Thread-Local Storage Extension to Support Thread-Based MPI/OpenMP Applications

Patrick Carribault, Marc Pérache, and Hervé Jourdren

CEA, DAM, DIF, F-91297 Arpajon, France
{patrick.carribault,marc.perache,herve.jourdren}@cea.fr

Abstract. With the advent of the multicore era, the architecture of supercomputers in HPC (High-Performance Computing) is evolving to integrate larger computational nodes with an increasing number of cores. This change contributes to evolve the parallel programming models currently used by scientific applications. Multiple approaches advocate for the use of thread-based programming models. One direction is the exploitation of the thread-based MPI programming model mixed with OpenMP leading to *hybrid* applications. But mixing parallel programming models involves a fine management of data placement and visibility. Indeed, every model includes extensions to privatize some variable declarations, i.e., to create a small amount of storage only accessible by one task or thread. This article proposes an extension to the Thread-Local Storage (TLS) mechanism to support data placement in the thread-based MPI model and the data visibility with nested hybrid MPI/OpenMP applications.

1 Introduction

With the advent of the multicore era, the architecture of supercomputers in HPC (High-Performance Computing) is evolving to integrate larger computational nodes with an increasing number of cores. This change advocates for an adaptation of parallel programming models currently used by scientific applications. Within the current Petaflop/s supercomputers, the MPI-everywhere programming model (or Pure-MPI mode [1]) is still widely exploited by scientific applications but this solution may evolve to integrate a full thread-aware programming model to benefit from the capacity of a single memory address space on a computational node. One direction is the exploitation of the thread-based MPI implementation, i.e., running every MPI task as a thread instead of a process. Such implementations [2,3] are then able to increase performance and reduce the memory footprint. This is a crucial advantage when the amount of memory per core is decreasing. But evolving to the *hybrid* paradigm by mixing MPI with another thread-based model is another solution. OpenMP [4] is a promising candidate to exploit a large number of cores sharing the same physical or virtual memory space. On the other hand, various multithreaded programming models are emerging to exploit multiple nodes of a cluster. For example, PGAS languages (Unified-Parallel C, Co-Array Fortran, ...) belong to such models.

B.M. Chapman et al. (Eds.): IWOMP 2011, LNCS 6665, pp. 80–93, 2011.

But underneath, message passing and thread creation/manipulation are still the main mechanisms used at execution time.

Mixing multithreaded programming models involves a fine management of data placement and visibility. Indeed, every model proposes extensions to privatize some variable declarations, i.e., to create a small amount of storage only accessible by one task or thread. For example, the OpenMP standard includes the *threadprivate* construct to duplicate the declaration of such variables per thread [5]. But the stacking of programming-model runtimes does not guarantee that such privatization will remain valid. Depending on the model standard, it may be difficult to determine the data sharing among parallel programming models. Furthermore, with the decreasing amount of memory per core, going to thread-based programming models might be an unavoidable step towards the exascale milestone [6]. But the thread-based parallel programming models complicate the shared and private attributes of variables because, for example, global variables are shared by default among all runtimes implementing such models.

This article proposes an extension to the Thread-Local Storage (TLS [7]) mechanism to support nesting of hybrid thread-based MPI and OpenMP applications to handle data visibility among these programming models. With a cooperation between the compiler and the runtime library, such a mechanism allows a larger flexibility to manipulate data within an application exploiting multiple multithreaded models.

This article is organized as follows: Section 2 introduces three examples motivating for a two-level private storage. Section 3 discusses the related work. Section 4 exposes the main contribution of the paper: the extension of the TLS mechanism, Section 5 illustrates this extension on hybrid MPI/OpenMP applications while Section 6 applies it to experiments before concluding in Section 7.

2 Motivating Examples

Figure 1-a introduces a short example of using a global variable inside an MPI program. The variable `a` is used to catch the rank of each MPI task (or MPI agent). With an MPI implementation running every task in a separate process, each task will have its own copy of this variable, leading to the expected behavior. But with a thread-based MPI, each task runs as a thread (through process virtualization, see Section 5.1). Therefore the variable `a` will be shared by default, leading to a non-deterministic result: every MPI task will print the same value corresponding to the last thread that wrote `a` before reaching the barrier. With a thread-based MPI implementation, a mechanism is needed to privatize every global variable to each MPI task and, therefore, each thread.

