Automatic MPI to AMPI Program Transformation using Photran

Stas Negara[1], Gengbin Zheng[1], Kuo-Chuan Pan[2], Natasha Negara[3], Ralph E. Johnson[1], Laxmikant V. Kalé[1], and Paul M. Ricker[2]

[1] Department of Computer Science
University of Illinois at Urbana-Champaign
Urbana, IL 61801, USA
{snegara2,gzheng,rjohnson,kale}@illinois.edu
[2] Department of Astronomy
University of Illinois at Urbana-Champaign
Urbana, IL 61801, USA
{kpan2,pmricker}@illinois.edu
[3] Department of Computing Science
University of Alberta
Edmonton, Alberta T6G 2E8, Canada
negara@ualberta.ca

Abstract. Adaptive MPI, or AMPI, is an implementation of the Message Passing Interface (MPI) standard. AMPI benefits MPI applications with features such as dynamic load balancing, virtualization, and checkpointing. Because AMPI uses multiple user-level threads per physical core, global variables become an obstacle. It is thus necessary to convert MPI programs to AMPI by eliminating global variables. Manually removing the global variables in the program is tedious and error-prone. In this paper, we present a Photran-based tool that automates this task with a source-to-source transformation that supports Fortran. We evaluate our tool on the multi-zone NAS Benchmarks with AMPI. We also demonstrate the tool on a real-world large-scale FLASH code and present preliminary results of running FLASH on AMPI. Both results show significant performance improvement using AMPI. This demonstrates that the tool makes using AMPI easier and more productive.

1 Introduction

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. Most implementations of MPI are highly optimized for message passing performance, as efficient communication is one of the most important design goals of the MPI Standard.

However, the new generation of parallel applications are complex, involve simulation of dynamically varying systems, and use adaptive techniques such as multiple timestepping and adaptive refinements, as exemplified in [10, 1, 2]. The conventional implementations of the MPI standard tend to associate one

MPI process per processor, which limits their support of the dynamic nature of these applications, for example, load balancing is challenging, and must be handled by the application programmer. As a result, application performance and programmer productivity suffer.

One approach to decouple an MPI process from its OS process is to adopt a finer grained decomposition using light-weight threads. In this execution model, each MPI "process" is running in the context of a thread, and there are multiple threads running on a processor. One advantage of this approach is to allow automatic adaptive overlap of communication and computation, i.e., when one MPI "process" (or thread) is blocked to receive, another MPI thread on the same processor can be scheduled for running. Another advantage is that it allows different mapping of MPI threads to processors to take advantage of the multicore architectures. With sophisticated thread migration techniques [17], dynamic load balancing via migratable user-level threads can be supported at the run-time.

Adaptive MPI (AMPI) [6] exemplifies this approach. It is an adaptive implementation and extension of MPI with migratable threads, implemented on top of Charm++ [8]. With AMPI, computation is divided into a number V of virtual processors (VPs), and the runtime system maps these VPs onto P physical processors. Virtual processors are implemented as user-level threads. The number of VPs V and the number of physical processors P are independent, allowing the programmer to design a more natural expression of the algorithm. Dynamic load balancing is supported via thread migration. More recent work in FG-MPI [9] also follows this direction; however, it does not support thread migration and dynamic load balancing yet.

One major obstacle for switching a legacy MPI application to this multi-threaded MPI execution model is global (and static) variables. These variables in the MPI code cause no problem with traditional MPI implementations, since each process image contains a separate copy. However, they are not safe in the multi-threading paradigm. Therefore, the global variables in the MPI code need to be privatized to ensure thread safety. One approach is to manually remove global variables at source code level. However, this process is mechanical and sometimes cumbersome. Other more sophisticated approaches described in [17] enable run-time to automatically privatize global variables by analyzing GOT (Global Offset Table) in ELF (Executable and Linkable Format) executables. These approaches however do not handle static variables, and are limited to the platforms that support ELF executables.

