclosure of this document complies fully with any relevant export laws and regulations to assure that this document or any portion thereof is not exported, directly or indirectly, in violation of such export laws. Use of the word "partner" in reference to Arm's customers is not intended to create or refer to any partnership relationship with any other company. Arm may make changes to this document at any time and without notice.

This document may be translated into other languages for convenience, and you agree that if there is any conflict between the English version of this document and any translation, the terms of the English version of this document shall prevail.

The validity, construction and performance of this notice shall be governed by English Law.

Arm Limited. Company 02557590 registered in England.

110 Fulbourn Road, Cambridge, England CB1 9NJ.

PRE-1121-V1.0

Confidentiality Status

This document is Non-Confidential. The right to use, copy and disclose this document may be subject to license restrictions in accordance with the terms of the agreement entered into by Arm and the party that Arm delivered this document to.

Unrestricted Access is an Arm internal classification.

Product Status

The information in this document is Final, that is for a developed product.

Feedback

Arm welcomes feedback on this product and its documentation. To provide feedback on the product, create a ticket on https://support.developer.arm.com

To provide feedback on the document, fill the following survey: https://developer.arm.com/documentation-feedback-survey.

Inclusive language commitment

Arm values inclusive communities. Arm recognizes that we and our industry have used language that can be offensive. Arm strives to lead the industry and create change.

This document includes language that can be offensive. We will replace this language in a future issue of this document.

To report offensive language in this document, email terms@arm.com.

Contents

1. Overview

To avoid multithreading performance problems when using Arm Compiler for Linux, it is important that you have the appropriate environment set up.

This guide will help you avoid some of the common pitfalls.

2. Set the number of OpenMP threads

To set the number of threads to use in your program, set the environment variable `OMP_NUM_THREADS`. `OMP_NUM_THREADS` sets the number of threads used in OpenMP parallel regions defined in your own code, and within Arm Performance Libraries. If you set `OMP_NUM_THREADS` to a single value, your program uses a single level of parallelism. In this case, nested parallelism is disabled.


Note

The information about setting `OMP_NUM_THREADS` applies to both compilers supported by Arm Performance Libraries in release 22.0: Arm Compiler 22.0 and GCC 11.2.

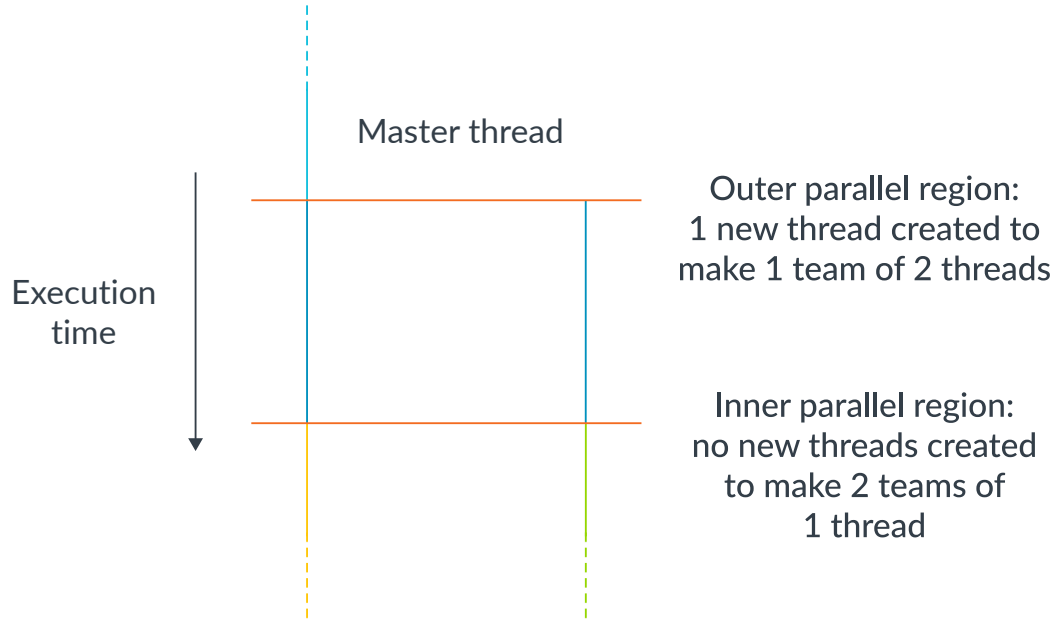
For example, consider the following code, which defines a nested parallel region:

```
#include <stdio.h>
#include <omp.h>

int main() {
        #pragma omp parallel
        {
                printf("outer: omp_get_thread_num = %d omp_get_level = %d\n",
 omp_get_thread_num(), omp_get_level());
                #pragma omp parallel
                {
                    printf("inner: omp_get_thread_num = %d omp_get_level = %d\n",
 omp_get_thread_num(), omp_get_level());
                }
        }
}

> armclang -o a1.out -fopenmp threading.c
> OMP_NUM_THREADS=2 ./a1.out
outer: omp_get_thread_num = 0 omp_get_level = 1
inner: omp_get_thread_num = 0 omp_get_level = 2
outer: omp_get_thread_num = 1 omp_get_level = 1
inner: omp_get_thread_num = 0 omp_get_level = 2

> gcc -o g1.out -fopenmp threading.c
> OMP_NUM_THREADS=2 ./g1.out
outer: omp_get_thread_num = 0 omp_get_level = 1
inner: omp_get_thread_num = 0 omp_get_level = 2
outer: omp_get_thread_num = 1 omp_get_level = 1
inner: omp_get_thread_num = 0 omp_get_level = 2
```