But this might not be enough. The second example (Figure 1-b) exposes an example using the hybrid MPI/OpenMP programming model. In this program, the global variable `a` is still used to store the MPI rank of each task but it is accessed inside an OpenMP parallel region. The expected behavior is to have the correct rank in every OpenMP thread created by the same MPI task, i.e., the value should be the same for every OpenMP thread belonging to the same

```
int a = -1 ;

void f() { printf(
     "Rank = %d\n", a ) ; }

int main( int argc,
     char ** argv ) {
  MPI_Init( &argc, &argv ) ;
  MPI_Comm_rank(
    MPI_COMM_WORLD, &a) ;
  MPI_Barrier(
    MPI_COMM_WORLD);
  f() ;
  MPI_Finalize() ;
}
```

-a- Example 1

```
int a = -1 ;

void f() { printf(
     "Rank = %d\n", a ) ; }

int main( int argc,
     char ** argv ) {
  MPI_Init( &argc, &argv ) ;
  MPI_Comm_rank(
    MPI_COMM_WORLD, &a) ;
  MPI_Barrier(
    MPI_COMM_WORLD);
#pragma omp parallel
  {
    f() ;
  }
  MPI_Finalize() ;
}
```

-b- Example 2

```
int a = -1 ;
int b = -1 ;

#pragma omp threadprivate(b)

void f() {
  printf( "MPI Rank = %d and"
    " OpenMP Rank = %d\n",
    a, b ) ; }

int main( int argc,
     char ** argv ) {
  MPI_Init( &argc, &argv ) ;
  MPI_Comm_rank(MPI_COMM_WORLD,
    &a) ;
  MPI_Barrier(MPI_COMM_WORLD);
#pragma omp parallel
  {
    b = omp_get_thread_num() ;
    f() ;
  }
  MPI_Finalize() ;
}
```

-c- Example 3

Fig. 1. Hybrid MPI/OpenMP Running Examples

OpenMP team. With a full thread-based approach (i.e., each MPI task runs on a thread), it means that the variable `a` should be privatized for every MPI thread, but each copy has to be accessible by the OpenMP threads located inside the same team. Therefore, this variable should not be duplicated for every thread in the application process, but only for the threads related to the MPI semantics. Threads created by the OpenMP programming model will have to share the same copy of this variable.

The previous example advocates for an adaptive behavior according to the thread semantics. The final example, presented in Figure 1-c, uses the mechanisms proposed by the OpenMP standard to privatize a variable for every thread. Indeed, the variable `b` is declared as `threadprivate`. This leads to the privatization of this variable for every OpenMP thread while the global variable `a` still holds the MPI rank of every MPI task. In addition, the context-switch mechanism has to be aware of this data-visibility issue when scheduling multiple threads on the same physical core. This program illustrates the need to handle variable visibility with a two-level hierarchy: one for the MPI task (in case of a thread-based MPI implementation) and one for the OpenMP thread, with the ability to access private variables for the corresponding MPI task.

3 Related Work

Multiple work deals with data visibility either automatically or manually. One area is related to the thread-based implementation of the MPI programming model standard. Adaptive MPI (AMPI [2,8]) is an MPI runtime relying on

multiple user-level threads per physical core to represent each MPI task. With such representation, as discussed in Section 2, every global variable has to be privatized per MPI task to be compliant with the MPI standard. The authors propose a set of source-to-source tools based on the Photran [9] plugins to remove all global variables and adapt the original legacy-MPI code to respect the MPI semantics [8]. They pack global variables together and include them inside a module. Then functions are modified to accept this new module as a parameter if needed. Even if this transformation applies well to Fortran, this might be trickier to exploit it inside a C++ program. Indeed, the object-oriented languages involve more complicated declaration and definition. Therefore heavy modifications to the source code might hamper some compiler optimizations. Furthermore, when dealing with multiple programming models, it would be necessary to pack variables in various modules/objects. The data-flow restoration would then be more complicated.

