Predicting Execution Readiness of MPI Binaries with FEAM, a Framework for Efficient Application Migration

Karolina Sarnowska-Upton and Andrew Grimshaw
Department of Computer Science
University of Virginia
Charlottesville, VA

***Abstract*—As computational science becomes increasingly relevant for performing research, shared computing resources made accessible by cyberinfrastructures emerge as especially valuable for the majority of scientists who have not traditionally been the dominant users of such resources. However, in order to provide these newer computational scientists the opportunities to do great research, the ease-of-use of shared computing resources needs to be increased. In this paper, we present techniques that aim to make the migration to (and between) shared computing resources more efficient. Specifically, we focus on determining whether a computing site is a good fit for running an MPI binary. We present our methods and a Linux-based implementation called FEAM (a Framework for Efficient Application Migration). FEAM predicts execution readiness, resolves missing shared libraries, and composes site-specific configurations. We show that FEAM is more than 90% accurate at predicting execution readiness of MPI application binaries from the NAS Parallel and SPEC MPI2007 benchmark suites. In our evaluation, only half of the migrated binaries execute successfully at sites only configured with a matching MPI implementation. We show that by automatically resolving shared libraries requirements, FEAM is able to increase the number of successful executions by a third.**

***Index Terms*—execution prediction, environment configuration, MPI, migration**

I. Introduction

Although digital techniques permeate all disciplines, the majority of scientists have not traditionally been the users of shared computing resources provided by cyberinfrastructures like XSEDE [1]. Such resources have historically been used predominantly by participants in "big science," i.e., projects with large budgets, extensive collaborations, large data generation, complex instruments, and/or lengthy timescales. As computational science becomes increasingly relevant for performing research, shared computing resources emerge as especially valuable for the majority of scientists. However, in order to provide these newer computational scientists the opportunities to do great research, the ease-of-use of shared computing resources needs to be increased.

As scientists naturally want to maximize efficiency, they often turn to parallel computing to perform more computations in a given amount of time or to perform a given amount of computation in less time. When running computations in parallel, scientists generally use shared computing resources to get more computing capability, whether in terms of processing power or memory capacity. Typically, parallel computations employ the Message Passing Interface (MPI) standard, the de facto standard for running parallel programs on distributed memory systems. Indeed, many community applications use MPI to enable parallel execution. However, running an MPI application on shared computing resources requires that the execution environment be configured correctly (e.g. with dependencies related to MPI, libraries, and other software). This can be a tedious and time consuming process to do manually, especially for the majority of scientists who are not traditional users of shared computing resources.

In this paper, we present techniques that aim to make the migration to (and between) shared computing resources more efficient. Specifically, we focus on determining whether a computing site is a good fit for running an application. We present a framework that predicts whether an application will execute without being modified, thus enabling scientists to know how much effort will be required to get their applications running at new sites. In particular, we focus on the execution readiness of MPI application binaries. Migrating binaries instead of source code can be beneficial when optimal performance is not a concern. In this manner, scientists can avoid long compile times or compiling unfamiliar codes like community applications. Scientists can also gain quicker access to sites with more resources or sites experiencing shorter queuing delays.

Our work helps bridge the knowledge gap and lessen the learning curve encountered by new users of shared computing resources and users of new shared computing resources. We leverage techniques that may be familiar to more experienced computational scientists and system administrators to compose a framework that can aid any scientist in beginning the migration process. In this paper, we present our methods and a Linux-based implementation called FEAM (a **F**ramework for **E**fficient **A**pplication **M**igration). FEAM predicts execution readiness, resolves missing shared libraries, and composes site-specific configurations. We also present an evaluation of FEAM using MPI binaries from the NAS Parallel and MPI SPEC2007 benchmark suites. Our evaluation finds that FEAM can predict an application's execution readiness with more than 90% accuracy. Our evaluation also finds that by ensuring the presence of shared libraries at computing sites, FEAM increases the number of successful executions by more than 30% as compared to when a user only selects a compatible MPI implementation.

