OpenMP - shared memory and device parallelism

This document provides a guide to usage of OpenMP and availability of OpenMP on the HPC systems at LRZ.

- An abstract description of OpenMP
- The OpenMP parallelization model
- Comparison with other parallelization methods
- Overview of OpenMP functionality
- Remarks on the usage of OpenMP
  - OpenMP compilers
  - Compiler switches
  - Controlling the Run Time environment
  - Stub library, module and include file for Fortran
  - Additional environment variables supported by the Intel Compilers
  - Additional environment variables supported by the GNU Compilers
- References

An abstract description of OpenMP

OpenMP is a parallelization method available for the programming languages Fortran, C and C++, which is targeted toward use on shared memory systems. Since the OpenMP standard was developed with support from many vendors, programs parallelized with OpenMP should be widely portable.

The current standard document is that for OpenMP 5.0. For the examples annex, only the 4.5 version exists as a (separately published) document; this in particular extends OpenMP to apply to accelerator devices. Actual compiler support is now reasonably complete, however support for specific accelerator devices is heavily compiler dependent, which rather limits portability of code that uses the target directive.

The OpenMP parallelization model

From the operating system point of view, OpenMP functionality is based on the use of threads, while the user's job simply consists in inserting suitable parallelization directives into her/his code. These directives should not influence the sequential functionality of the code; this is supported through their taking the form of Fortran comments and C/C++ preprocessor pragmas, respectively. However, an OpenMP aware compiler will be capable of transforming the code-blocks marked by OpenMP directives into threaded code; at run time the user can then decide (by setting suitable environment variables) what resources should be made available to the parallel parts of his executable, and how they are organized or scheduled. The following image illustrates this.
However hardware and operation mode of the computing system put limits to the application of OpenMP parallel programs: Usually, it will not be sensible to share processors with other applications because scalability of the codes will be negatively impacted due to load imbalance and/or memory contention. For much the same reasons it is in many cases not useful to generate more threads than CPUs are available. Correspondingly, you need to be aware of your computing centers’ policies regarding the usage of multiprocessing resources.

At run time the following situation presents itself: Certain regions of the application can be executed in parallel, the rest - which should be as small as possible - will be executed serially (i.e., by one CPU with one thread).

### OpenMP Execution Model

[Diagram showing the execution model of OpenMP]

Program execution always starts in serial mode; as soon as the first parallel region is reached, a team of threads is formed ("forked") based on the user’s requirements (4 threads in the image above), and each thread executes code enclosed in the parallel region. Of course it is necessary to impose a suitable division of the work to be done among the threads ("work sharing"). At the end of the parallel region all threads are synchronized ("join"), and the following serial code is only worked on by the master thread, while the slave threads wait (shaded yellow squares) until a new parallel region begins. This alteration between serial and parallel regions can be repeated arbitrarily often; the threads are only terminated when the application finishes.

The OpenMP standard also allows nesting of parallelism: A thread in a team of threads may generate a sub-team; some OpenMP implementations however do not allow the use of more than one thread below the top nesting level.

A priori the number of threads used does not need to conform with the number of CPUs available for a job. However for achieving good performance it is necessary to determine and possibly enforce an optimal assignment of CPUs to threads. This may involve additional functionality in the operating system, and is not covered by the OpenMP standard.

### Comparison with other parallelization methods

In contrast to using MPI (which usually requires a lot of work), one can very quickly obtain a functioning parallel program with OpenMP in many cases. However, in order to achieve good scalability and high performance it will be necessary to use suitable tools to perform further optimization. Even then, scalability of the resulting code will not always be on par with the corresponding code parallelized with MPI.

<table>
<thead>
<tr>
<th></th>
<th>MPI (shared and distributed memory systems)</th>
<th>OpenMP (shared memory systems)</th>
<th>proprietary directives</th>
<th>PGAS (coarrays, UPC)</th>
</tr>
</thead>
<tbody>
<tr>
<td>portable</td>
<td>yes</td>
<td>yes</td>
<td>no</td>
<td>yes</td>
</tr>
</tbody>
</table>
On high performance computing systems with a combined shared and distributed memory architecture using MPI and OpenMP in a complementary manner is one possible strategy for parallelization ("hybrid parallel programs"). Schematically one obtains the following hierarchy of parallelism:

- The job to be done is subdivided into large chunks, which are distributed to compute nodes using MPI.
- Each chunk of work is then further subdivided by suitable OpenMP directives. Hence each compute node generates a team of threads, each thread working on part of a chunk.
- On the lowest level, e.g. the loops within part of a chunk, the well-known optimization methods (either by compiler or by manual optimization) should be used to obtain good single CPU or single thread performance. The method used will depend on the hardware (e.g., RISC-like vs. vector-like).

Note that the hardware architecture may also have an influence on the OpenMP parallelization method itself. Furthermore, unlimited intermixing of MPI and OpenMP requires a thread-safe implementation of MPI; the level of thread-safeness can be obtained by calling the \texttt{MPI Init thread} subroutine with suitable arguments; depending on the result, appropriate care may be required to follow the limitations inherent in the various threading support levels defined by the MPI standard.

**Overview of OpenMP functionality**

Please consult the \texttt{Wikipedia OpenMP article} for such an overview.

**Remarks on the usage of OpenMP**

**OpenMP compilers**

OpenMP enabled compilers are available on all HPC systems at LRZ:

- The current (19.0) Intel Compiler suite supports most features of OpenMP 4.5, particularly the SIMD directives. These compilers are available on all HPC systems at LRZ.
- The PGI Compiler (2019) suite supports OpenMP 3.1 and some 4.5 features on x86_64 based systems. Accelerator support however does not use OpenMP, but is based on \texttt{OpenACC}.
- The currently deployed default release (7) of GCC supports OpenMP.
- The NAG compiler fully supports OpenMP 3.1 in its 6.2 release.