Another direction to deal with the visibility of such variables has been developed in the context of automatic parallelization. Variable privatization and array expansion [10] are two compiler techniques used to duplicate variables to allow a larger amount of parallelism. This transformation is mainly used to parallelize a loop nest, allowing a concurrent access to a variable if the dependencies carried by those variables were only anti- and false-dependencies (Write-After-Read and Write-After-Write). One advantage of this approach is that the compiler can still fully optimize the code after the transformation because, with an accurate pointer analysis, every thread accesses only one specific cell of each newly-created array. However, the major drawback is related to the source-code transformation. Arrays are created, packing every copy of one variable for each thread. It means that false sharing may append because data accessed by multiple threads are likely to fall inside the same cache line. Moreover, it is necessary to re-allocate every expanded array in case of new thread creation.

The final approach to deal with the motivating examples presented in Section 2 is to use a cooperative work between the compiler and the runtime library: Thread-Local Storage (TLS). This is the standard solution to implement the OpenMP `threadprivate` construct [5]. But this is not enough to express the needs for hybrid MPI/OpenMP programs. The next section explains the TLS mechanism before introducing the contribution of this paper: an extension to the TLS mechanism to deal with hybrid MPI/OpenMP applications.

4 Thread-Local Storage Extension

The previous sections illustrate the need for a flexible mechanism to handle the mix of multiple thread-based programming models like MPI and OpenMP. The contribution of this article is the extension of the thread-local storage (TLS) mechanism with an additional hierarchical level to support such model stacking. This section describes the TLS mechanism and explains our extension before detailing our current implementation.

4.1 Thread-Local Storage (TLS) Mechanism

The TLS mechanism [7] allows an efficient and flexible usage of thread-local data. It has been introduced to extend the existing POSIX keys. The TLS mechanism is an extension of the C and C++ programming languages to let the compiler take over the job in conjunction with the thread runtime. This extension is based on the new keyword `__thread`. It can be used in variable definitions and declarations. A variable defined and declared this way would automatically be allocated local to each thread. Moreover the TLS mechanism supports dynamically loaded libraries. The only real limitation is that in C++ programs the thread-local variables must not require a static constructor. This limitation will be removed in the next generation C++ standard (C++0x).

The TLS mechanism requires a compiler/runtime cooperation. The compiler parses the `__thread` keyword and replaces each access to a thread-local variable with a call to an operator that returns the variable address for the current thread. During the execution of the program, the runtime allocates enough memory for all thread-local variables. As far as dynamically-loaded modules are concerned, the runtime dynamically allocates memory required for the thread-local variables and frees the corresponding memory if the module is unloaded. To support this mechanism, the binary format is also extended to define thread-local variables separated from regular variables. The dynamic loader is able to initialize these special data sections. The thread library is in charge of allocating new thread-local data sections for new threads.

The handling of thread-local storage is not as simple as that of regular data. The data sections cannot simply be made available to the process and then used. Instead multiple copies must be created, all initialized from the same initialization image. To set up the memory for the thread-local storage, the dynamic linker gets the information about each module's thread-local storage requirements from the PT_TLS program header entry. The information of all modules is collected. This can possibly be handled with a set of records that contain:

- A pointer to the TLS initialization image;
- The size of the TLS initialization image;
- The offset of the variable inside the module m: $tlsoffset_m$;
- A flag indicating whether the module uses the static TLS model (only if the architecture supports the static TLS model);

Figure 2 illustrates the TLS general model.

Usually, finding the address of a thread-local variable is deferred to a function named `__tls_get_addr` provided by the runtime environment. This function is also able to allocate and initialize the necessary memory if it happens for the first time. The values for the module ID and the TLS block offset are determined by the dynamic linker at runtime and then passed to the `__tls_get_addr` function in an architecture-specific way. Then the `__tls_get_addr` function returns the computed address of the variable for the current thread.

The TLS mechanism is now the standard way to privatize a variable per thread. OpenMP compilers often use TLS to deal with the `threadprivate` attribute [5].