To the best of our knowledge, our implementation is the first to automatically predict execution readiness and compose site-specific configurations. In contrast, existing technologies assume that developers or users are able to fully describe application characteristics that are relevant for deployment. Other solutions also assume that developers or users will create configuration procedures for new computing sites.

The remainder of this paper is organized as follows. We begin with a discussion of related work in the next section. We then present our proposed methods and their implementation. In Section III, we describe our prediction model of execution readiness. In Section IV, we describe our resolution scheme for handling missing shared library dependencies. Then, in Section V, we explain how FEAM implements our methods. Finally, Section VI presents our evaluation of FEAM's effectiveness using binaries from the NAS Parallel and MPI SPEC2007 benchmark suites. Section VII discusses future work and Section VIII concludes.

II. Related Work

Various technologies exist to aid in the preparation of computing sites for the execution of applications. Commonly used HPC resource managers, like PBS [2], SGE [3], and SLURM [4], ensure applications are run on compute nodes with specified hardware characteristics like size of memory and number of nodes. They provide a mechanism with which users can deploy scripts to configure an environment further. Representation formats, such as SDD [5] and CDDLM [6], can be used to describe application information related to deployment. Frameworks, such as GLARE [7] and DistributedAnt [8], can be used to describe deployment instructions and perform them at new computing sites. There are also other technologies that utilize related methods while addressing different problems. Build tools, like Autoconf [9], gather information about execution environments to aid with resolving compile dependencies. Package managers provide tools for administrative management of software on computing systems [10,11,12]. Virtual machines mask the heterogeneity of execution environments while posing their own deployment limitations and incurring a performance overheard for many parallel applications [10]. However, unlike our solution, existing technologies do not automatically identify and describe the application information related to deploying an application at a new site. It is assumed that application developers or users will describe application characteristics as relevant for deployment. Similarly, unlike our resolution techniques, existing technologies do not automatically compose site-specific configuration instructions. It is assumed that application developers or users will develop the deployment procedures for new computing sites. Existing technologies cannot be utilized as efficiently by scientists trying to run their application binaries in new environments as the automated methods we present in this paper.

III. Execution Prediction Model

We have created a model to predict whether an MPI application is likely to execute at a new computing site. To make a prediction, our model considers the following questions: 1) Does the new computing site have a compatible and functioning MPI stack? 2) Does the new computing site meet the application's shared library requirements, including its C library version requirements? 3) Was the application compiled for a compatible hardware architecture? In this section, we explain how our model assesses whether these determinants are met. (For implementation details of our model, see Section V.)

Please note our use of the following terms:

- *Target site*: a computing site to where an application is being migrated
- *Execution readiness:* an application's ability to run successfully
- *Guaranteed execution site:* a site where an application runs successfully (typically this is the site from where an application is being migrated; it may be a site where an application was compiled)
- *MPI stack:* a combination of an MPI implementation, compiler, and interconnection network

A. MPI Stack Compatibility

The main determinant of our model considers whether a target site has a compatible and functioning MPI stack. Matching the MPI stack is critical when dealing with applications that use the MPI standard, as MPI is only an interface specification: the MPI standard specifies a library interface, not a specific library. An implementation of the MPI standard consists of a library (i.e. Open MPI, MPICH, MVAPICH) that can have various dependencies. Accordingly, an application compiled with a particular MPI implementation inherits the specific set of dependencies related to that MPI implementation type. Therefore, to predict execution of a dynamically-linked application at a target site, our model considers whether the necessary MPI library and its dependencies are available. Similarly, matching the associated compiler (i.e. GNU, Intel, or PGI) and, if applicable, the interconnection network (i.e. Ethernet or Infiniband) is important in order to match other dynamically-linked shared libraries.

We define a compatible MPI implementation as any implementation of the same type (i.e. an application compiled with Open MPI is only compatible with Open MPI and not MVAPICH or MPICH). In determining compatibility, our model does not consider the MPI implementation version because we have found that different version numbers do not necessarily imply incompatibility or compatibility. For example, we have found that an application compiled with Open MPI version 1.4 executes on a system using Open MPI version 1.3 in some instances but not in others. We do, however, consider version compatibly between shared library dependencies of MPI libraries. This compatibility is determined using the same methods as for any other shared library (described in Section III.B).