In this paper, we present a compiler-based tool that automatically transforms a user program to run with MPI implementations that support the multi-threaded execution model. Since a significant number of legacy MPI applications are written in Fortran, we will mainly target Fortran language in this paper. Our tool employs Photran's [11] source-to-source compiler infrastructure for Fortran that we discuss in more details in Sect. 3. We will focus only on AMPI as the target MPI implementation for code transformation from now on. However, the transformed code is a legitimate Fortran MPI program with only a couple of AMPI specific extensions to support thread migration and load balancing. The

transformed program is portable and can run on any other MPI implementation as long as the AMPI thread migration feature is disabled.

2 MPI to AMPI Transformation

The design goal of our tool is to automatically transform Fortran 90 MPI programs to run on AMPI, and take full advantage of AMPI's load balancing capability. Two major tasks are: privatizing global variables as we already mentioned, and generating a pack/unpack subroutine for moving global data at load balancing time.

Fortran Global Variables Privatization. Global variables are those variables that can be accessed by more than one subprogram[4] (including several calls of the same subprogram) and are not passed as arguments of these subprograms. In Fortran 90, global variables are module variables, variables that appear in common blocks, and local variables that are `saved` (i.e. local variables that keep their values between subprogram calls like `static` variables in C).

Privatizing global variables means giving every MPI "process" its own copy of these global variables. This happens automatically in most MPI implementations, where each MPI process is a separate operating system process, while multithreaded AMPI requires that it be ensured by the programmer. One way to do this is, essentially, to put all of the global variables into a large object (a derived type in Fortran, or `struct` in C), and then to pass this object around between subprograms. Each AMPI thread can be given a different copy of this object. Figure 1 presents an example of privatizing a common block variable. Although this variable has two different names (`i` in `MyProg` and `v` in `PrintVal`), it is a single global variable in the original program.

A high level description of the global variables privatization procedure implemented by our tool is as follows. First, a new derived type is declared in a new module. This derived type contains a component for every global variable in the program. Every MPI process has its own instance of this type. A pointer to this type is passed as an argument to every subprogram. Throughout the program, every access to a global variable is replaced with an access to the corresponding field of the derived type. Finally, the declarations of global variables are removed from the program.

In certain cases sharing of global variables is not a problem. For example, a global variable that is a constant can not be modified by any process. A more subtle example is a variable that is assigned the same value by every process. In this scenario, it does not matter whether a process reads a value assigned by itself or by a different process. Another example is a global variable that is never read. Our tool does not privatize global variables that are constants, but employs a conservative approach for more complex cases to avoid performing a costly program analysis.

[4] We use *subprograms* to refer to both subroutines and functions in Fortran 90.

```
PROGRAM MyProg
  include 'mpif.h'
  INTEGER :: i, ierr
  COMMON /CB/ i
  CALL MPI_Init(ierr)
  i = 3
  CALL PrintVal
  CALL MPI_Finalize(ierr)
END PROGRAM


SUBROUTINE PrintVal
  INTEGER :: v
  COMMON /CB/ v
  print *, ``val='', v
END SUBROUTINE
```

```
MODULE GeneratedModule
  TYPE GeneratedType
    INTEGER :: f
  END TYPE GeneratedType
END MODULE GeneratedModule

SUBROUTINE MPI_Main
  USE GeneratedModule
  include 'mpif.h'
  INTEGER :: ierr
  TYPE(GeneratedType) :: p
  CALL MPI_Init(ierr)
  p%f = 3
  CALL PrintVal(p)
  CALL MPI_Finalize(ierr)
END SUBROUTINE MPI_Main

SUBROUTINE PrintVal(p)
  USE GeneratedModule
  TYPE(GeneratedType) :: p
  print *, ``val='', p%f
END SUBROUTINE
```

Fig. 1. Example of the code transformation that privatizes a common block variable. The original code of an MPI program is on the left; the transformed code, which can be executed on AMPI, is shown on the right.

Pack/Unpack Subroutine Generation. AMPI uses CHARM++ runtime system, and its automatic load balancing layer. Periodically, it collects load statistics and decides which threads (if any) need to be migrated to which processors. To implement such migrations, it is necessary to have our tool generate a pack/unpack subroutine, which is used to migrate the already privatized global variables between processors. AMPI already provides basic APIs to pack/unpack primitive data types (e.g. `INTEGER`, `REAL`, etc.) and one-dimensional fixed size arrays of primitive types. However, it does not handle multi-dimensional arrays or allocatable arrays. Our tool generates additional code for them, e.g., loops that iterate over some dimensions of an array, conditional statements that check whether arrays are allocated or not, etc. The current version of our tool does not generate code to migrate more complex types (e.g. linked lists).