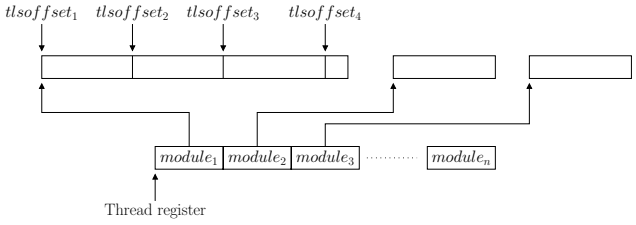


Fig. 2. TLS General Model

4.2 Our Extension: The Extended TLS

In order to deal with global variables in thread-based MPI implementations, we are tempted to use the TLS mechanism. As described in Section 2, this usage is incompatible with the OpenMP implementations that use the TLS approach to support the `threadprivate` variable attribute. To allow thread-based MPI and OpenMP runtimes to use the TLS mechanism, we propose to extend the regular TLS model with additional levels.

The main idea of this extension is to add the notion of levels to thread-local data. We propose two levels: the MPI and the OpenMP levels. In a thread-based MPI context, a variable is associated to the MPI level if all threads sharing the same MPI rank are able to access to the same copy of this variable. This level will therefore handle the global variables in the standard MPI programming way. The OpenMP level will be used for `threadprivate` variables. Figure 3 illustrates this extended TLS mechanism.

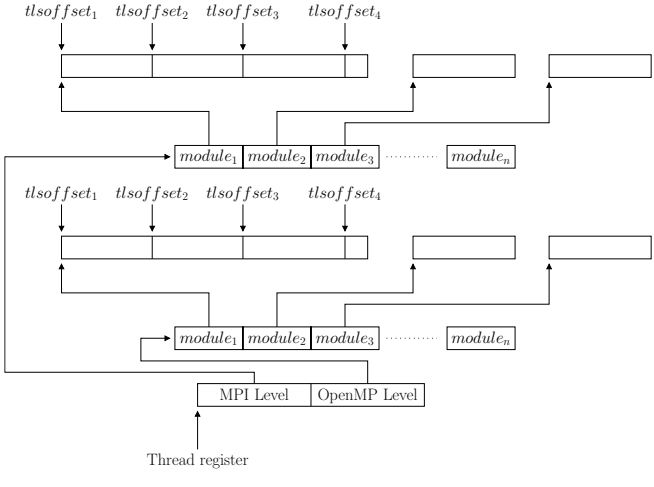


Fig. 3. Extended TLS model

4.3 Implementation

We implemented the extended TLS approach inside the MPC framework [11]. MPC is composed of a user-level thread library implementing a thread-based MPI runtime fully compatible with the 1.3 standard [3]. Furthermore, it provides a full OpenMP runtime compatible with the 2.5 standard [12] unified with the way MPI tasks are internally represented. In addition to this library, the MPC framework is shipped with a patched version of GCC to provide the compiler part of the OpenMP programming model. Because it already contains multiple implementations of thread-based programming models and a C/C++/Fortran compiler, it seems to be the right candidate to implement and to test the extended TLS mechanism.

Therefore, we implemented the main approach described in Section 4.2 inside the library of MPC. Thus the MPC user-level thread-library part has been updated to handle multiple levels of thread-local storage. It involves some modifications during thread creation and context switches. During the thread-creation process, we have to distinguish the thread semantics, i.e., the creation of threads that are designed to support MPI tasks. These threads have their own MPI and OpenMP levels of thread-local data. To avoid useless memory consumption, we implemented a lazy memory allocation. The thread-local data section for each module is allocated during the first access to the module. This allocation is performed using the *copy-on-write* mechanism. Instead of allocating and initializing a full section of a module, the runtime maps a common memory section of initalized variables with the *copy-on-write* attribute. This approach allows read-only variable sharing. Then the real memory allocation is performed during the first write. The creation of OpenMP threads requires a different behavior because OpenMP threads belonging to the same team share the variables related to the MPI programming model, i.e., they share the MPI level of the extended TLS. The MPI level of an OpenMP team points to the same memory area. Then, the OpenMP TLS level is managed with the same memory allocation policy as the MPI level one.

The second modification of MPC is context switching. At each context switch, it is mandatory to store/restore all levels of TLS mechanism. It implies an extension of context switching data structures.