The program above reports the thread number and level of parallel nesting. Executables built with either GCC or Arm Compiler for Linux show the same behavior when `OMP_NUM_THREADS` is set to a single value (and all other settings use default values).

The example above sets `OMP_NUM_THREADS=2` and the output shows that two threads are used for the outer parallel region. The nested parallel regions create no new threads:

Figure 2-1: no nested parallelism



Note

The actual number of threads used during execution of your program might differ from the value specified in `OMP_NUM_THREADS` if the number of threads is set explicitly in the code using the OpenMP API, or if a system-defined limit is encountered.

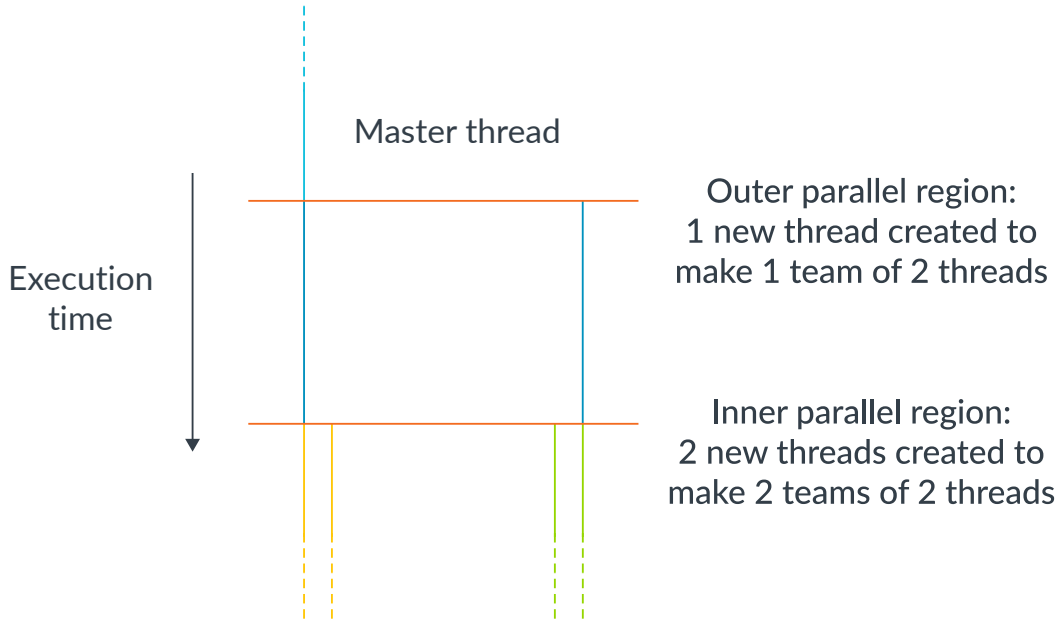
`OMP_NUM_THREADS` can also be set to a comma-separated list of values. Where a list of values are passed to `OMP_NUM_THREADS`, the values denote the number of threads to use at each level of nesting, starting from the outermost parallel region.

The default behavior when using a list of values with `OMP_NUM_THREADS` differs between Arm Compiler for Linux and GCC. For example, using the same executables as compiled earlier:

```
> OMP_NUM_THREADS=2,2 ./a1.out

outer: omp_get_thread_num = 0 omp_get_level = 1
outer: omp_get_thread_num = 1 omp_get_level = 1
inner: omp_get_thread_num = 0 omp_get_level = 2
inner: omp_get_thread_num = 1 omp_get_level = 2
inner: omp_get_thread_num = 0 omp_get_level = 2
inner: omp_get_thread_num = 1 omp_get_level = 2

> OMP_NUM_THREADS=2,2 ./g1.out
outer: omp_get_thread_num = 0 omp_get_level = 1
inner: omp_get_thread_num = 0 omp_get_level = 2
outer: omp_get_thread_num = 1 omp_get_level = 1
inner: omp_get_thread_num = 0 omp_get_level = 2
```

The example above specifies that the two parallel regions in the code can each use two threads. The Arm-compiled executable creates a new thread in each of the two inner parallel regions, enabling nested parallelism:

Figure 2-2: nested parallelism



However, the GCC-compiled executable shows the same output as with `OMP_NUM_THREADS=2`, keeping nested parallelism disabled.