In addition to considering whether a compatible MPI stack exists, our model also takes into account whether the stack is functioning. For our purposes, an MPI stack is considered *functioning* if a basic MPI program is able execute using that stack. We have found that available MPI stacks may not be functioning due to misconfiguration. For example, an MPI implementation or compiler version may have been updated or a network may have been reconfigured; in both of these situations, a particular combination of MPI library, compiler, and interconnection network may no longer be compatible.

B. Shared Library Compatibility

Another key determinant of our prediction model is the presence of required shared libraries. If an application's shared library requirements are not met at a new computing site, the application will not execute. Thus, our model considers whether target sites are equipped with compatible versions of required shared libraries. Shared library compatibility is assessed based on library naming and version conventions as well as word-length (32 vs. 64-bit). Shared libraries are named using the following format convention that indicates the library name as well as the major and minor release version numbers:

lib<name>.so.<major_version>.<minor_version>

Compatibility is guaranteed for shared libraries with the same major version.

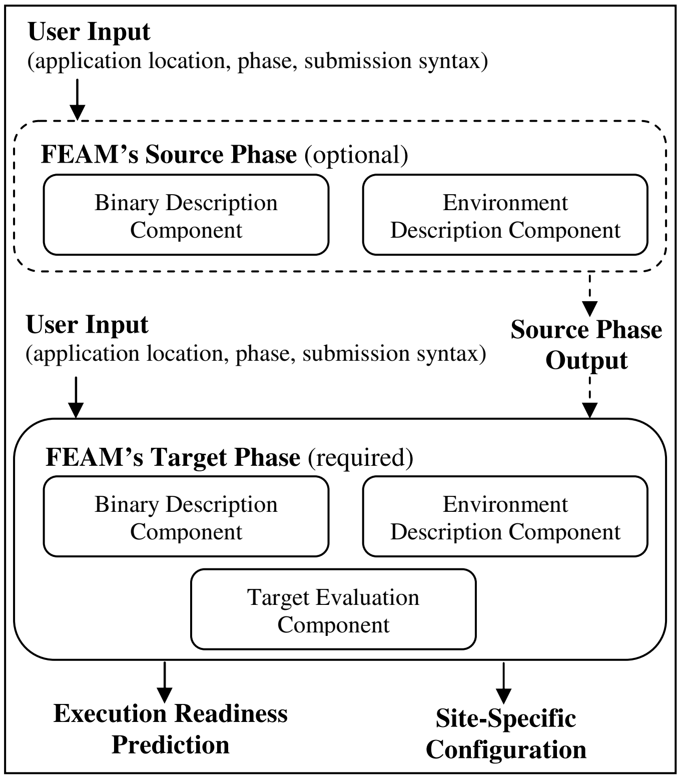
In assessing shared library compatibility, our model pays particular attention to the C library. As is well known, C library compatibility is a major determinant in the ability of an application to execute at a new computing site. Most applications, as well as their shared library dependencies, are dynamically-linked against a system's C library. Different versions of the C library exist at sites as a result of administrators updating their computer systems' C library versions at different times during the lifetime of a system. In our model, we assess C library compatibility based on the highest version of the C library that is used by an application. We call this an application's *required C library version*. Our model considers a target site's C library version to be compatible if it is equal to or greater than an application's required C library version.

C. Hardware Architecture Compatibility

The most basic determinant our model considers is hardware architecture compatibility. Our model evaluates compatibilities related to instruction set architectures (i.e. PPC, X86) and word-length to determine whether the format into which an application was compiled is executable at a target site.

IV. Resolution Scheme

In addition to a prediction model, we have created a scheme to determine whether some execution blocking issues can be resolved. In creating this scheme, we considered how the determinants of our execution prediction model could be influenced to enable execution. We recognized that, in general, execution readiness can be influenced by modifying an application or by modifying a target site. As our work assumes access to only an application binary, we were not interested in modifying the application. Thus, we investigated how a target site could be adapted to enable application execution. We found that resolving shared library requirements can often be accomplished in an unobtrusive manner. As a result, we created a scheme that focuses on the resolution of shared library requirements.