As said in Section 4.1, the TLS approach requires a compiler support. The GCC compiler has been modified to deal with the right level of each variable. These modifications are compatible with the GCC patches shipped with the MPC distribution. The first modification to the compiler is the extension of the TLS definition inside the intermediate representations (GIMPLE and RTL). This is the most flexible solution to avoid breaking some optimization parts of the compiler. We then added one analysis and transformation pass to the compiler to flag variables with the right TLS level depending on the variable semantics. Global variables are put inside the MPI level while *threadprivate* variables are marked as extended-TLS with the OpenMP level. Our current implementation works with C, C++ and Fortran applications. The final modification is related to the code generation part of the compiler (*backend*). Depending on the TLS

level, the compiler generates a pattern of instructions with the right function calls. Instead of calling the standard function `__tls_get_addr`, the generated code calls the function related to the MPI or OpenMP level. These functions handle the extended TLS mechanism as previously-described in this section.

This full implementation is freely available with the 2.1 release of the MPC framework available at `http://mpc.sourceforge.net`.

5 Application to Hybrid Programming

The extension of the Thread-Local Storage mechanism allows more flexibility to deal with hybrid MPI/OpenMP programming and, with stacking multiple parallel programming runtimes, through the mix of various optimized libraries for example. This section describes the multiple benefits of our proposed TLS extension.

5.1 Thread-Based MPI Programming

First of all, the extended TLS mechanism provides an elegant way to deal with thread-based MPI implementations. MPI runtimes using thread-based implementations rely on the concept of process virtualization. Instead of creating one process per MPI task, those runtimes run each MPI task on a separate thread. Furthermore such implementations use multiple levels of user-level threads to deal with MPI applications. For example AMPI [2] and MPC [3] respect these concepts. The main advantage of such implementations is the possibility to optimize application execution based on the single address-space available between tasks located on the same computational node. Such runtime libraries are able to optimize both the execution time through decreasing the communication overhead, and the memory consumption by memory-page recycling between tasks and communication-buffer removal.

By applying the extended TLS, it is then possible to privatize every global variable to obtain one copy per MPI task. This is even possible for a user-level thread library such as AMPI and MPC. With the compiler support, it is only necessary to traverse all global variables and activate the right extended TLS flag to influence the code generation part. Thus it generates call to the runtime library to deal with such privatization level. Unlike array expansion, privatization or assembly manipulation, this solution does not change the source level and is completely transparent to the user.

5.2 Hybrid MPI/OpenMP Programming

The design of the extended TLS mechanism supports all hybrid thread-based MPI/OpenMP applications. Because MPC supports a unified representation of the MPI and OpenMP parallel programming models, the extended TLS is directly mapped to these models. As discussed in Section 5.1, the global variables are automatically privatized to the first level of the extended TLS (the MPI level) to deal with MPI semantics. With this transformation, each OpenMP parallel

region created by an MPI task will share the same copy of every global variable. This resolves the issues discussed on the motivating example Figure 1-b.

Furthermore, this extension and its implementation resolve the issues encountered by mixing these models with global variables and OpenMP *threadprivate* variables. Global variables are automatically placed inside the first level of our extended TLS mechanism while threadprivate variables are placed inside the second level. Therefore, each OpenMP thread will have its own copy of each threadprivate variable while threads inside the same OpenMP team will still share the same copy of each previously-global variable. Thus the variable `a` inside the example presented Figure 1-c will be put inside the MPI level while the variable `b` will be located inside the OpenMP level. With the modifications added to the memory allocation and the context switching parts of MPC, the data visibility will be correct in this example.

5.3 Other Applications

Beyond the applications described in Section 5.1 and 5.2, one may use this extended TLS mechanism to deal with embedded sub-programs inside a parallel thread-based application. For example JIT compilers may not be thread safe. With this approach, it is possible to recompile JIT compilers and privatize the global variables to the right level depending on the use of these compilers. For example, if some source code is compiled inside a thread-based MPI application, then by moving the global variables of the JIT compiler to the MPI level, it is guaranteed that multiple source codes could be compiled in parallel. This principle can be also applied to applications using interpreters to read input files in parallel.

6 Experimental Results

This section illustrates the extended TLS mechanism through experimental results: statistics on the amount of variables that should be privatized in well-known benchmarks and the overhead of the extended TLS compared to the standard TLS approach.

