Automatic Handling of Global Variables for Multi-threaded MPI Programs

Gengbin Zheng, Stas Negara, Celso L. Mendes, Laxmikant V. Kalé
Department of Computer Science
University of Illinois at Urbana-Champaign
Urbana, IL 61801, USA
{*gzheng,snegara2,cmendes,kale*}*@illinois.edu*

Eduardo R. Rodrigues
Institute of Informatics
Federal University of Rio Grande do Sul
Porto Alegre, Brazil
errodrigues@inf.ufrgs.br

***Abstract*—**

Hybrid programming models, such as MPI combined with threads, are one of the most efficient ways to write parallel applications for current machines comprising multi-socket/multi-core nodes and an interconnection network. Global variables in legacy MPI applications, however, present a challenge because they may be accessed by multiple MPI threads simultaneously. Thus, transforming legacy MPI applications to be thread-safe in order to exploit multi-core architectures requires proper handling of global variables.

In this paper, we present three approaches to eliminate global variables to ensure thread-safety for an MPI program. These approaches include: (a) a compiler-based refactoring technique, using a Photran-based tool as an example, which automates the source-to-source transformation for programs written in Fortran; (b) a technique based on a global offset table (GOT); and (c) a technique based on thread local storage (TLS). The second and third methods automatically detect global variables and privatize them for each thread at runtime. We discuss the advantages and disadvantages of these approaches and compare their performance using both synthetic benchmarks, such as the NAS Benchmarks, and a real scientific application, the FLASH code.

I. Introduction

In computer programming, a global variable is a variable that is accessible in more than one scope of a program. It is widely and conveniently used in many applications written in Fortran, C/C++ and many other languages. However, global variables potentially cause many issues such as lack of access control and implicit coupling, which make the program more error-prone. Some of those issues may also be caused by the use of static variables. Therefore, it is often desirable to remove global and static variables as a way of promoting better programming practices.

Since the appearance of multicore processors, there is an increasing trend towards the use of multi-threaded programming to exploit parallelism in those processors, such as with OpenMP [1] and Intel's Threading Building Blocks (TBB) [2]. In many cases, it is desirable that two threads referring to the same global or static variable actually refer to different memory locations, thereby making the variable local to a thread. Unintentional sharing of global variables often causes race conditions when synchronization is not properly implemented. Due to this concurrency issue, it is necessary to privatize global and static variables to ensure the thread-safety of an application.

In high-performance computing, one way to exploit multi-core platforms is to adopt hybrid programming models that combine MPI with threads, where different programming models are used to address issues such as multiple threads per core, decreasing amount of memory per core, load imbalance, etc. When porting legacy MPI applications to these hybrid models that involve multiple threads, thread-safety of the applications needs to be addressed again. Global variables cause no problem with traditional MPI implementations, since each process image contains a separate instance of the variable. However, those variables are not safe in the multi-threading paradigm. Therefore, the global variables in the MPI code need to be privatized to become local to each MPI thread in order to preserve their original semantics. OpenMP solves this problem by offering a specific privatization directive for the key variables, but that solution is obviously specific to OpenMP programs.

In this paper, we investigate three approaches that *automatically* privatize global and static variables via compiler and runtime techniques. Handling the privatization problem in an automatic fashion relieves programmers from the onerous and error-prone process of manually changing their application codes. Also, it allows the use of the same original source code on different platforms where distinct privatization schemes are available. In addition to these benefits, our techniques can be uniformly applied to various programming paradigms that target multi-threaded execution.

We present the advantages and disadvantages of our privatization approaches, and compare their performance. We demonstrate the usefulness of these techniques in converting legacy MPI applications to a multi-threaded environment. In addition, by employing migratable user-level threads, rather than kernel threads like in other existing solutions, we show that significant performance improvements can be obtained by migrating those threads across processors to achieve better load balance.

The rest of this paper is organized as follows. Section II describes in detail the problem posed by global variables and the importance of properly handling it. Section III presents the techniques we use to approach the privatization problem.

Section IV contains our experimental performance results. Finally, Section V reviews related work and Section VI concludes our presentation.

II. Motivation

The Message Passing Interface (MPI) is a standardized library API for a set of message passing functions. It has become the *de facto* standard for parallel programming on a wide range of platforms. Due to its high efficiency in supporting communication, there are a large number of parallel applications already developed using MPI.

However, the conventional implementations of the MPI standard tend to associate one MPI process per processor, which limits their support for increasingly popular multi-core platforms. Thus, hybrid programming models that involve both MPI and threads, such as combining OpenMP with MPI, become attractive. As an example, the MPI + OpenMP model [3] parallelizes compute-bound loops with OpenMP, and inter-processor communication is handled by MPI. This hybrid model allows separation of concerns, isolating parallelism at multiple levels. However, this hybrid model can make the parallelization more difficult. It also requires a special thread-safe MPI implementation (which in many cases is not available), due to implicit use of threads in OpenMP. Global variables are handled by using the `threadprivate` directive, which specifies a variable to be private to a thread.

Another approach of combining MPI with threads is to adopt a finer-grained decomposition model using light-weight threads and run MPI "processes" in each thread. Adaptive MPI (AMPI) [4], [5], FG-MPI [6], Phoenix [7] and TMPI (Threaded MPI) [8] exemplify this approach. One advantage of this approach is that it allows automatic adaptive overlap of communication and computation — when one MPI thread is blocked to receive a message, another MPI thread on the same processor can be scheduled to be executed. Another advantage, obtained with user-level threads implemented in user space, is that the threads can be made migratable across processors. With sophisticated thread migration techniques [9], dynamic load balancing via migratable user-level threads can be supported at run-time.

In this paper, we will focus our work in the context of this MPI + user-level threads model. Our original motivating application was *Rocstar* [10], a very large-scale detailed whole-system multiphysics simulation of solid rocket motors under normal and abnormal operating conditions. Another motivating application was FLASH [11], [12], [13], a large scale multi-dimensional code used to study astrophysical fluids. Both codes are written in MPI, either in pure Fortran or mixed with C/C++ (*Rocstar*). In both applications, load imbalances due to different scale of physics make them ideal cases for the dynamic load balancing. There is virtually no source code change required to run these applications with the multi-threaded MPI model, since it is just another implementation of the MPI standard. However, one major obstacle for running these legacy MPI applications with the multi-threaded MPI execution model is global (and static) variables. In the original MPI codes, those variables cause no problem, since global and static variables reside in separate MPI process images. However, they are not thread-safe in the multi-threaded MPI case.