User Input
(application location, phase, submission syntax)
FEAM's Source Phase (optional)
Binary Description Component
Environment Description Component
User Input
(application location, phase, submission syntax)
Source Phase Output
FEAM's Target Phase (required)
Binary Description Component
Environment Description Component
Target Evaluation Component
Execution Readiness Prediction
Site-Specific Configuration

Fig. 1. **The Phases and Components of FEAM**
FEAM, our Framework for Efficient Application Migration, consists of two phases and three components. The source phase is optionally run a guaranteed execution site while the target phase must always be run at a target site. Running both phases enables our resolution methods and additional compatibility tests to be applied.

In creating our resolution scheme, we recognized that whenever there is access to an application's guaranteed execution site, there is also access to the application's required shared libraries. These shared libraries present at guaranteed execution sites can be copied for use at target sites. (Licensing issues are out of scope of our work.) Copying binaries, such as shared libraries, requires no special privileges or infrastructure. In contrast, influencing the other determinants of execution readiness, such as by installing a missing MPI stack or emulating a mismatched instruction set, would be beyond the scope of what most users are allowed to do at target sites.

There are two parts to our resolution scheme. First, we determine whether copies of missing shared libraries are available. Then, we determine whether the available library copies will execute at the target site. To assess the execution readiness of the library copies, we apply our execution prediction analysis. This may include recursively resolving missing shared libraries that a library copy requires. If our analysis predicts that the library copies are ready to execute at the target site, our resolution scheme concludes that the missing shared library requirements are resolvable.

V. Implementation

We have realized our prediction model and resolution scheme in FEAM, a Framework for Efficient Application Migration. FEAM is composed of three components that

1. ISA and file format of binary
2. Library name and version, if applicable
3. Required shared libraries, with copies and descriptions if applicable
4. C library version requirements
5. MPI stack, operating system, and C library version used to build binary

Fig. 2. **Information Gathered by the BDC**
The Binary Description Component (BDC) of FEAM gathers information about MPI application binaries.

execute in two phases (depicted in Figure 1). During a *target phase*, FEAM determines an application's execution readiness. During a preliminary but optional *source phase*, FEAM gathers information to enable resolution and support better prediction. FEAM source phases are carried out at guaranteed execution sites while target phases are carried out at target sites. As part of any phase, FEAM executes its *Binary Description Component* and *Environment Discovery Component*. The Binary Description Component gathers information about an application binary. The Environment Discovery Component gathers information about a computing environment. During a target phase, FEAM additionally executes the *Target Evaluation Component* to resolve execution issues and form the prediction about an application's execution readiness. After first providing an overview of the requirements for using FEAM, this section describes the implementation of each of its components. While our implementation techniques are relatively simple, their composition efficiently and automatically provides a determination of an application's execution readiness.

FEAM can be used to predict the execution readiness of applications only at sites with Unix-like operating systems. While our prediction model and resolution scheme could be implemented for any operating system type, we have developed the current implementation of FEAM for sites with Unix-like operating system utilities. To use FEAM, a user needs to provide minimal input. The user specifies the application binary location and which FEAM phase is to be run. If running a target phase, the user can also specify the location of a source phase output bundle, if applicable. Additionally, the user provides template submission scripts for running serial and parallel jobs. This submission format is the only site information that is not automatically determined by FEAM. The information that FEAM discovers about a target site, including the site-specific configuration details, are output to the user along with a prediction of execution readiness.

A. Binary Description

The Binary Description Component (BDC) of FEAM gathers information about an application binary. The five types of information collected by the BDC are listed in Figure 2. To gather this information, the BDT requires access to the application binary during a target or source phase. As the BDT is the only FEAM component that requires access to the application binary, running both phases of FEAM provides the additional benefit of not needing the application binary to be present at the target site. In this subsection, we describe how the BDT gathers the information needed to create an application description.