6.1 Statistics

To test and to validate the approach of extended TLS, we present here statistics about the number of global variables and threadprivate variables in various well-known benchmarks. For this purpose, we implemented a new analysis pass inside the GCC compiler to detect every global variable in a file, including the one created by the compiler during the early stage of the compilation process.

Intel MPI Benchmarks (IMB). IMB [13] is a benchmark collection to test the validity and performance of an MPI implementation (point-to-point communications, collective communications, ...). The source code used to check the MPI standard contains 77 global variables among 16,684 lines of source code. Thanks

to a simple symbol analysis, there are 298 static direct accesses to these global variables. For these simple benchmarks, it might be difficult to convert the source code to apply variable privatization because of the number of global variables and the static number of read/write accesses.

NAS-MPI 3.3. Table 1 exposes the number of global variables inside each NAS benchmark [14] using the MPI parallel programming model. This table summarizes the high-level language (C or Fortran), the number of lines of code and the total amount of global variables inside the whole application. Converting these benchmarks to be compatible with a thread-based MPI implementation requires some effort because the number of global variables is large compared to the amount of source code.

Table 1. Statistics NAS 3.3 Benchmarks (MPI Version)

Benchmark	Language	Lines of code	Number of global variables
BT	Fortran	9,217	200
CG	Fortran	1,796	95
DT	C	1,031	10
EP	Fortran	356	21
FT	Fortran	2,165	35
IS	C	1,126	39
LU	Fortran	5,937	115
NG	Fortran	2,543	57
SP	Fortran	4,922	173

NAS-MZ 3.2 Benchmark. The previous results expose statistics about converting simple benchmarks to thread-based MPI implementation. Table 2 presents the same statistics for the MultiZone version of the NAS benchmarks. These NAS-MZ programs use an hybrid MPI/OpenMP model. This table shows the number of global variables and threadprivate variables located inside each benchmark with their corresponding number of static accesses (read and write).

Table 2. Statistics of MultiZone Version of NAS 3.2 Benchmarks

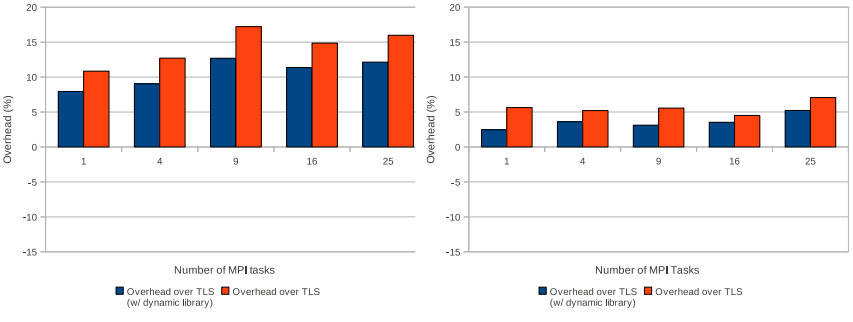
Statistics	BT	LU	SP
Language	Fortran	Fortran	Fortran
Lines of code	5,154	4,618	5,085
Number of global variables	173	175	132
Accesses to global variables	258	274	273
Number of threadprivate variables	11	11	5
Accesses to threadprivate variables	44	50	58

EPCC EPCC [15] is a set of micro-benchmarks to test the overhead of each OpenMP construct (entering/exiting a parallel region, executing a barrier, acquiring a lock, ...). It contains 8 global variables and 1 thread private variable. This amount is low but the source files are relatively small (2,971 lines of codes).

Linpack The HPL (High-Performance Linpack) benchmark contains 26 global variables with the MPI implementation.