3 Code Transformation Techniques

We implemented global variables privatization and its pack/unpack subroutine generation for Fortran 90 using the refactoring infrastructure in Photran, an Eclipse-based [4] Integrated Development Environment (IDE) for Fortran [11]. Although the tool is intended to be used as a preprocessor immediately before compilation (so the programmer never sees the transformed version of the program), currently it is accessible as a code transformation within the IDE.

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 pro-

```
1 SUBROUTINE MySub
2   REAL :: ar
3   ALLOCATABLE :: ar
4   DIMENSION :: ar(:,:)
5   SAVE :: ar
   ...
6 END SUBROUTINE
```

```
MODULE GeneratedModule
  TYPE GeneratedType
    REAL, ALLOCATABLE :: MySub_ar(:,:)
  END TYPE GeneratedType
END MODULE GeneratedModule
```

Fig. 2. Example of the global variable declaration, whose specifications span several statements (on the left), and the corresponding field declaration that combines all relevant information in a single statement (on the right).

gram'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. In the following section we present code analysis and transformation performed by our tool to privatize global variables and generate pack/unpack subroutine.

3.1 Code Analysis and Transformation

The overall code transformation performed by our tool proceeds in five steps:

1. Stubs are generated for the derived type and the module that contains this type. Our tool ensures that their names do not conflict or shadow names of other entities in the program.
2. Subprograms are processed. An extra parameter is added to each subprogram and each call site within its body. Components for `saved` variables are inserted into the derived type, accesses to these variables are replaced with accesses to the corresponding derived type components, and finally, the `saved` variables are deleted from the subprogram.
3. Common blocks are eliminated in a manner similar to `saved` local variables.
4. Module variables are eliminated similarly.
5. Pack/unpack subroutine is generated.

The first four steps privatize global variables, and the last step enables migration of MPI threads between processors in AMPI.

As a result of the code transformation, every global variable is replaced in the program's code with the corresponding field of the generated derived type. The type and specifications of the replacing field should be consistent with those of the replaced global variable. According to the Fortran standard, specifications of a variable may be defined by multiple specification statements. Our tool uses variable binding information provided by Photran infrastructure to collect the type and all specifications of a particular global variable, which are combined in a single declaration statement of the replacing field.

Figure 2 shows a `saved` variable `ar` declared in subroutine `MySub` and the corresponding field `MySub_ar` in the generated derived type. The type of variable `ar` is defined in the declaration statement at line 2. Lines 3-5 contain three specification statements that define variable `ar` as an allocatable two-dimensional `saved` array. All this information is integrated in a single declaration statement of field `MySub_ar`, where irrelevant (`SAVE`) and redundant (`DIMENSION`) specifications are filtered out.

```
SUBROUTINE MySub
  INTEGER, PARAMETER :: offset = 5
  INTEGER, PARAMETER :: x = offset + 10
  INTEGER, PARAMETER :: y = offset + 20
  INTEGER, PARAMETER :: total = x * y
  INTEGER :: boundary = y
  REAL, SAVE :: ar(total)
  ...
END SUBROUTINE
```

```
MODULE GeneratedModule
  INTEGER, PARAMETER :: CN_offset = 5
  INTEGER, PARAMETER :: CN_y = CN_offset + 20
  INTEGER, PARAMETER :: CN_x = CN_offset + 10
  INTEGER, PARAMETER :: CN_total = CN_x * CN_y
  TYPE GeneratedType
    INTEGER :: MySub_boundary = CN_y
    REAL :: MySub_ar(CN_total)
  END TYPE GeneratedType
END MODULE GeneratedModule
```

Fig. 3. Example of two global variable declarations that contain constants (on the left), and the corresponding generated module (on the right).

Declarations with Constants. Declarations of global variables may contain constants, e.g. a variable may be initialized with a constant, or dimensions of an array may be specified using constants. To make the declaration of the replacing field in the generated derived type consistent with the declaration of such global variable, our tool moves declarations of all constants contained in the variable's declaration to the generated module (i.e. the declarations of constants are deleted from the original code and placed in the generated module, and all accesses to the deleted constants in the original code are replaced with accesses to the corresponding constants from the generated module). These moved declarations of constants may contain some other constants, whose declarations also need to be moved to the generated module, and so on.