A. Problem with Global Variables

To concretely understand the problem induced by global variables in multi-threaded programs, it is instructive to analyze a real situation where that problem can arise. Consider the part of a hypothetical MPI program depicted in Figure 1. This code uses a global variable *var* to pass values between the main program and function *sub*; this illustrates a typical usage of global variables in traditional programs. In a pure MPI implementation, this usage is perfectly valid: there exists a separate instance of variable *var* for each MPI task, and all accesses to any particular instance are done by the corresponding MPI task only. Hence, when this particular code executes, each task will end up assigning a well-defined value to local variable *x* in the function, namely the value of *my_rank* that was set in the main program. This happens regardless of the temporal order of execution of the assignments across the various MPI tasks.

```
int var;  /* global variable */
...
int main(...) {
      ...
      MPI_Init(...)
      MPI_Comm_rank(...,&my_rank);
      ...
      MPI_Recv(...)
      var = my_rank;
      sub();
      ...
}
void sub() {
      int x;   /* local variable */
      ...
      MPI_Wait(...);
      x = var;
      ...
}
```

Figure 1. MPI code example showing the use of global variable

However, if the code in Figure 1 is used in a multi-threaded execution, a severe problem may arise. Suppose that each thread is executing the parts of the code where the accesses to *var* occur. Two threads belonging to the same process would then access the same instance of *var*, because there is a single instance of that global variable in a process. Two such threads, say those corresponding to MPI ranks 3

and 4, might interleave their accesses to *var* in a temporal order such as:

```
Rank 3:              Rank 4:
 var=3
                      var=4
                      x=4
 x=4
```

Hence, the thread corresponding to rank 3 ends up with its local variable *x* being assigned an incorrect value. In fact, this code makes the values assigned to *x* in each thread become totally dependent on the way in which the threads temporally interleave their executions. Thus, despite starting from a valid MPI code, the result of a multi-threaded execution is clearly unacceptable.

The problem illustrated in this example arises from the shared access to global variable *var*. Notice that this problem is not restricted to MPI codes. A similar kind of variable-sharing across OpenMP threads would present the same complication. A possible solution to this problem, without a complete restructuring of the code, is to make those critical variables become private to each thread (such as in OpenMP's *threadprivate* directive). With this privatization, each thread accesses a unique instance of the variable, thus avoiding data races due to unintentional sharing as in the example above. This paper presents various methods to automatically accomplish this privatization, in the context of threads (particularly user-level threads), and compares the characteristics of each method. We expect these privatization techniques to be useful to other MPI + threads programming paradigms as well.

III. Techniques to Privatize Global/Static Variables

The key to privatize global and static variables in user code is to identify these variables and automatically make multiple copies of them so that there is one copy for each thread. This can be done either at compile or run time. Three privatization techniques are investigated in this paper.

A. Source Code Transformation — Compiler Refactoring Tool

Global and static variables can be privatized by transforming the source code of an application. One way to accomplish this is, essentially, to put all global and static variables into a large object (a derived type in Fortran, or structure in C/C++), and then to pass this object around between subprograms. Thus, each MPI rank is given a different copy of this object. Figure 2 presents such privatizing transformation applied to the MPI code example of Figure 1. Note that this approach is valid for Fortran programs as well: Figure 3 shows a Fortran analogue of the MPI code example from Figure 1 and the corresponding privatizing transformation.

A more formal description of the code transformation required to privatize global and static variables in a Fortran (or C/C++) program is as follows. First, a new derived type (structure) is declared in a new module (file). This derived type (structure) contains a component for every global and static variable in the program. Every MPI process has its own instance of this type (structure). A pointer to this type (structure) is passed as an argument to every subprogram. Throughout the program, every access to a global or static variable is replaced with an access to the corresponding field of the derived type (structure). Finally, the declarations of global and static variables are removed from the program.

Even moderate size programs may contain hundreds of global and static variables scattered all over the code, which makes identifying such variables and, especially, privatizing them manually not only tedious, but also a highly error-prone process. To address this problem, we implemented a source-to-source transformation tool that automates global and static variables privatization in Fortran programs [14]. Note that although our implementation is language dependent, the underlying techniques are valid for other languages, including C/C++.

Our tool is implemented on top of the refactoring infrastructure in Photran [15], an Eclipse-based [16] Integrated Development Environment (IDE) for Fortran. Photran IDE exposes an Application Programming Interface (API) that provides functionality to parse a Fortran program and construct its Abstract Syntax Tree (AST) representation. The produced AST is *rewritable*, i.e. Photran's API allows AST manipulation and generation of the corresponding Fortran code. Also, the constructed AST is augmented with information about *binding* of program's entities (variables, subprograms, interfaces, etc.). Our tool analyzes the underlying Fortran program using information from its AST and transforms the program by manipulating its AST.

An important requirement of our source-to-source transformation tool is to produce efficient code. Our empirical experience suggests that the way our tool handles global fixed size arrays has a decisive impact on the performance of the transformed code. In real-world scientific computation programs there are many large fixed size arrays declared in different modules. If all these global arrays are placed in a single derived type, its size would exceed the maximum allowed size of a derived type, which may vary for different Fortran compilers, and is usually around several megabytes.

One solution to this problem is to transform fixed size arrays into pointer arrays and generate an initialization subroutine that allocates these arrays according to their sizes in the original program. This initialization subroutine is called right after `MPI_Init`, ensuring that every MPI process gets its own allocated and initialized copy of the transformed arrays. The shortcoming of this solution is that a Fortran compiler can not perform aggressive optimizations on pointers, and the transformed code is up to 20% slower

```
int var;   /* global variable */
...


int main(...) {
  ...
  MPI_Init(...);
  MPI_Comm_rank(...,&my_rank);
  ...
  MPI_Recv(...);
  var = my_rank;
  sub();
  ...
}


void sub() {
  int x;   /* local variable */
  ...
  MPI_Wait(...);
  x = var;
  ...
}
```

```
struct data{
  int var;
};
...
int main(...) {
  struct data *d;
  ...
  MPI_Init(...);
  d = (struct data*)malloc(sizeof(struct data));
  MPI_Comm_rank(...,&my_rank);
  ...
  MPI_Recv(...);
  d->var = my_rank;
  sub(d);
  ...
}
void sub(struct data *d){
  int x;
  ...
  MPI_Wait(...);
  x = d->var;
  ...
}
```

Figure 2. Example of the code transformation that privatizes C global variable `var`. The original code of a C MPI program is on the left; the transformed code, which does not contain global variables, is shown on the right.

than the original one [14].

To reduce this significant overhead, we implemented a different approach, which avoids dynamic allocation of global fixed size arrays. In this approach, we keep fixed size arrays and distribute them across multiple derived types, one array per type. Pointers to all these derived types are placed in a single derived type, which is used to pass around all previously global and static variables (including fixed size arrays). As a result, the overhead is reduced to 1% - 3%, while for some benchmarks, as demonstrated in Section IV, we even observed a speed up of up to 8% due to better spacial locality of these arrays.