TABLE I. Identifying Libraries of MPI Implementations

MPI Implementation	Library Dependencies
MVAPICH2	libmpich/libmpichf90, libibverbs, libibumad
Open MPI	libnsl, libutil
MPICH2	libmpich/libmpichf90 (and not other identifiers)

The BDT extracts the first four types of information listed in Figure 2 using the GNU Binary Utility [14] objdump. We call the objdump utility with the "-p" flag to view information about a binary's file format (i.e. the executable and linkable format ELF). The header of the resulting description specifies the file format of the binary as well as the instruction set architecture and the number of bits for which the binary was compiled. Under the description's "Dynamic Section", we extract the "NEEDED" components to create a list of shared libraries the application was linked against. We also consult the "Dynamic Section" to extract naming and version information for binaries that are shared libraries. Finally, using information from the "Version Definitions" and "Version References" sections, we find the newest version of the C library that is listed as being used by the application.

The BDT verifies an application's shared library dependencies with the Unix utility ldd. We call ldd with the -v command-line option to view a list of the shared libraries an application has been linked against along with their locations in the local file system and version information. We do not rely on ldd for shared library information, as we have found that in some cases this utility does not recognize an application as being dynamically linked.

To gather information about the operating system and C library version used to create an application binary, the BDT uses the GNU Binary Utility readelf. This utility provides information about ELF files (the standard binary format for Unix and Unix-like system on x86). We call readelf with the "-p .comment" flag to display a binary's comment header section. This optional section may contain compiler and linker specific version control information with operating system and C library version details.

To identify the MPI stack used to compile an application, we examine an application's shared library dependencies. Our identification scheme takes advantage of the fact that MPI is not a link-level specification. Rather, the MPI standard specifies a library interface; consequently, implementations of the standard have different link-level dependencies. We have created an identification scheme for the three dominant open source MPI implementations (Open MPI [15], MPICH2 [16], and MVAPICH2 [17]). The shared libraries that we use as MPI implementation identifiers are listed in Table I.

When running during a source phase, the BDC also gathers a description and copy of all of the shared libraries against which an application has been linked. A description of each shared library is gathered in the same manner as the description for any application binary. A copy of each shared library (except for the C library) is then made by locating each shared library in the local file system. If the ldd utility is unable to provide this location information, we search for each shared library using three possible methods. If available, the

1. ISA format
2. Operating system
3. C library version
4. Available or currently loaded MPI stacks
5. Missing shared libraries

Fig. 3. **Information Gathered by the EDC**
The Environment Description Component (EDC) of FEAM gathers information about a computing site.

locate utility can be used to reveal the locations of files with matching names. The find utility can be used to search common library locations as well as locations set in the LD_LIBRARY_PATH environment variable. Alternately, a locally compiled "Hello World" program can be created to reveal commonly linked against shared libraries.

B. Environment Discovery

The Environment Discovery Component (EDC) gathers information about a computing environment. The five types of information gathered by the EDC are listed in Figure 3. The BDT gathers this information during target and source phases to form descriptions of target sites and guaranteed execution sites. In this subsection, we describe how the EDC gathers the information needed to create an environment description.

To determine a system's instruction set architecture format, the EDC uses the Linux utility uname. We call uname with the "-p" flag to view processor related information. During target phases, we also gather information about the operating system to provide the user with additional information about an environment. We examine standard files with system information under /proc and /etc to determine which Linux distribution is running on the system. We consult the /proc/version file to determine the OS type and version information. We confirm this information by examining files under /etc/*release.

To determine the version of a system's C library, the EDC uses the C library's application programming interface. We call the **gnu_get_libc_version** function to get a list of available C library versions. Alternately, we determine the version by invoking the C library binary. The binary is located for invocation via the same methods the BDC uses to locate shared libraries.

To determine what MPI stacks are available at a computing site, the EDC consults user-environment management tools. These tools support the discovery of packages and help manage the shell environment. We search for the presence of Environment Modules [18] or SoftEnv [19] configuration files to assess if these user-environment management tools are present. If one of these tools is located, we use its search mechanism to locate MPI implementations and compiler combinations (e.g. module avail for Environment Modules). If no user-environment management tools are found, we search for MPI implementation identifying libraries (listed in Table I) using the same methods as used by the BDC to locate shared libraries. We also search for commonly used wrappers for compiling MPI programs (e.g. mpicc).