The reason for this difference in behavior is because the OpenMP runtime provided with Arm Compiler for Linux (version 21.1 and later) uses `OMP_NESTED=true` when `OMP_NUM_THREADS` is a comma-separated list. The OpenMP runtime provided with the GCC 10.2 (and later) compiler has `OMP_NESTED=false` when `OMP_NUM_THREADS` is a comma-separated list.

Notes:

- The `OMP_NESTED` setting is being deprecated for OpenMP 5.0.
- This is a change of behavior for executables linked to the OpenMP runtime in Arm Compiler for Linux (version 21.1 and later). Previous Arm Compiler for Linux behavior matched the current behavior for gcc.

To enable nested parallelism for the GCC-compiled executable, explicitly turn on nesting:

```
> OMP_NESTED=true OMP_NUM_THREADS=2,2 ./g1.out
outer: omp_get_thread_num = 0 omp_get_level = 1
outer: omp_get_thread_num = 1 omp_get_level = 1
inner: omp_get_thread_num = 0 omp_get_level = 2
inner: omp_get_thread_num = 1 omp_get_level = 2
inner: omp_get_thread_num = 0 omp_get_level = 2
```

```
inner: omp_get_thread_num = 1 omp_get_level = 2
```

Nested parallelism in Arm Performance Libraries is handled in the same way as shown in these examples; if an Arm Performance Libraries routine is called from a parallel region in your code, then the routine spawns threads in the same way as shown for the nested parallel region in the examples above.

3. Control the placement of OpenMP threads

The value of the environment variable `OMP_PROC_BIND` affects how threads are assigned to cores on your system (also known as thread affinity). If `OMP_PROC_BIND=false` or is unset, then threads are unpinned; they might be migrated between cores in the system during execution, and thread migration will most likely degrade performance significantly.

Arm recommends setting `OMP_PROC_BIND` to either `true`, `close` or `spread`, as required.

If set to `close` then the OpenMP threads are pinned to cores close to the parent thread. `OMP_PROC_BIND=close` is useful where threads in a team are working on locally shared data. For example, if threads are pinned to neighboring cores there might be a performance benefit from the data being stored in a shared level of cache.

If set to `spread` then the OpenMP threads are pinned to cores that are distant from the parent thread. `OMP_PROC_BIND=spread` is useful to avoid contention on hardware resources. For example, if threads are working on large amounts of private data then there might be an advantage to using `spread` to reduce contention on a shared level of cache or memory bandwidth.

Setting the value to `true` avoids thread migration, but does not specify a particular affinity policy.

Another option is to set `OMP_PROC_BIND` to master. If `OMP_PROC_BIND=master`, all OpenMP threads in a team are pinned to the same core as the master thread.

- `OMP_PROC_BIND` can be set to a comma-separated list of the values described above, which sets the affinity policy separately for each level of nested parallelism.
- The values assigned to OpenMP environment variables are case insensitive.

The descriptions above describe how OpenMP threads are pinned to cores in the system. However, the OpenMP specification uses the term *place* to denote a hardware resource for which threads can have affinity. The environment variable `OMP_PLACES` allows you to define what is meant by a `place` in the system.

`OMP_PLACES` can be set to one of three pre-defined values: `threads`, `cores` or `sockets`. Setting `OMP_PLACES=threads` assigns OpenMP threads to hardware threads in the system. On a system where a single core supports multiple hardware threads (for example, Marvell ThunderX2 systems with SMT>1), assigning OpenMP threads to hardware threads allows for the co-location of several threads in a single core.

If the value is set to `cores` then each OpenMP thread is assigned to a different core in the system, which might support more than one hardware thread.

If the value is set to `sockets` then each OpenMP thread is assigned to a single socket in the system, which contains multiple cores. Where `sockets` is set, the OpenMP threads might migrate in the assigned socket.

To more finely control the placement of OpenMP threads in your system, set `OMP_PLACES` to a list of numbers that indicate the IDs of hardware places in your system (typically hardware threads). There is a considerable amount of flexibility availability using `OMP_PLACES`, including the ability to exclude places from thread placement. If you are interested in this level of control, refer to the OpenMP specification and experiment on your system.

4. Report OpenMP settings

Another useful environment variable to use when running OpenMP-enabled programs is `OMP_DISPLAY_ENV`. `OMP_DISPLAY_ENV` can be set to one of `true`, `false` or `verbose`. If `OMP_DISPLAY_ENV=true` is set, on startup your program displays the version of OpenMP along with the value for all of the OpenMP internal control variables (ICVs), which are affected by environment variables, such as those seen in this document, in addition to other factors.

There might be a discrepancy between the value of your environment variables and ICVs reported at runtime because ICVs can be controlled in other ways.

If `OMP_DISPLAY_ENV=verbose` is set, the values of any implementation-specific variables are displayed in addition to the standard OpenMP ICVs.

If `OMP_DISPLAY_ENV=false` or is undefined, no output is produced.

5. Related information

Here are some resources related to material in this guide:

- To learn more about OpenMP thread mapping, see the OpenMP thread mapping topic in the Porting and Tuning HPC Applications for Arm guide.
- To learn more about Arm Performance Libraries, see the Arm Performance Libraries Reference guide.