Pros: The major advantage of the source-to-source transformation approach is its universality. It does not impose any additional requirements on a compiler or a runtime system. The result of the transformation is a regular Fortran code that can be compiled by any Fortran compiler and executed on any platform that supports the original code. Another important advantage of this approach is that it does not rely on the thread migration mechanisms provided by the runtime environment to support dynamic load balancing. Instead, our tool automatically produces a pack/unpack subroutine that takes care of migrating the generated derived type object, which contains all originally global and static variables. Also, by improving locality, the transformation may speed up small and medium size programs that contain global fixed size arrays scattered throughout the code.

Cons: Although source code transformation to privatize global and static variables is a language independent approach, its implementations are inherently language dependent, because they are analyzing and manipulating language-specific constructs. For example, our implementation handles only Fortran programs. Another limitation of this approach is that being independent of a runtime environment, it may not benefit from some of the runtime's features.

B. GOT-Globals Scheme — Using Global Offset Table

One interesting partial solution to the privatization problem is to take advantage of the position-independent code (PIC), which implements indirect access to global and static variables in Executable and Linking Format (ELF) binary format. Due to the support for dynamic shared libraries, data references to global variables from position-independent code[1] are usually made indirectly through the Global Offset Table (GOT), which stores the addresses of all accessed global variables. Note that static variables are treated differently. They are accessed through offsets from the beginning of the GOT [17].

At run time, an application can access the GOT information, and even modify it[2]. After the loader creates memory segments for the binary code, the GOT can be accessed by looking at the section headers through the address of the dynamic structure, referenced with the symbol _DYNAMIC. All relocatable variables can be identified by type `R_xxx_GLOB_DAT`, referring to the GOT. Having access to the GOT information at run time, an application can browse through all the global variables in the GOT entries and determine their sizes. For each thread, the application runtime makes a separate copy of the global variables exclusively for that thread. During execution, before the application switches to a given thread, the GOT entries are

[1]To generate position independent code, the source code must be compiled using position-independent compilation flags (e.g. -fPIC for GNU compilers).

[2]Some recent Linux kernels add protection to GOT, but calling *mprotect* can unprotect it

```
MODULE variables
  INTEGER :: var   ! global variable
END MODULE variables
...


PROGRAM Main
  USE variables
  ...
  CALL MPI_Init(...)
  CALL MPI_Comm_rank(...,my_rank)
  ...
  CALL MPI_Recv(...)
  var = my_rank
  CALL Sub
  ...
END PROGRAM Main


SUBROUTINE Sub
  USE variables
  INTEGER :: x   ! local variable
  ...
  CALL MPI_Wait(...)
  x = var
  ...
END SUBROUTINE Sub
```

```
MODULE variables
  TYPE data
    INTEGER :: var
  END TYPE data
END MODULE variables
...
PROGRAM Main
  USE variables
  TYPE(data) :: d
  ...
  CALL MPI_Init(...)
  CALL MPI_Comm_rank(...,my_rank)
  ...
  CALL MPI_Recv(...)
  d%var = my_rank
  CALL Sub(d)
  ...
END PROGRAM Main

SUBROUTINE Sub(d)
  USE variables
  TYPE(data) :: d
  INTEGER :: x
  ...
  CALL MPI_Wait(...)
  x = d%var
  ...
END SUBROUTINE Sub
```

Figure 3. Example of the code transformation that privatizes Fortran global variable `var`. The original code of a Fortran MPI program is on the left; the transformed code, which does not contain global variables, is shown on the right.

rewritten so that each GOT entry points to the global variable that is local to that thread.

This approach, which we name *GOT-Globals*, has been demonstrated on Linux OS on both 32-bit and 64-bit Intel x86 architectures. However, it does not support privatization of static variables, because the GOT table does not contain information on static variables. Clearly, this implementation relies on the ELF binary format to work. Although the ELF binary format is widely accepted, it still limits the scope of this method. It does not work, for example, on IBM's BlueGene/P, where shared libraries are not supported. However, the idea may apply to other object file formats that support shared libraries and generate position-independent code (such as XCOFF in IBM's AIX).

Pros: This approach is based on runtime techniques and features provided by the ELF object file format, therefore it does not require any source code modification. It is also language independent, and works for both C/C++ and Fortran. Using GOT-Globals allows the thread library to support the thread migration easily — since the user-level thread library allocates the memory segments for global and static variables, it can pack the memory data and move it together with the thread to a new processor.

Cons: The biggest constraint of this approach is that it can not handle static variables. Unfortunately, most legacy C/C++ and Fortran codes do use many static variables. These variables have to be handled separately by the application developer. In terms of overhead, this scheme requires a compile-time option (e.g. -fPIC) to generate position-independent code for global variables. Doing this alone, however, can slow down the application due to the indirect access to global variables, which can be a considerable factor depending on how frequently global variables are accessed. The overhead at thread context switching is $O(n)$, where n is the total number of global variables. This approach may also incur some overhead in memory usage, due to certain alignment requirements for global variables (for example, 16-byte alignment required by Intel's SSE instructions) that need to be respected when the application runtime creates separate copies of these global variables. Since the application runtime does not know exactly the alignment requirements (those decisions were made by the compiler), it has to conservatively assume the largest alignment requirement and apply it to every global variable. This may potentially result in unnecessary paddings that waste memory.

C. TLS-Globals Scheme — Using Thread Local Storage

Another privatization scheme is based on Thread Local Storage (TLS). By marking each global variable in a C program with the compiler specifier "*__thread*", these variables are allocated such that there is one instance of the variable per extant thread. This keyword is not an official extension of the C language, however compiler writers are encouraged to implement this feature. Currently, the ELF file format supports Thread Local Storage [18].

This mechanism, which we name *TLS-Globals*, relies on the compiler, linker and thread library to work. The compilers must issue references to private data through a

level of indirection [3]. The linker has to initialize a special section of the executable that holds exclusively thread-local variables. In addition, the thread library must allocate new thread-local data segments for newly created threads during execution, and switch the TLS segment register that points to the thread's local copy when a context switching between threads happens.

