Automatic MPI to AMPI Program Transformation

Stas Negara[1], Kuo-Chuan Pan[2], Gengbin Zheng[1], Natasha Negara[3],
Ralph E. Johnson[1], Laxmikant V. Kalé[1], 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 is an implementation of the Message Passing Interface (MPI) standard. AMPI benefits MPI programs with features such as dynamic load balancing, virtualization, and checkpointing. AMPI runs each MPI process in a user-level thread, therefore causing problems when an MPI program has global variables. Manually removing the global variables in the program is tedious and error-prone. In this paper, we present a tool that automates this task with a source-to-source transformation that supports Fortran. We evaluate our tool on a real-world large-scale FLASH code and present preliminary results of running FLASH on AMPI. Our results demonstrate that the tool makes it easier to use AMPI.

1. Introduction

Adaptive MPI [5] is an adaptive implementation and extension of MPI with migratable threads. AMPI includes a powerful run-time support system that takes advantage of the freedom of mapping virtual MPI processes (VPs) onto processors. With this run-time system, AMPI supports such features as automatic adaptive overlap of communication and computation and automatic load balancing. It can also support other features such as checkpointing [8] without additional user code, and the ability to shrink and expand the set of processors used by a job at runtime.

One obstacle for switching an MPI application to AMPI 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 AMPI's multi-threading paradigm. AMPI VPs are executed as user-level threads, many of which can run on one processor. Therefore, AMPI run-time needs to ensure thread safety of the global variables in the MPI code by privatizing the global variables. One approach is to manually remove global variables at source code level. However, this process is mechanical and sometimes cumbersome. In addition to this, there are several other changes required to transform the original MPI code to support dynamic load balancing in AMPI. For example, a pack/unpack subroutine needs to be written to serialize heap allocated user data so that it can be transfered to a different processor. Another change is to rename the main `PROGRAM` in Fortran to AMPI's `MPI_MAIN`, which is used as the entry point for an AMPI thread. In this paper, we will present a source-to-source transformation tool for Fortran programs that automatically does the above mentioned tasks by parsing the original source files and transforming them to run on AMPI.

2. MPI to AMPI Transformation

Our tool automates the global variables privatization, and other required changes for AMPI. It operates on MPI programs written in Fortran 90 programming language. In section 2.1 we describe code transformations required to privatize global variables in a Fortran 90 program. Section 2.2 presents a high level overview of how our tool is implemented.

```
PROGRAM MyProgram
  include 'mpif.h'
  INTEGER :: ierr
  CALL MPI_Init(ierr)
  CALL count_calls
  CALL count_calls
  CALL MPI_Finalize(ierr)
END PROGRAM MyProgram

SUBROUTINE count_calls
  INTEGER :: counter = 0
  counter = counter + 1
  print *, 'I was called ', counter, ' times.'
END SUBROUTINE count_calls
```

```
MODULE GeneratedModule
  TYPE GeneratedType
    INTEGER :: counter = 0
  END TYPE GeneratedType
END MODULE GeneratedModule

SUBROUTINE MPI_Main
  USE GeneratedModule
  include 'mpif.h'
  INTEGER :: ierr
  TYPE(GeneratedType) :: var
  CALL MPI_Init(ierr)
  CALL count_calls(var)
  CALL count_calls(var)
  CALL MPI_Finalize(ierr)
END SUBROUTINE MPI_Main

SUBROUTINE count_calls (var)
  USE GeneratedModule
  TYPE(GeneratedType) :: var
  var%counter = var%counter + 1
  print *, 'I was called ', var%counter, ' times.'
END SUBROUTINE count_calls
```

Figure 1. Example of the code transformation that privatizes the "saved" local variable `counter` of the subroutine `count_calls`. 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.

2.1. Fortran Global Variables Privatization

Global variables are those variables that can be accessed by more than one subprogram[1] (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 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 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 process can be given a different copy of this object. Figure 1 presents an example of privatizing the global variable `counter`, which is the only global variable in the original program (according to the Fortran standard, the local variable `counter` is implicitly a `save` variable because its declaration includes an initializer).

[1]There are two kinds of subprograms in Fortran 90: subroutines and functions. The main difference between them is that functions return values, while subroutines do not. Although we distinguish subroutines from functions in the implementation of our tool, the differences between them do not affect the concept of global variables. Therefore we refer to both these entities by the same word - subprograms.