To determine what MPI stacks are currently accessible in a computing environment, the EDC examines environment settings. If user-environment management tools are present, we use the corresponding mechanisms to reveal these setting. For example, if Environment Modules are present, we use the module list command to get a listing of what MPI implementation and compiler the shell is configured to access. Alternately, we search for MPI implementations accessible via the PATH and LD_LIBRARY_PATH environment variables. We can also determine information about the currently accessible MPI stack by identifying which commonly used wrappers for compiling MPI programs are currently accessible at the site.

When running during a source phase, the EDC additionally creates two MPI "Hello World" programs, one compiled in C and the other in Fortran. We compile these programs using the application's required MPI stack.

When running during a target phase, the EDC additionally identifies whether any required shared libraries are missing at the target site. If the application binary is present at the target site, we use the ldd tool to get a list of missing shared libraries. If the application binary is not present at the target site, or if the ldd tool is not working, we perform a search using the same methods as used by the BDC to locate shared libraries.

C. Target Evaluation

The Target Evaluation Component (TEC) evaluates the execution readiness of an application during target phases. To determine execution readiness, the TEC investigates the determinants outlined by our prediction model. If following a source phase, the TEC also applies our resolution scheme. In this subsection, we describe the process the TEC follows to arrive at a final prediction.

The TEC first determines whether the hardware architecture and C library version requirements are met. To evaluate compatibility—as outlined in our prediction model in Section III—we analyze the application description created by the BDC and the environment descriptions created by the EDC.

The TEC next evaluates MPI stack compatibility. The TEC can search for a compatible MPI stack or for a specific MPI stack requested by the user. For each matching MPI stack, we compile and run test MPI programs to determine if the stack is functioning as defined by our prediction model. If a source phase has previously occurred, we also run the test MPI programs generated at the guaranteed execution site. The success of these tests further shows compatibility between the selected MPI stack and an MPI stack used to run the application. Running these test MPI programs requires knowing the execution command that corresponds to the selected MPI stack. Our methods by default use the *mpiexec* command unless otherwise specified by users in the template submission format description.

Lastly, the TEC investigates whether an application's shared library requirements are met. The EDC has already determined if any required shared libraries are missing at the target site. If a source phase has previously occurred, we attempt to resolve any missing library requirements by applying our resolution scheme. We make copies of missing shared libraries that have been determined to be ready to execute at the target site accessible for use at runtime by adding their location to the PATH environment variable.

The TEC makes the final prediction that an application is ready for execution at a target site if all of our prediction model determinants are evaluated to be satisfied. FEAM outputs the prediction along with site-specific configuration details to the user. The output includes an explanation of general site information and any detected problems. Additionally, the user is provided with scripts to automatically set the determined configurations at runtime.

VI. Evaluation

To evaluate our prediction model and resolution scheme, we used FEAM to predict the execution readiness of MPI application binaries from two benchmark suites across five computing sites. To determine the effectiveness of our prediction model, we compared FEAM's prediction of execution readiness with actual execution results, i.e., whether the applications ran or not. To evaluate the impact of our resolution scheme, we calculated the increase in successful executions enabled by FEAM. In this section, before discussing the evaluation results, we discuss the binaries and sites used for our tests as well as how our tests were set up.

A. Test Set Description

For our evaluation, we used MPI applications from two benchmark suites: the NAS Parallel Benchmarks [17] and SPEC MPI2007 [18]. The NAS Parallel Benchmarks (NPB) suite consists of applications derived from computational fluid dynamics. SPEC MPI2007 benchmark suite was developed from native MPI-parallel end-user applications. We used version 2.4 of the MPI reference implementation of the NPB suite and version 2.0 of the SPEC MPI2007 benchmark suite for our evaluation.