There are two issues regarding the use of TLS in our context of MPI programs. First, the user must explicitly mark those variables that are intended to be private with the specifier "*__thread*", in C, or an equivalent in other paradigms (such as OpenMP's *threadprivate* directive). However, if the user's intention is to privatize the entire set of global and static variables as in our use case, this may become a tedious process. A possible approach is to customize a compiler so that all global and static variables are treated as private by default. In this paper, we show examples where we modified the GFortran compiler to perform this kind of privatization. This was achieved by changing GFortran's back-end to generate code with the proper form of access to thread-private variables, i.e. by means of a level of indirection. That was simple to perform, since that back-end (which is the same as in GCC) already had routines to generate this type of code. In this way, all global, save and common variables are placed in the TLS segment.

A second issue with respect to the use of TLS is that the support of TLS is typically provided by the kernel thread library. For example, the pthread library directly supports this feature. However, until recently there was no user-level thread library that provided support for TLS. Nevertheless, the implementation of that support can be easily achieved [19].

Pros: The TLS scheme has the advantage that when it is applicable, it works uniformly on both kernel and user-level threads. It provides a simple and reliable privatization mechanism for both global and static variables. The overhead at thread context switch is to change the TLS segment register, which is $O(1)$. Compared with the GOT-Globals scheme, the overhead does not increase as the number of global/static variables increases. Similar to the GOT-Globals scheme, when the TLS-Globals scheme is used with user-level threads, migration of threads is easily supported — since the user-level thread library allocates the memory segments for global and static variables, it can pack the memory data and move it together with the thread to a new processor.

Cons: The main disadvantage of the TLS scheme, at this moment, is the fact that it is not yet universally supported. We believe, however, that such support is becoming increasingly common among system development tools, and thus it should gain wide adoption in the near future. Another disadvantage is that it may require modifications to the compiler such that every global and static variable be treated as thread-private. The use of such modified compilers would relieve the programmer from the burden of adding the "*__thread*" qualifier in C/C++ codes, or provide similar functionality in Fortran codes where an equivalent qualifier is not available. Although the modification to the compiler is simple (as we demonstrate with the GFortran compiler), it is not always possible if the compiler is not open source. Another practical limitation in our implementation is that the source code needs to be linked statically to ensure that there is only one TLS segment. This is to avoid the complexity incurred by linking with shared libraries, where new TLS segments are created when dynamic libraries are loaded.

[3]This sometimes requires that the source code be compiled with special flags e.g. -mno-tls-direct-seg-refs for GNU compilers.

D. Implementation

We evaluated the three previously described privatization schemes in a multi-threaded implementation of MPI called Adaptive MPI (AMPI) [4], [5]. AMPI executes MPI processes in light-weight user-level threads. These threads can be implemented by AMPI in various ways, depending on the underlying architectures. The most frequently used implementation is based on the system calls `setcontext()` and `swapcontext()`.

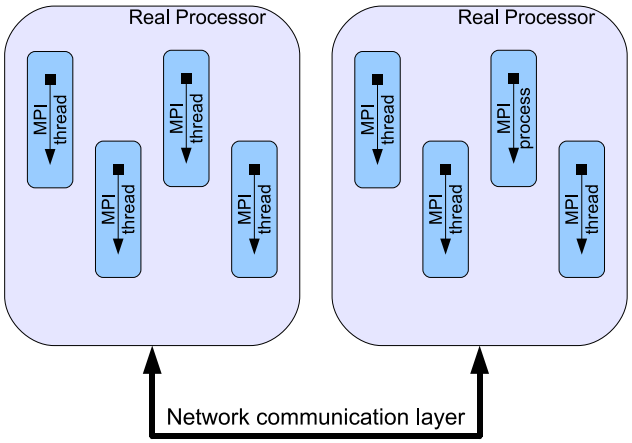


Figure 4. Diagram of AMPI model of virtualization of MPI processes.

Standard MPI programs divide the computation onto P MPI processes, and typical MPI implementations simply execute each process on one of the P processors. In contrast, an AMPI program divides the computation into a number V of AMPI user-level threads, and a runtime system maps these threads onto P physical processors, as illustrated in Figure 4. The number of threads V and the number of physical processors P are independent, allowing more flexibility. Dynamic load balancing is achieved in AMPI by means of moving user-level threads from overloaded processors to underloaded ones. When a thread migrates, it moves its

Privatization Scheme	X86	IA64	Opteron	Mac OS X	IBM SP	SUN	BG/P	Cray/XT	Windows
Transformation	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
GOT-Globals	Yes	Yes	Yes	No	No	Maybe	No	No	No
TLS-Globals	Yes	Maybe	Yes	No	Maybe	Maybe	No	Yes	Maybe

Table I
PORTABILITY OF CURRENT IMPLEMENTATIONS OF THREE PRIVATIZATION SCHEMES. "YES" MEANS WE HAVE IMPLEMENTED THIS TECHNIQUE. "MAYBE" INDICATES THERE ARE NO THEORETICAL PROBLEMS, BUT NO IMPLEMENTATION EXISTS. "NO" INDICATES THE TECHNIQUE IS IMPOSSIBLE ON THIS PLATFORM.

private copy of global and static variables, together with its stack and heap data, using a runtime memory management technique called *isomalloc* [9].

E. Portability

Table I illustrates the portability of our implementation of each privatization technique on various platforms. Not surprisingly, the program transformation technique is portable across all platforms. This is because the transformed/new MPI code is still a legitimate MPI program that can run on all the platforms that support MPI. The restriction on GOT-Globals is the requirement that the compiler be capable of generating ELF position-independent code with GOT. The ELF binary format is widely accepted (e.g. on Linux OS), which makes this GOT-Globals scheme fairly portable. However, note that in some scenarios, such as on the Cray/XT, although the ELF binary format is supported, the GOT is still missing in the produced binary (due to the light-weight Linux kernel used on that system). This limits the applicability of the GOT-Globals scheme. By contrast, the TLS-Globals scheme is constrained by OS support for TLS. Our implementation of TLS-Globals is for Linux x86 ELF object file format. However, as TLS becomes more widely supported on a variety of platforms, it should be possible to implement our TLS-Globals scheme on those platforms.

IV. PERFORMANCE COMPARISON

This section offers comparative evaluations for all three privatization schemes. We ran several micro-benchmarks, NAS benchmarks and a real-world application, FLASH. Enabled by the privatization techniques presented in this paper to handle the global variables, we demonstrate one of the benefits of using multi-threaded AMPI, namely dynamic load balancing, with the NAS benchmarks and FLASH code.