A more detailed 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.

2.2. High Level Tool Overview

We implemented global variables privatization for Fortran using the refactoring infrastructure in Photran, an Eclipse-based [2] Integrated Development Environment (IDE) for Fortran [6]. Although the tool is intended to be used as a preprocessor immediately before compilation (so the programmer never sees the privatized version of the program), it is also accessible as a code transformation within the IDE. The privatization procedure proceeds in four passes:

1. Stubs are generated for the derived type and the module that contains this type. Their names should 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

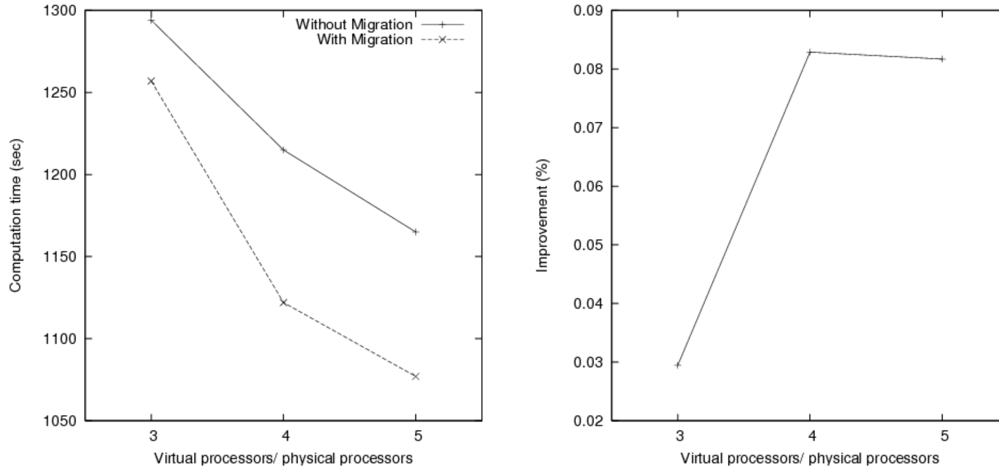

Figure 2. Left: Computation time versus the virtual processors to physical processors ratio. Right: The improvement by object migration versus the virtual processors to physical processors ratio.

its body. Components for `save` variables are inserted into the derived type, accesses to these variables are replaced with accesses to the corresponding derived type components, and finally, the `save` variables are deleted from the subprogram.

3. Common blocks are eliminated in a manner similar to `save` local variables.

4. Module variables are eliminated similarly.

5. Packing/unpacking subroutine is generated to enable migration of MPI processes between processors[2].

3. Evaluation

We evaluated our tool on the large-scale project FLASH [3]. FLASH is a parallel, multi-dimensional code used to study astrophysical systems, including compressible hydrodynamics, magneto-hydrodynamics (MHD), or special relativitistic hydrodynamics (RHD) [4, 1]. Many astrophysical environments are highly turbulent, e.g. star forming molecular clouds, accretion disks, etc., 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 issue becomes critical in recent computational astrophysics research, which makes it an ideal case for AMPI and its dynamic load balancing capability.

The FLASH code 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. For example, FLASH provides two types of grid structures that can be used for different geometries: a uniform grid and a block-structured adaptive mesh refinement (AMR) grid based on the PARAMESH library. We applied our tool on individual simulation problems (e.g. Sedov-Taylor), which are generated by a Python setup script that makes part of the FLASH distribution. Our tool works on a "pure" Fortran code, i.e. the code should not contain preprocessor directives. Therefore, before applying our transformation tool, we ran a preprocessor on the source code of the considered simulation problem. In particular, we transformed a simulation problem, Sedov-Taylor explosion, to AMPI and evaluated it on the Abe cluster at the National Center for Supercomputing Applications (NCSA).

3.1. The Sedov-Taylor Problem

The Sedov-Taylor explosion [7] 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. For the first test, only two-dimensional fluids are considered.

3.2. Preliminary Results

With the automatically transformed code that has all global varaibles removed, the program runs fine on AMPI with more than one virtual processor on a physical processor, and produces correct physics output. We further tested the performance by applying AMPI's load balancing. We varied the ratio of virtual processor to physical processor

[2]Current version of our tool does not generate code to migrate complex types (e.g. linked lists).