Figure 3 illustrates a code sample (on the left), where declarations of two global variables, `boundary`[5] and `ar`, contain constants `y` and `total` respectively. Declarations of constants `y` and `total` contain other constants. Moreover, the declaration of constant `total` contains constant `y`. To generate the correct code, we need to detect all constants that are immediately or transitively contained in the declarations of global variables `boundary` and `ar` and also, we need to establish an order of appearance of these declarations in the generated module such that if a declaration of some constant `C1` contains constant `C2`, then the declaration of constant `C2` comes before the declaration of constant `C1` in the generated module.

To achieve this goal, our tool constructs a graph, where nodes represent constants and edges represent "is contained in" relationship, i.e., there is an edge going from a node that represents constant `C1` to a node that represents constant `C2` if and only if constant `C1` is contained in the declaration of constant `C2`. The graph construction starts with the initial set of nodes for constants that are immediately contained in the declarations of global variables and proceeds recursively by adding nodes and edges for constants that are contained in the declarations of constants that are already present in the graph.

Figure 4 shows the constructed graph for the code sample on the left in Fig. 3. Double circled nodes represent the initial set of nodes. All constants, whose nodes appear in the graph, are moved to the generated module. The order of appearance of the declarations of these constants in the generated module is

[5] According to the Fortran standard, the local variable `boundary` is implicitly a `saved` variable because its declaration includes an initializer.

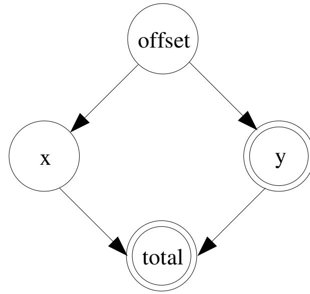


Fig. 4. Graph that represents "is contained in" relationship between constants of the code sample on the left in Fig. 3.

the topological order of the graph. For the graph in Fig. 4 this means that the declaration of constant `offset` is the first, it is followed by the declarations of constants `x` and `y` in any order, and finally comes the declaration of constant `total`. Figure 3 (on the right) presents the resulting generated module. Note that all constants, whose declarations are moved to the generated module, are renamed by prefixing their original names with "CN_". In real-world programs these constants may be from different subprograms and modules, and our tool ensures that they have unique names both in the generated module and in all places in the program, where they are accessed.

Derived Type Global Variables. A global variable may be of a derived type. The generated replacing field for this variable should be of the same derived type, therefore our tool moves the declaration of this derived type from the original code to the generated module. The moved derived type may contain fields, whose type is also derived, and, thus, needs to be moved to the generated module as well, and so on. In order to detect all derived types that have to be moved to the generated module and to establish the correct order of their appearance in it, our tool employs an approach similar to the one used for constants that are contained in the declarations of global variables.

To privatize global variables of derived types, our tool constructs a graph, where nodes represent derived types and edges represent "is used in" relationship, i.e., there is an edge going from a node that represents derived type `DT1` to a node that represents derived type `DT2` if and only if derived type `DT1` is used as a type of any field in the declaration of derived type `DT2`. The graph construction starts with the initial set of nodes for derived types of global variables and proceeds recursively by adding nodes and edges for derived types that are used in the declarations of derived types that are already present in the graph. All derived types, whose nodes appear in the graph, are moved to the generated module. The order of appearance of the declarations of these derived types in the generated module is the topological order of the constructed graph.