A. Micro-benchmarks

We started our experiments by comparing the three privatization schemes in terms of their basic performance effects on computation and communication. An important factor in a multi-threaded execution is the cost of context switch between threads. We created a simple benchmark that executes a few floating-point operations followed by a call to MPI_Barrier. This sequence is repeated multiple times, and we execute the code with two threads on one processor, under our AMPI environment. Because each call

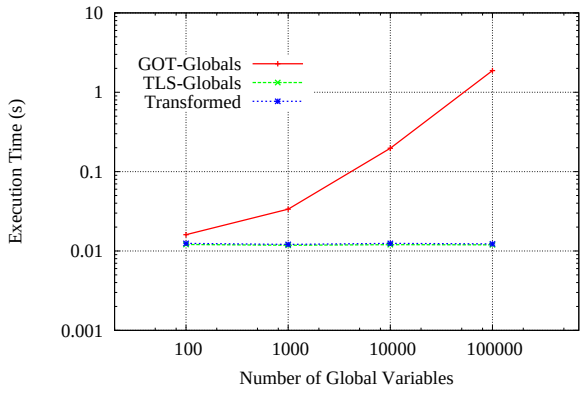


Figure 5. Effects of context switch overhead with all three schemes

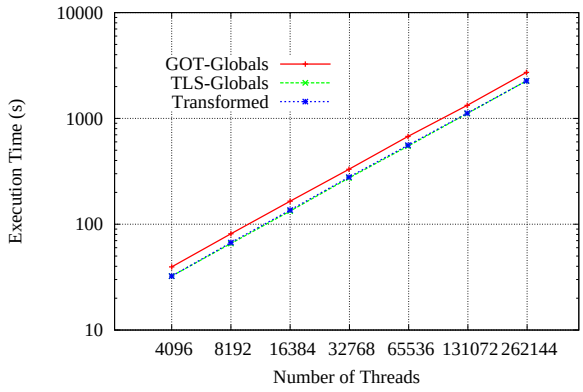


Figure 6. Execution times for test with 100 globals and varying number of threads

to MPI_Barrier implies a context-switch, the duration of this execution is directly affected by the cost of context-switching. Figure 5 shows the durations obtained with the three schemes, for executions with versions of the program containing a varying number of global variables (we stress that the amount of computation done is fixed, and we simply varied the number of *declared* global variables).

Since the GOT-Globals scheme changes all the existing globals at each context-switch moment, the cost of the executions with GOT-Globals in Figure 5 grows proportionally to the number of globals in use. Meanwhile, the

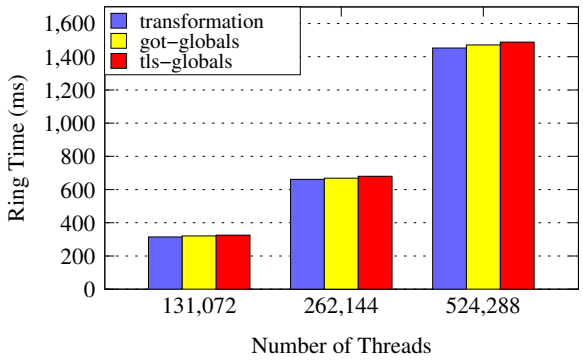


Figure 7. Ring execution times with up to half million threads with all three schemes

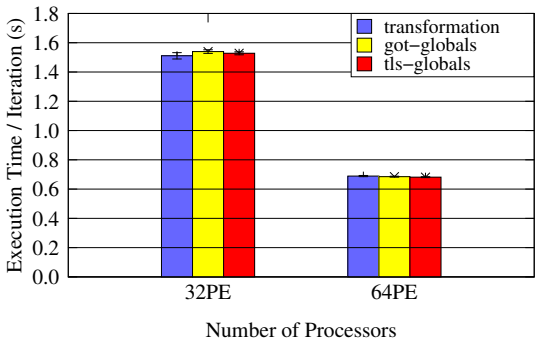


Figure 8. 7-point stencil execution time with 524,288 MPI threads with all three schemes (Ranger cluster)

TLS-Globals and Transformation schemes do not suffer this effect: the context-switch time is constant, regardless of how many globals exist in the code. Hence, the performance of an application with a large number of global variables can be severely hurt when GOT privatization is used.

We also executed the same program with multiple threads under the three privatization schemes, to assess their scalability with an increasing number of threads per processor. We employed the program version containing 100 global variables. Figure 6 shows the measured results. Since the amount of computation done by each thread is fixed at the same level as before, the execution durations depend only on the context-switch time and on the time for the threads to perform their work. Hence, the durations grow linearly with the number of threads, as confirmed by Figure 6. GOT-Globals, which requires copying global variables at context switch, is slightly more expensive than the other schemes.

Another simple benchmark we created to compare the three privatization schemes is an MPI ring program. In this program, the ring pattern communication starts from MPI rank 0 by sending a small message (around 100 bytes) to its next rank, and so on until the message comes back to rank 0. This sequence is repeated 100 times, and the average execution time of a single ring is reported. The only global variable in this program is "my_rank_id". We executed the program varying the number of threads from $1/8$ million to half million threads, all running on a multicore desktop; each thread had an $8K$-byte stack. The measured total number of context-switching with 131,072 threads is about 27,000,831, and 108,003,327 with 524,288 threads. Due to the fact that the number of global variables is so small, we see in Figure 7 that all three privatization schemes perform almost equally well. This experiment also shows that the AMPI runtime is capable of handling hundreds of thousands of threads on one processor.

B. 7-point Stencil

Next, we used a 7-point stencil program with 3-D domain decomposition written in MPI to evaluate our three privatization schemes. The stencil program was written so that in every iteration each MPI thread exchanges messages with its six neighbors and then performs Jacobi relaxation computations.

We ran this program for a total of 524,288 MPI threads (i.e. the size of the MPI_COMM_WORLD) with a 3-D block data of size $5120 * 2560 * 40$. Each MPI thread calculates a small block of size $10*10*10$, which is declared as a global array variable. Our test environment is a Sun Constellation Linux Cluster called Ranger installed at the Texas Advanced Computing Center. Ranger is comprised of $3,936$ 16-way SMP compute nodes providing a total of $62,976$ compute cores.

Figure 8 shows the execution times obtained with the three schemes using 32 and 64 processors of the Ranger cluster, respectively, where the 524,288 MPI threads were evenly distributed. The results, again, show that the three privatization schemes present nearly the same performance.

C. Multi-zone NAS Benchmarks

The NAS multi-zone benchmarks [20] are derived from the well-known NAS Parallel Benchmarks (NPB) suite. Application benchmarks LU-MZ, SP-MZ and BT-MZ solve discretized versions of the unsteady, compressible Navier-Stokes equations in three spatial dimensions. Multi-zone benchmarks use a strategy that exploits coarse-grain parallelism between meshes. Specifically, in BT, the partitioning of the mesh is done such that the sizes of the zones span a significant range, therefore creating imbalance in workload across processors, which provides a good case study for AMPI and its load balancing capability.