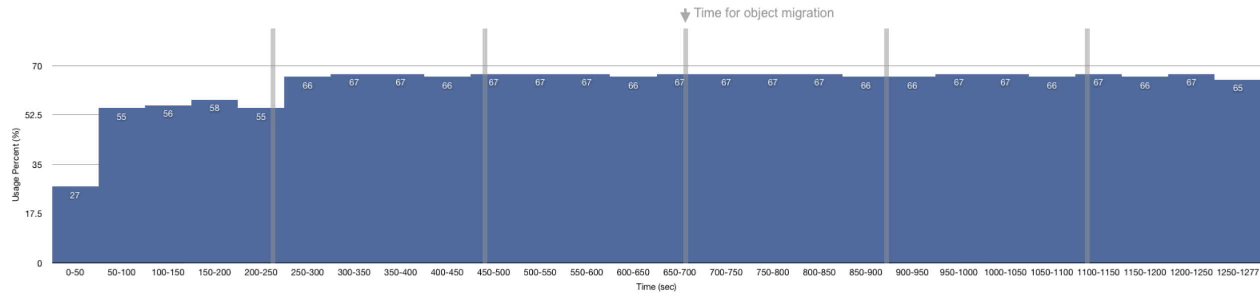

Figure 3. The overall CPU utilization over time. CPU utilization is averaged by every 50 s and traced by Projections. Object migration routine is called for every hundred steps in total 576 steps, and the corresponding time is marked by gray lines. The first load balancer is GreedyLB, others are RefineLB. The significant usage increment after the first load balancing is observed at 240 s.

and compared the performance with and without load balancing. The highest AMR level is set to 8 and run with 16 physical processors on NCSA Abe cluster for 483 steps. Figure 2 shows the computation time and the improvement with load balancing. In our experiments, we changed the virtual processors to physical processors ratio from 3 to 5 and monitored the computation time in the evolution stage of the Sedov-Taylor problem. A load balancer is called for every hundred steps. The first load balancer used is the Greedy load balancer which aggressively rebalances load by mapping AMPI threads to processors from scratch, and a refinement-based load balancer is used in the following load balancing steps. As shown in figure 2 (right plot), the improvement increased from 3% to 8% when the virtual processor to physical processor ratio is increased to 4.

Figure 3 shows the overall CPU utilization across all processors in one simulation. In this simulation on 16 physical processors, each processor has four AMPI virtual processors. The program calls the load balancing for every hundred steps. A significant improvements is observed at the time right after the first load balancing as shown in figure 3. The overal CPU utilization is increased from 55% to 66%, and remains stable throughout the rest of the run.

4. Future work

We plan to extend our tool such that it automatically generates the complete packing/unpacking subroutine for load balancing (currently it does not handle complex types like linked lists). 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. Additionally, we are going to employ more sophisticated load balancers that could further improve the resulting performance.

Acknowledgments

This work was partially supported by the Institute for Advanced Computing Applications and Technologies. FLASH was developed largely by the DOE-supported ASC/Alliances Center for Astrophysical Thermonuclear Flashes at the University of Chicago.

References

[1] A. Dubey, L. B. Reid, and R. Fisher. Introduction to flash 3.0, with application to supersonic turbulence. *Physica Scripta*, T132:014046, 2008.
[2] Eclipse Foundation. http://eclipse.org.
[3] ASC Center for Astrophysical Thermonuclear Flashes. http://flash.uchicago.edu/website/home/.
[4] B. Fryxell et al. Flash: An adaptive mesh hydrodynamics code for modeling astrophysical thermonuclear flashes. *ApJS*, 131:273, Nov 2000.
[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] Photran - An Integrated Development Environment for Fortran. http://www.eclipse.org/photran/.
[7] L. I. Sedov. *Similarity and Dimensional Methods in Mechanics*. 1959.
[8] G. Zheng, C. Huang, and L. V. Kalé. Performance evaluation of automatic checkpoint-based fault tolerance for ampi and charm++. *ACM SIGOPS Operating Systems Review: Operating and Runtime Systems for High-end Computing Systems*, 40(2), April 2006.