Global Fixed Size Arrays. In real-world scientific computation programs (like the one we use for our case study) there are many large fixed size arrays declared in different modules. If all these global arrays are placed in the gener-

```
MODULE MyMod
  INTEGER :: ar1(3)
  REAL :: ar2(5,5)
  REAL, ALLOCATABLE :: ar3(:)
END MODULE
```

Fig. 5. Example of a module that contains fixed size arrays.

```
MODULE GeneratedModule
  TYPE GeneratedType
    INTEGER, POINTER :: MyMod_ar1(:)
    REAL, POINTER :: MyMod_ar2(:,:)
    REAL, ALLOCATABLE :: MyMod_ar3(:)
  END TYPE GeneratedType
END MODULE GeneratedModule
```

```
SUBROUTINE GeneratedInit(p)
  USE GeneratedModule
  TYPE(GeneratedType) :: p
  ALLOCATE(p%MyMod_ar1(3))
  p%MyMod_ar1 = 0
  ALLOCATE(p%MyMod_ar2(5,5))
  p%MyMod_ar2 = 0.0
END SUBROUTINE
```

Fig. 6. The generated derived type (on the left) and initialization subroutine (on the right) for the module in Fig. 5.

ated 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. To avoid this problem, our tool transforms fixed size arrays into pointer arrays and generates 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.

Figure 5 shows a module that contains two fixed size arrays, `ar1` and `ar2`, and one allocatable array `ar3`. Figure 6 presents the generated derived type (on the left) and initialization subroutine (on the right) for the module in Fig. 5. Both fixed size module arrays are transformed to pointer arrays in the generated derived type. These pointer arrays are allocated and initialized in the generated initialization subroutine. The initialization to value `0` is required in order to be consistent with the original code, where these pointer arrays are fixed size, because Fortran compilers initialize fixed size arrays to value `0` by default.

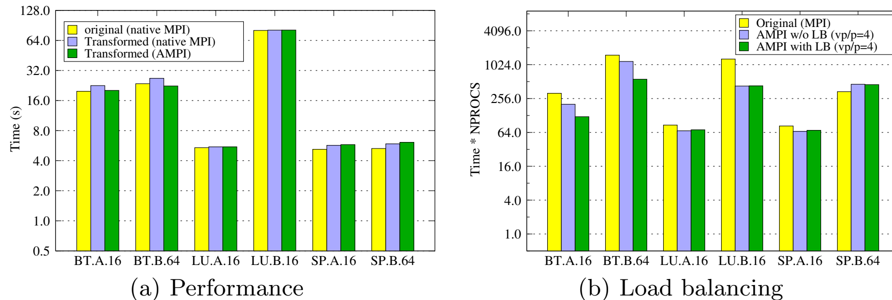
4 Evaluation

This section offers comparative evaluations between the original MPI code and the transformed version with AMPI. We use NAS Benchmarks and a real-world application FLASH for the study. By simply compiling the transformed code with AMPI, these programs benefit with the AMPI's dynamic load balancing.

4.1 Multi-zone NAS Benchmark

NAS Parallel Benchmark (NPB) is a well known parallel benchmark suite. Benchmarks in its Multi-Zone version [7], LU-MZ, SP-MZ and BT-MZ, which are written in Fortran, solve discretized versions of the unsteady, compressible Navier-Stokes equations in three spatial dimensions. Among these benchmarks, LU and SP are well-balanced, while BT is imbalanced application. 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.



(a) Performance (b) Load balancing

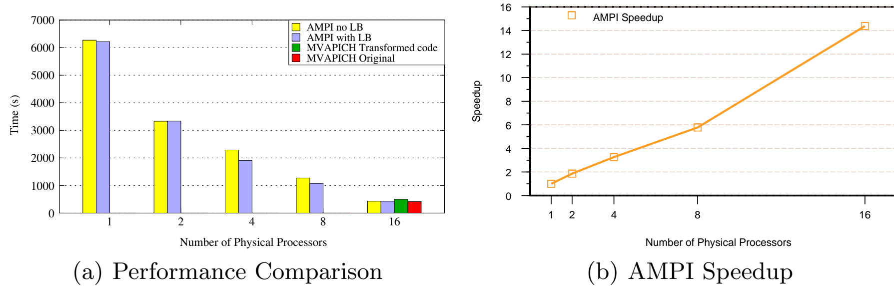
Fig. 7. Comparing NAS benchmarks time on a logarithmic scale (Queen Bee cluster).