We transformed the three benchmarks, BT-MZ, LU-MZ and SP-MZ, which are all written in Fortran, using the Photran-based transformation tool, and compared them with GOT-Globals and TLS-Globals versions for various configurations of problem sizes and number of MPI ranks. Table II shows the total number of global and static variables, in the three benchmarks, that are handled by our privatization schemes. Note that although there are two static variables

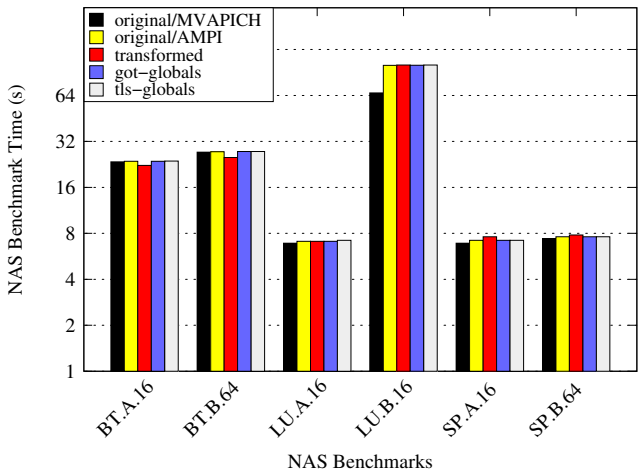


Figure 9. NAS Benchmark times on all three schemes vs. non-threaded MPI (Ranger cluster)

in these benchmarks, they do not cause problem for the GOT-Globals scheme because those two variables are never changed. The test environment was the Ranger cluster.

Benchmark	Global variables	Static variables	Total
BT-MZ	156	2	158
LU-MZ	99	2	101
SP-MZ	157	2	159

Table II
NUMBER OF GLOBAL AND STATIC VARIABLES IN NAS BENCHMARKS

For a fair comparison to the normal non-threaded MPI case (which was MVAPICH), in the first experiment with AMPI we limited the total number of MPI threads so that there was only one MPI thread per processor. The results of comparison of all three privatization schemes to the original code are shown in Figure 9. The first two bars represent a comparison between MVAPICH and AMPI with the same code, without any handling of global variables (the AMPI execution corresponding to the second bar had one thread per processor, so there was no concurrent access to global variables). Except for LU.B.16, AMPI performs almost as well as MVAPICH. Note that due to practical limitations on Ranger, there is no statically built Infiniband-ibverbs library. At the same time, our TLS-Globals scheme should be built statically, and thus, can not use the dynamically built Infiniband-ibverbs library. So, for a fair comparison among all three schemes, we built AMPI without specialized support for Infiniband. The significant difference between MVAPICH and AMPI in the case of LU.B.16 is probably due to the fact that the LU program is a communication intensive application.

Focusing on the AMPI cases (the last four bars in each test-case in Figure 9), we see that all three privatization schemes performed equally well, and did not add noticeable overhead in comparison to the AMPI execution of the original code. The transformed code performs substantially better in the BT benchmark case: instead of an overhead in comparison to the AMPI execution of the original code, we observed a speed up of almost 6% for BT.A.16 and more than 8% for BT.B.64 due to better spatial locality of the originally global fixed size arrays that became components of the same derived type object in the transformed code.

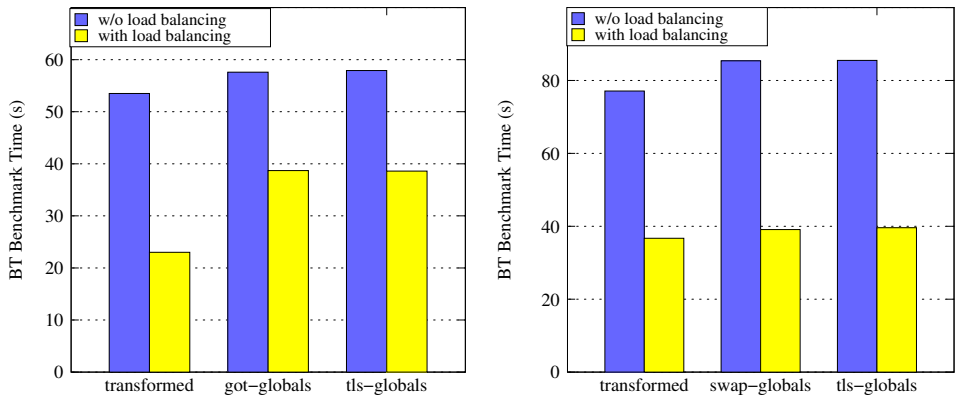
With the global variables properly handled in these benchmarks by the three privatization techniques, we ran the BT-MZ and LU-MZ benchmarks with AMPI and multi-threading, enabling dynamic load balancing. We employed a greedy-based load balancer that is invoked once after the third simulation step. For simple illustration, we always ran 4 MPI threads per processor when the program started. For example, the BT.B.64 test-case that is designed for 64 MPI ranks was executed on 16 real processors with 4 MPI threads each. However, since the load balancer may move MPI threads across processors during execution, the number of threads on a processor is not fixed.

The results with the BT-MZ and LU-MZ benchmarks are shown in Figure 10 and Figure 11, respectively. In BT-MZ, for all three privatization schemes, execution time improves dramatically after applying dynamic load balancing. The transformed code runs noticeably faster with and without dynamic load balancing. This is probably due to the runtime overhead that the other two schemes incur in order to enable migration of heap/stack data across processors, while the transformed code has pack/unpack functions automatically generated and therefore, it does not rely on the runtime memory management technique to move thread data across processors. The LU-MZ case does not benefit from dynamic load balancing, possibly because there is no load imbalance problem in LU. However, we can see that all three privatization schemes work equally well, and there is very little overhead when load balancing is enabled.

D. FLASH

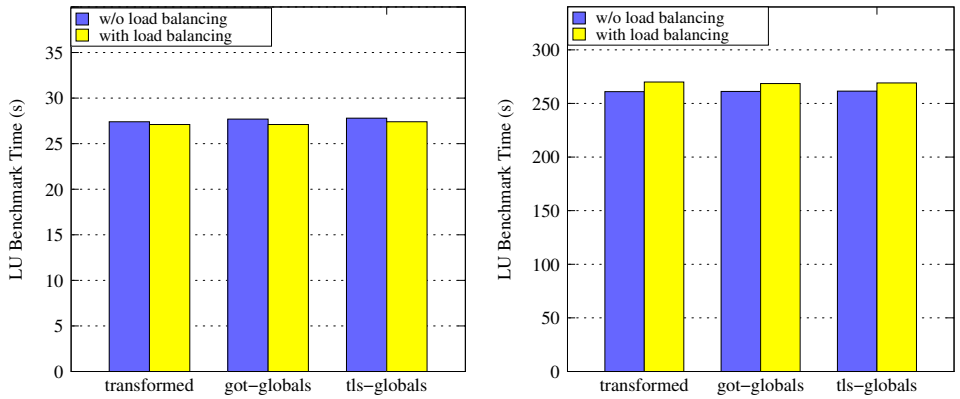
We evaluated the three privatization schemes on a large-scale project: FLASH, version 3 [11], [12], [13]. FLASH is a parallel, multi-dimensional code used to study astrophysical fluids. Many astrophysical environments are highly turbulent, and have structure on scales varying from large scale, like galaxy clusters, to small scale, like active galactic nuclei, in the same system. Thus, load balance issues become critical in recent computational astrophysics research, which makes it a good case for AMPI and its dynamic load balancing capability.