**Compiler switches**

For activation of OpenMP directives at least one additional compiler switch is required.

<table>
<thead>
<tr>
<th>Vendor</th>
<th>Compiler calls</th>
<th>OpenMP option</th>
</tr>
</thead>
<tbody>
<tr>
<td>Intel</td>
<td>ifort / icc / icpc</td>
<td>-qopenmp</td>
</tr>
<tr>
<td>GCC</td>
<td>gfortran / gcc / g++</td>
<td>-fopenmp</td>
</tr>
<tr>
<td>PGI</td>
<td>pgf90 / pgcc / pgCC</td>
<td>-mp</td>
</tr>
</tbody>
</table>

Please specify this switch for all program units that contain either OpenMP directives, or calls to OpenMP run time functions. Also, the OpenMP switch must be specified at link time.

**Controlling the Run Time environment**

Before executing your OpenMP program, please set up the OpenMP environment using one or more of the variables indicated in the following table.

<table>
<thead>
<tr>
<th>Variable name</th>
<th>possible values</th>
<th>meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>OMP_NUM_THREADS</td>
<td>a positive integer, or a comma-separated list of positive integers</td>
<td>The number of threads that will be spawned by the first encountered parallel regions, or by subsequent nested parallel regions.</td>
</tr>
</tbody>
</table>
### Simple example (using the bash or ksh shell):

```bash
echo OMP_NUM_THREADS=8
OMP_PLACES=cores
OMP_PROC_BIND=spread
./my_openmp_program.exe
```

This will execute the OpenMP-compiled binary "my_openmp_program.exe" with up to 8 threads.

### Stub library, module and include file for Fortran

In order to keep code compilable for the serial functionality, any OpenMP function calls or declarations should also be decorated with an active comment:

```fortran
implicit none
...
!$ integer OMP_GET_THREAD_NUM
!$ external OMP_GET_THREAD_NUM
...
!$ mythread = 0
!$ mythread = OMP_GET_THREAD_NUM()
```

Note that without the `IMPLICIT NONE` statement and missing declaration `OMP_GET_THREAD_NUM` has the wrong type!

If you do not wish to do this, it is also possible to use

- either an include file `omp_lib.h`
- or a Fortran 90 module `omp_lib.f90`

for compilation of the serial code. For linkage one also needs a stub library. All this is provided in the Intel compilers via the option `-openmp-stubs`, which will otherwise produce purely serial code. The above code can then be written as follows:

<table>
<thead>
<tr>
<th>Fortran 77 style</th>
<th>Fortran 90 style (recommended)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### Additional environment variables supported by the Intel Compilers

<table>
<thead>
<tr>
<th>Name</th>
<th>Explanation</th>
<th>Default value</th>
</tr>
</thead>
<tbody>
<tr>
<td>KMP_AFFINITY and /or KMP_PLACE_THREADS</td>
<td>see the description on Intel's web site</td>
<td>schedule threads to cores or threads (logical CPUs) in a user controlled manner.</td>
</tr>
<tr>
<td>KMP_ALL_THREADS</td>
<td>maximum number of threads available to a parallel region</td>
<td>max(32, 4<em>OMP_NUM_THREADS, 4</em>(No. of processors))</td>
</tr>
<tr>
<td>KMP_BLOCKTIME</td>
<td>Interval after which inactive thread is put to sleep, in milliseconds. Should be short in throughput mode, can be longer in turnaround mode; see KMP_LIBRARY below.</td>
<td>200 milliseconds</td>
</tr>
<tr>
<td>KMP_LIBRARY</td>
<td>Select execution mode for OpenMP runtime library. Possible values are:</td>
<td>throughput</td>
</tr>
<tr>
<td></td>
<td>• throughput: optimized for sharing resources with other programs</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• turnaround: suited to dedicated use of resources, as in HPC.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• serial: enforce serial execution</td>
<td></td>
</tr>
<tr>
<td>KMP_MONITOR_STACKSIZE</td>
<td>Set stacksize in bytes for monitor thread</td>
<td>max(32768, system minimum thread stack size)</td>
</tr>
<tr>
<td>KMP_STACKSIZE</td>
<td>stack size (in bytes) usable for each thread. <strong>Change if your application segfaults for no apparent reason.</strong> You may also need to increase your shell stack limit appropriately.</td>
<td>2 MByte</td>
</tr>
<tr>
<td></td>
<td>With OpenMP 3.0 and higher, the standardized variable OMP_STACKSIZE should be used.</td>
<td></td>
</tr>
</tbody>
</table>

**Notes:**

- Setting suitable postfixes where appropriate allows you to specify units. I.e., KMP_STACKSIZE=6m sets a value of 6 MByte.
- There are also some extension routine calls, i.e. kmp_set_stacksize_s(...) with an implementation dependent integer kind as argument, which can be used instead of the environment variables described above. However this will usually not be portable and usage is hence discouraged unless for specific needs.

### Additional environment variables supported by the GNU Compilers

- **GOMP_CPU_AFFINITY**: Bind threads to specific CPUs
- **GOMP_STACKSIZE**: Set default thread stack size

### References

- OpenMP home page. *The central source of information about OpenMP*
- OpenMP Specifications for Fortran, C, and C++. Older versions of the spec can also be picked up here.
- Course material for a [two day OpenMP course](#) (collaboration between LRZ and RRZE)