We transformed the above mentioned three benchmarks, and evaluated the transformed code on the Queen Bee cluster at LSU. The native MPI we used for comparison is MVAPICH, which takes advantage of the Infiniband interconnect. Figure 7(a) illustrates the execution time of the original benchmarks on the native MPI, and the transformed benchmarks on the native MPI and AMPI. The X axis displays the name of a benchmark, the problem class, and the number of processors it was run on. The transformed code introduces some overhead that ranges from a fraction of one percent for LU.B.16 up to 14% for BT.A.16. Although the transformation overhead is the highest for both BT-MZ benchmarks, running on AMPI almost completely eliminates it. Note that in this comparison, we do not employ any specific benefits of AMPI, and the observed speed up is solely due to the efficient implementation of its communication layer.

Figure 7(b) compares the total resource consumption (execution time multiplied by the number of physical processors used) between the native MPI and AMPI. In AMPI runs, we mapped four MPI threads to a single physical processor, therefore reducing the number of physical processors used by a factor of four. The second bar shows the AMPI time without load balancing. The decrease in the total processor time demonstrates one of the benefits of using AMPI, i.e., adaptive overlapping of the computation/communication. The third bar shows the AMPI time with dynamic load balancing. We employed a greedy-based load balancer that is called once after the third simulation step. We see that BT-MZ benchmarks take advantage of both computation/communication overlap and load balancing, while LU.A.16, LU.B.16, and SP.A.16 benefit only from computation/communication overlap (since there is no load balance problem in both LU and SP). SP.B.64 is the only case that does not benefit from any advantages offered by AMPI.

4.2 Case Study – FLASH

We evaluated our tool on a large-scale project: FLASH, version 3 [3, 5, 2], which was developed by the University of Chicago. FLASH is a parallel, multi-



(a) Performance Comparison

(b) AMPI Speedup

Fig. 8. Sedov simulation performance (Abe cluster, NCSA).

dimensional code used to study astrophysical fluids. It is written mainly in Fortran 90 and parallelized using MPI. It is essentially a collection of code pieces, which are combined in different ways to produce different simulation problems, e.g., FLASH supports both uniform grid and a block-structured adaptive mesh refinement (AMR) grid based on the PARAMESH library.

We transformed and evaluated Sedov-Taylor explosion simulation problem [13], which is a common test problem for strong shocks and non-planar symmetry. The problem is set up using a delta function initial pressure perturbation in an uniform medium. We use 9 AMR levels and two-dimensional fluids for our tests. The experiments are run on the Abe cluster at NCSA.

Figure 8(a) compares the execution time of the transformed Sedov simulation on AMPI with and without load balancing. We vary the number of physical processors (X axis) from 1 to 16, while the number of virtual processors is 16 for all AMPI runs. The maximum benefit from load balancing is achieved for the execution on 4 physical processors (vp/p ratio 4) which is 16.8%. The two additional bars of the last group reflect the execution time of the original and the transformed Sedov simulation on the native MPI (MVAPICH). We were a little surprised to see that the code transformation incurs about 20% overhead compared to the original code when both running on MVAPICH. However, we see that the overhead is almost completely eliminated while running on AMPI, showing again that AMPI is an efficient implementation of MPI. The corresponding speedup of the simulation with AMPI is illustrated in Fig. 8(b).

Our investigation shows that the 20% overhead is almost entirely due to transforming global fixed size arrays to pointer arrays as described in Sect. 3, as it prevents Fortran compiler from performing aggressive optimizations. We elaborate a different approach that avoids dynamic allocation of the fixed size arrays. In this approach we keep fixed size arrays and distribute them across several derived types such that no derived type exceeds the maximum allowed size. Pointers to all these derived types are placed in a single derived type, which is used to pass around all previously global variables (including fixed size arrays). We plan to implement this approach in the next version of our tool.

Although our evaluation of Sedov simulation shows that code transformation incurs considerable overhead for this application, the results prove the usefulness of AMPI features. After we fix the overhead problem in the next version of our

tool, we believe that AMPI execution would demonstrate considerably better performance than the original MPI execution.

5 Related Work

Much work has been done for supporting multi-threaded programming in MPI to exploit overlapping of communication with computation. Hybrid programming model with MPI+OpenMP [14] approaches the problem by distributing OpenMP threads among MPI processes. Users need to specify thread private variables by explicitly using "private" OpenMP clauses. A compiler that supports OpenMP is required to compile such applications.

TMPI [16] uses multithreading for performance enhancement of multi-threaded MPI programs on shared memory machines. More recent work in FG-MPI [9] 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 source-to-source transformation implemented in our tool will benefit these MPI implementations as well.