The FLASH code is written mainly in Fortran 90 and is parallelized using MPI. It is essentially a collection of code pieces, which can be customized in different ways to produce different simulation problems. FLASH supports both a uniform grid and a block-structured adaptive mesh refinement (AMR) grid based on the PARAMESH library.



(a) BT.A.16 (b) BT.B.64

Figure 10. BT-MZ Benchmark time with all three schemes (with and without load balancing)



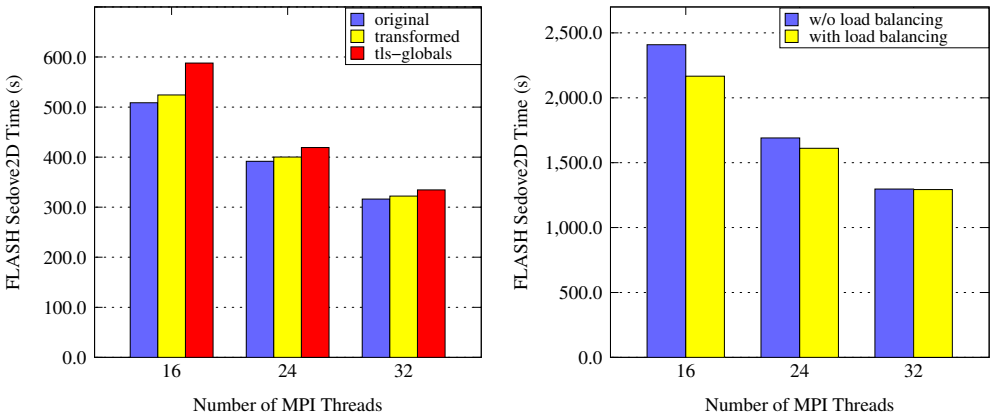
(a) LU.A.16 (b) LU.B.64

Figure 11. LU-MZ Benchmark time with all three schemes (with and without load balancing)

In the following experiments, we chose a certain simulation problem, Sedov-Taylor explosion, to evaluate our privatization schemes. We use 9 AMR levels and two-dimensional fluids for our tests.

The FLASH code contains 855 global and 399 static variables (total of 1254 variables) in this simulation test case. Due to the wide presence of static variables, we were not able to use the GOT-Globals scheme without a significant manual effort to handle them. The comparison between using the transformation and the TLS-Globals schemes is illustrated in Figure 12(a). The runs were performed with one AMPI thread per processor, so that we could compare them to the case of the original code that does not privatize the global/static variables. We see that the transformed code is only marginally slower than the original code, due to minimal overhead at runtime. The TLS-Globals scheme runs noticeably slower than the transformed code. This is possibly due to the fact that unlike the transformed code, the TLS-Globals scheme incurs considerable runtime overhead, in particular for the memory management of the TLS segment.

Finally, to demonstrate the load balancing capability, we executed 4 MPI threads on each processor to give enough opportunity for AMPI's load balancer to move threads around. We inserted load balancing calls in the simulation at every 100 steps. A greedy-based load balancer was invoked at the first load balancing step, and a refinement-based load balancer was used thereafter. The Sedov2D simulation execution times with and without load balancing are illustrated in Figure 12(b). We see about 10% of performance improvement after load balancing in the 16 MPI thread case (on 4 real processors), and 5% improvement for the 24-thread case. There is virtually no performance improvement for the 32-thread case, possibly due to the performance gain being offset by the load balancing overhead itself. These results are still encouraging, considering that PARAMESH, used in FLASH, is also performing its own load balancing at every refinement step, and the load balancing strategies we used here are simple off-the-shelf load balancers that do not take mesh refinement into account.



(a) Comparison of two privatization schemes

(b) Load balancing with the transformed code

Figure 12. FLASH Sedov2D simulation (Ranger cluster)

V. Related Work

Much work has been done in hybrid programming models that involve MPI and threads. Hybrid programming model with MPI+OpenMP [3] approaches the problem by distributing OpenMP threads among MPI processes. Users need to specify thread private variables by explicitly using "threadprivate" OpenMP directives. A compiler that supports OpenMP is required to compile such applications.

TMPI [8] uses multithreading for performance enhancement of multi-threaded MPI programs on shared-memory machines. Because it targets a shared-memory scenario, its main goal in implementing MPI tasks via threads is to provide a common address space to the threads such that memory copy in message-passing is avoided. There is no way to use more than one thread per processor, hence the number of threads is limited to the number of available processors.

More recent work, in FG-MPI [6], shares the same idea with AMPI by exploiting fine-grained decomposition using threads. However, FG-MPI does not support thread migration and dynamic load balancing. The three techniques to handle global variables described in this paper will benefit these MPI implementations as well.

Phoenix [7] is a runtime environment that, like our AMPI, transforms MPI processes into light-weight threads. However, Phoenix implements those threads with pthreads, and no support for thread migration exists. In addition, the Phoenix creators did not propose any scheme for variable privatization; they simply rely on the use of an existing source-to-source translator for C/C++ codes. Hence, Fortran programs would have to be be manually transformed. Here, again, our presented techniques could be effectively used to handle existing MPI programs.

SPAG [21] is a tool for analyzing and transforming Fortran programs. It provides both static and dynamic analysis, but its transformation capabilities are limited to a predefined set. ROSE [22] is a source-to-source compiler infrastructure to analyze and transform C, C++, and Fortran programs. Like in Photran, programs are represented with ASTs that can be manipulated and unparsed back to source code, but, to the best of our knowledge, no work has been done in ROSE to implement a tool that automatically privatizes global variables in legacy Fortran applications.

Weaves [23] is a run-time framework for an object-based compositional design and development of high performance codes. It uses similar ELF analysis techniques as our GOT-Globals scheme, although with a different purpose. It is used to enables selective sharing of code and state between parallel subprograms, whereas we used the same technique for privatization of global variables.