6.2 Overhead of Extended TLS Mechanism

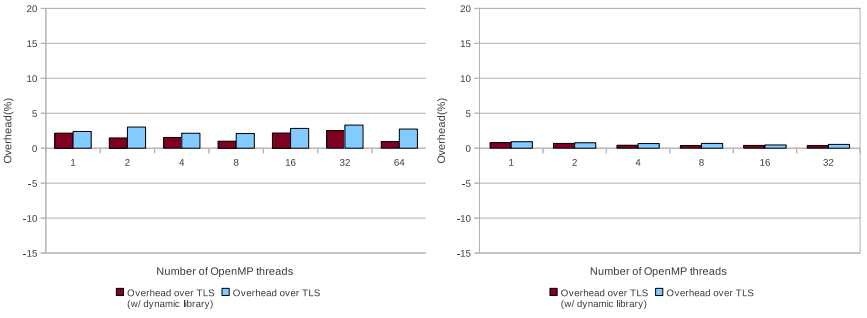
All the benchmarks presented in the previous section have been successfully compiled and executed with our extended TLS mechanism implemented inside the MPC framework. To check the overhead of our extension compared to the standard TLS mechanism, we measured the execution time of multiple NAS benchmarks with and without our extension. The target machine is one computational node of the Tera100 supercomputer: 4 sockets of Nehalem-EX processors for a total of 32 cores. Figures 4, 5 and 6 show the overhead of using the extended TLS mechanisms over the standard TLS with or without optimizations done by the linker on parallel NAS benchmarks (MPI, OpenMP and MultiZone). The first column of each graph depicts the overhead over the application using the standard TLS compiled in a dynamic library. Each global variable and thread-private variables are flagged as TLS and the compilation process puts these TLS variables in another module and hampers some linker optimizations [7]. On the other hand, the second column illustrates the results over the standard TLS mechanism.



-a- SP-MPI Class B -b- BT-MPI Class B

Fig. 4. Overhead of Extended TLS Approach on NAS 3.3 MPI Benchmarks

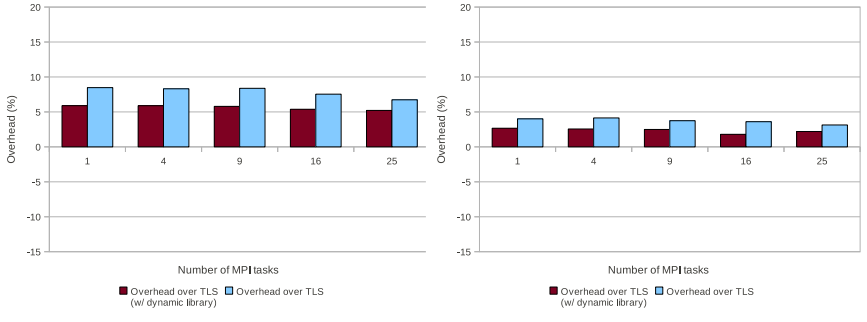
Figure 4 depicts the overhead on NAS-MPI benchmarks. In these examples, all global variables have been put to TLS or extended TLS. On the SP benchmark (Figure 4-a), the overhead can be as high as 17%. But it drops to 13% when the linker optimizations related to thread-local variables are disabled. It advocates for the modification of the linker tool to include the same kind of optimizations for the extended TLS. It would then be possible to reduce the overhead by hoisting many function calls. On the BT benchmark (Figure 4-b), the overhead is lower: between 1% and 5% without the linker optimizations. For this application, the overhead is low and it could be further reduced by improving the critical path when accessing extended-TLS variables. For example, the copy-on-write mechanism used to reduce the memory footprint may be optimized and/or disabled depending on the current memory consumption and the factor of NUMA accesses.



-a- SP-OpenMP Class B -b- BT-OpenMP Class B

Fig. 5. Overhead of Extended TLS Approach on NAS 3.3 OpenMP Benchmarks

Figure 5 depicts the overhead on NAS-OpenMP benchmarks. Variables flagged as threadprivate have been put to TLS or extended TLS. Compared to the MPI version of these benchmarks, the overhead is lower. This is related to the number of variables relying on the extended TLS mechanism. According to Section 6.1, the amount of threadprivate variables is relatively low (between 5 and 11).