SPAG [15] 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 [12] 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. 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.

6 Conclusions and Future Work

In this paper, we presented a Photran-based tool that automatically transforms legacy Fortran MPI applications to run on any MPI implementation that supports multi-threaded execution model. Specifically, we presented techniques to remove global variables in Fortran applications. We demonstrated the utility of the tool on AMPI, an MPI implementation that supports processor virtualization using user-level threads and dynamic load balancing with thread migration. We demonstrated the effectiveness of our tool on both NAS benchmarks and a real-world large scale FLASH application.

We plan to extend our tool such that it automatically generates the code for more complex types such as linked list in pack/unpack subroutine for load balancing. Also, we would like to minimize the computational overhead introduced in the transformed code. We are going to continue our performance evaluation. In particular, we would like to consider more complex and larger problems, which are expected to be inherently more load imbalanced, and, consequently, could benefit more from dynamic load balancing offered by AMPI.

Acknowledgments. This work was partially supported by the Institute for Advanced Computing Applications and Technologies (IACAT) at the University of Illinois at Urbana-Champaign. We used running time on Queen Bee cluster (LSU) and Abe cluster (NCSA), which is under TeraGrid allocation grant ASC050040N supported by NSF.

References

1. Bhatele, A., Kumar, S., Mei, C., Phillips, J.C., Zheng, G., Kale, L.V.: Overcoming scaling challenges in biomolecular simulations across multiple platforms. In: Proceedings of IEEE International Parallel and Distributed Processing Symposium 2008 (April 2008)
2. Dubey, A., Reid, L.B., Fisher, R.: Introduction to flash 3.0, with application to supersonic turbulence. Physica Scripta T132, 014046 (2008), `http://stacks.iop.org/1402-4896/T132/014046`
3. ASC Center for Astrophysical Thermonuclear Flashes. http://flash.uchicago.edu/website/home/
4. Foundation, T.E.: Eclipse - an open development platform, http://www.eclipse.org/
5. Fryxell, B., et al.: Flash: An adaptive mesh hydrodynamics code for modeling astrophysical thermonuclear flashes. ApJS 131, 273 (Nov 2000)
6. Huang, C., Lawlor, O., Kalé, L.V.: Adaptive MPI. In: Proceedings of the 16th International Workshop on Languages and Compilers for Parallel Computing (LCPC 2003), LNCS 2958. pp. 306–322. College Station, Texas (October 2003)
7. Jin, H., der Wijngaart, R.F.V.: Performance characteristics of the multi-zone nas parallel benchmarks. In: Proceedings of the International Parallel and Distributed Processing Symposium (IPDPS) (2004)
8. Kale, L.V., Zheng, G.: Charm++ and AMPI: Adaptive Runtime Strategies via Migratable Objects. In: Parashar, M. (ed.) Advanced Computational Infrastructures for Parallel and Distributed Applications, pp. 265–282. Wiley-Interscience (2009)
9. Kamal, H., Wagner, A.: 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)
10. Lawlor, O., Chakravorty, S., Wilmarth, T., Choudhury, N., Dooley, I., Zheng, G., Kale, L.: Parfum: A parallel framework for unstructured meshes for scalable dynamic physics applications. Engineering with Computers (September 2006)
11. Photran - An Integrated Development Environment for Fortran. http://www.eclipse.org/photran/
12. ROSE. http://www.rosecompiler.org/
13. Sedov, L.I.: Similarity and Dimensional Methods in Mechanics (1959)
14. Smith, L., Bull, M.: Development of mixed mode mpi / openmp applications. Scientific Programming 9(2-3/2001), 83–98
15. SPAG. http://www.polyhedron.co.uk/spag0html
16. Tang, H., Shen, K., Yang, T.: Program transformation and runtime support for threaded MPI execution on shared-memory machines. ACM Transactions on Programming Languages and Systems 22(4), 673–700 (2000)
17. Zheng, G., Lawlor, O.S., Kalé, L.V.: Multiple flows of control in migratable parallel programs. In: 2006 International Conference on Parallel Processing Workshops (ICPPW'06). pp. 435–444. Columbus, Ohio (August 2006)