VI. Conclusions and Future Work

In this paper, we presented three techniques to automatically privatize global and static variables for any MPI implementation that supports a multi-threaded execution model. These approaches include: (a) a compiler-based refactoring technique, using a Photran-based tool as an example, which automates the source-to-source transformation for programs written in Fortran; (b) a technique based on a global offset table (GOT); and (c) a technique based on thread local storage (TLS). We discussed the advantages and disadvantages of these approaches and compared their performance using both synthetic benchmarks, such as the NAS Benchmarks, and a real scientific application, the FLASH code. We demonstrated that all these techniques work effectively in their domains, and they support thread migration for dynamic load balancing. When a thread migrates its data and execution to a new processor, it carries its "global" variables, which have been properly privatized. Using the NAS Benchmark and FLASH code as examples, we showed considerable performance improvement by dynamic load balancing, which is made possible partially by the techniques presented in this paper. With these techniques, legacy MPI

applications can be executed in the new MPI + threads model without change in their source code (or via automatic source transformation).

We plan to extend our privatization techniques to more platforms such as the upcoming Blue Waters machine. We also plan to apply these techniques to new applications, such as the emerging climate simulation model ESCM, and BigDFT, a density functional theory (DFT) massively parallel electronic structure code.

ACKNOWLEDGMENTS

This work was supported in part by NSF grant OCI-0725070 for Blue Waters deployment, by the Institute for Advanced Computing Applications and Technologies (IACAT) at the University of Illinois at Urbana-Champaign, and by Department of Energy grant DE-SC0001845. We used machine resources on the Range cluster (TACC), under TeraGrid allocation grant TG-ASC050039N supported by NSF. FLASH was developed by the DOE-supported ASC / Alliance Center for Astrophysical Thermonuclear Flashes at the University of Chicago.

REFERENCES

[1] L. Dagum and R. Menon, "OpenMP: An Industry-Standard API for Shared-Memory Programming," *IEEE Computational Science & Engineering*, vol. 5, no. 1, January-March 1998.

[2] J. Reinders, *Intel Threading Building Blocks: Outfitting C++ for Multi-Core Processor Parallelism*. O'Reilly Media, 2007.

[3] L. Smith and M. Bull, "Development of mixed mode mpi / openmp applications," *Scientific Programming*, vol. 9, no. 2-3/2001, pp. 83–98, 2001.

[4] C. Huang, O. Lawlor, and L. V. Kalé, "Adaptive MPI," in *Proceedings of the 16th International Workshop on Languages and Compilers for Parallel Computing (LCPC 2003), LNCS 2958*, College Station, Texas, October 2003, pp. 306–322.

[5] C. Huang, G. Zheng, S. Kumar, and L. V. Kalé, "Performance Evaluation of Adaptive MPI," in *Proceedings of ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming 2006*, March 2006.

[6] H. Kamal and A. Wagner, "Fg-mpi: Fine-grain mpi for multicore and clusters," in *The 11th IEEE International Workshop on Parallel and Distributed Scientific and Engineering Computing (PDESC)*. IEEE, Apr. 2010.

[7] A. Pant, H. Jafri, and V. Kindratenko, "Phoenix: A runtime environment for high performance computing on chip multiprocessors," *Parallel, Distributed, and Network-Based Processing, Euromicro Conference on*, vol. 0, pp. 119–126, 2009.

[8] H. Tang, K. Shen, and T. Yang, "Program transformation and runtime support for threaded MPI execution on shared-memory machines," *ACM Transactions on Programming Languages and Systems*, vol. 22, no. 4, pp. 673–700, 2000.

[9] G. Zheng, O. S. Lawlor, and L. V. Kalé, "Multiple flows of control in migratable parallel programs," in *2006 International Conference on Parallel Processing Workshops (ICPPW'06)*. Columbus, Ohio: IEEE Computer Society, August 2006, pp. 435–444.

[10] X. Jiao, G. Zheng, P. A. Alexander, M. T. Campbell, O. S. Lawlor, J. Norris, A. Haselbacher, and M. T. Heath, "A system integration framework for coupled multiphysics simulations," *Engineering with Computers*, vol. 22, no. 3, pp. 293–309, 2006.

[11] G. Weirs, V. Dwarkadas, T. Plewa, C. Tomkins, and M. Marr-Lyon, "Validating the Flash code: vortex-dominated flows," in *Astrophysics and Space Science*. Springer Netherlands, 2005, vol. 298, pp. 341–346.

[12] B. Fryxell *et al.*, "Flash: An adaptive mesh hydrodynamics code for modeling astrophysical thermonuclear flashes," *ApJS*, vol. 131, p. 273, Nov 2000.

[13] A. Dubey, L. B. Reid, and R. Fisher, "Introduction to flash 3.0, with application to supersonic turbulence," *Physica Scripta*, vol. T132, p. 014046, 2008. [Online]. Available: http://stacks.iop.org/1402-4896/T132/014046

[14] S. Negara, G. Zheng, K.-C. Pan, N. Negara, R. E. Johnson, L. V. Kale, and P. M. Ricker, "Automatic MPI to AMPI Program Transformation using Photran," in *3rd Workshop on Productivity and Performance (PROPER 2010)*, no. 10-14, Ischia/Naples/Italy, August 2010.

[15] J. Overbey, S. Xanthos, R. Johnson, and B. Foote, "Refactorings for Fortran and High-Performance Computing," in *Second International Workshop on Software Engineering for High Performance Computing System Applications*, May 2005.

[16] T. E. Foundation, "Eclipse - an open development platform," http://www.eclipse.org/.

[17] J. R. Levine, *Linkers and Loaders*. Morgan Kaufmann Publishers, 1999.

[18] U. Drepper, "ELF handling for thread-local storage," *Version 0.20, Red Hat Inc., Feb*, vol. 8, 2003.

[19] E. R. Rodrigues, P. O. A. Navaux, J. Panetta, and C. L. Mendes, "A new technique for data privatization in user-level threads and its use in parallel applications," in *ACM 25th Symposium On Applied Computing (SAC), Sierre, Switzerland*, 2010.

[20] H. Jin and R. F. V. der Wijngaart, "Performance characteristics of the multi-zone nas parallel benchmarks." in *Proceedings of the International Parallel and Distributed Processing Symposium (IPDPS)*, 2004.

[21] "SPAG," http://www.polyhedron.co.uk/spag0html.

[22] "ROSE," http://www.rosecompiler.org/.

[23] J. Mukherjee and S. Varadarajan, "Weaves: A framework for reconfigurable programming." *International Journal of Parallel Programming*, vol. 33, no. 2-3, pp. 279–305, 2005.