-a- SP-MZ Class B -b- BT-MZ Class B

Fig. 6. Overhead of Extended TLS Approach on NAS 3.2 Multi-Zone Benchmarks

Finally, Figure 6 shows the overhead of our extended TLS approach on NAS-MultiZone benchmarks. This graph depicts the overhead with only one OpenMP thread. Again, the SP benchmark leads to a larger overhead (up to 10%) compared to BT. But this is lower than the previous NAS-MPI benchmarks. With more than one OpenMP thread, the TLS approach is not working with a thread-based MPI implementation. Indeed, all OpenMP threads and MPI tasks are implemented with threads. By using the traditional TLS mechanism, global variables are not inherited between the MPI task and the corresponding OpenMP threads belonging to the same team. However, our extended TLS approach allows the use of multiple OpenMP threads and MPI tasks with thread-based implementations.

7 Conclusion and Future Work

With the evolution of computer architecture and the decreasing amount of memory per core, the mix of multiple thread-based programming models is one interesting direction to continue exploiting high performance. But because of the resulting model stacking, data visibility and placement is an unavoidable issue. To deal with this issue, this article introduces an extension to the Thread-Local Storage (TLS) mechanism. By adding one additional level to this approach, it is now possible to handle data visibility with multiple thread-based programming models like MPI and OpenMP. We implemented this extension to the MPC framework. It is freely available within MPC 2.1 at `http://mpc.sourceforge.net`.

For future work, the linker must be updated to consider the extended TLS and then apply optimizations to increase performance. Finally, these two levels might not be enough when dealing with multiple programming models (e.g., with function calls to optimized libraries). It could be interesting to add new levels and manage them dynamically.

References

1. Message Passing Interface Forum: MPI: A Message Passing Interface Standard (March 1994)
2. Huang, C., Lawlor, O., Kalé, L.V.: Adaptive MPI. In: Proceedings of the 16th International Workshop on Languages and Compilers for Parallel Computing, LCPC (October 2004)
3. Pérache, M., Carribault, P., Jourdren, H.: MPC-MPI An MPI Implementation Reducing the Overall Memory Consumption. In: Proceedings of the 16th European PVM/MPI Users' Group Meeting, EuroPVM/MPI (September 2009)
4. OpenMP Architectural Board: OpenMP Application Program Interface (version 3.0) (May 2008)
5. Martorell, X., Gonzàlez, M., Duran, A., Balart, J., Ferrer, R., Ayguadé, E., Labarta, J.: Techniques Supporting Threadprivate in OpenMP. In: Proceedings of the 20th International Parallel and Distributed Processing Symposium (IPDPS). IEEE, Los Alamitos (2006)
6. Dongarra, J., Beckman, P., et al.: The International Exascale Software Project Roadmap (2011)
7. Drepper, U.: ELF Handling for Thread-Local Storage (2005)
8. Negara, S., Zheng, G., Pan, K.-C., Negara, N., Johnson, R.E., Kale, L.V., Ricker, P.M.: Automatic MPI to AMPI Program Transformation using Photran. In: 3rd Workshop on Productivity and Performance (PROPER 2010), Ischia/Naples/Italy, vol. (10-14) (August 2010)
9. Photran: An Integrated Development Environment for Fortran, `http://www.eclipse.org/photran`
10. Feautrier, P.: Array Expansion. In: Proceedings of the 2nd International Conference on Supercomputing, ICS 1988, pp. 429–441. ACM, New York (1988)
11. Pérache, M., Jourdren, H., Namyst, R.: MPC: A Unified Parallel Runtime for Clusters of NUMA Machines. In: Proceedings of the 14th International Euro-Par Conference (August 2008)

12. Carribault, P., Pérache, M., Jourdren, H.: Enabling low-overhead hybrid mPI/OpenMP parallelism with MPC. In: Sato, M., Hanawa, T., Müller, M.S., Chapman, B.M., de Supinski, B.R. (eds.) IWOMP 2010. LNCS, vol. 6132, pp. 1–14. Springer, Heidelberg (2010)
13. IMB: Intel MPI Benchmarks, `http://software.intel.com/en-us/articles/intel-mpi-benchmarks/`
14. Jin, H., Frumkin, M., Yan, J.: The OpenMP Implementation of NAS Parallel Benchmarks and Its Performance. Technical Report: NAS-99-011
15. Bull, J.M., O'Neill, D.: A Microbenchmark Suite for OpenMP 2.0. SIGARCH Comput. Archit. News 29(5), 41–48 (2001)