We evaluated the execution readiness of the selected MPI applications at five computing environments. These target sites were chosen such that a diverse test set was created in terms of the operating system, hardware architecture, network interconnect, and MPI implementation. Three of the chosen test sites are state-of-the-art high performance computing systems available to researchers via the national XSEDE infrastructure [22]. They represent the main types of systems architectures available at the national supercomputing centers: symmetric multiprocessing (SMP), massively parallel processing (MPP), and hybrid CPU/GPU systems. The other two test sites are mid-size university clusters. One is part of the FutureGrid Project [23] test-bed while the other is a University of Virginia resource [24]. The test sites run different versions of three Linux-based operating systems and the C library. The sites support three open source MPI implementations. Open MPI is available at five sites, MVAPICH2 is available at four sites, and MPICH2 is available at two sites. Each MPI implementation is associated with GNU, Intel, and/or PGI compilers for C and/or Fortran. Table II details these characteristics for each computing site and lists the MPI stack combinations that we used to create application binaries.

To create the MPI application binaries for our tests, we compiled the selected MPI applications with multiple MPI stacks at each target site. Our final binary test set consisted of

TABLE II. Target Site Characteristics

Computing Site (Type - CPUs)	**Operating System**	**C Library & Compiler Versions**	**Utilized MPI Stacks: MPI Types & Versions (Compilers i:Intel, g:GNU, p:PGI)**
XSEDE Ranger, Texas Advanced Computing Center (MPP – 62,976)	CentOS 4.9	LibC v2.3.4, GNU CC v3.4.6, Intel v10.1	Open MPI v1.3 (i/g/p) MVAPICH2 v1.2 (i/g/p)
XSEDE Forge, National Center for Supercomputing Applications (Hybrid - 576)	Red Hat Enterprise Linux Server 6.1	LibC v2.12 GNU CC v4.4.5 Intel v12	Open MPI v1.4 (g/i) MVAPICH2 v1.7rcl (i)
XSEDE Blacklight, Pittsburg Supercomputing Center (SMP – 4,096)	SUSE Linux Enterprise Server 11	LibC v2.11.1, GNU CC v4.4.3, Intel v11.1	Open MPI v1.4 (i/g)
FutureGrid India, Indiana University (Cluster - 920)	Red Hat Enterprise Linux Server 5.6	LibC v2.5, GNU CC v4.1.2, Intel v11.1	Open MPI v1.4 (i/g) MVAPICH2 v1.7a2 (i/g) MPICH2 v1.4 (i/g)
ITS Fir, University of Virginia (Cluster - 1,496)	CentOS 5.6	LibC v2.5, GNU CC v4.1.2, Intel v12	Open MPI v1.4 (i/g/p) MVAPICH2 v1.7a (i/g/p) MPICH2 v1.3 (i/g/p)

a subset of the benchmarks suites as some benchmark applications would not compile with certain MPI stacks combinations while others would not run at the site where they were compiled. From the NPB suite, our test set consisted of four kernels (integer sort, embarrassingly parallel, conjugate gradient, and multi-grid on a sequence of meshes) as well as three pseudo applications (block tri-diagonal solver, scalar penta-diagonal solver, and lower-upper Gauss-Seidel solver). From the SPEC MPI2007 benchmark suite, our test set consisted of a quantum chromodynamics code (104.milc), two computation fluid dynamics codes (107.leslie3d and 115.fds4), a parallel ray tracing code (122.tachyon), a molecular dynamics simulation code (126.lammps), a weather prediction code (127.GAPgeofem), and a 3D Eulerian hydrodynamics code (129.tera_tf). In total, our binary test set was comprised of 110 NPB and 147 SPEC MPI2007 binaries.

B. Methodology

To evaluate the effectiveness of our prediction model, we compared FEAM's predictions with actual execution results. Initially, we migrated each MPI application binary to all target sites where the binary had not been compiled. We found that our methods were 100% accurate at assessing whether a matching MPI implementation was available for all target sites. Thus, we chose to restrict our evaluation to sites that had a matching MPI implementation. For example, for binaries compiled for MVAPICH2 on Ranger, the accuracy of our prediction was calculated only on Forge, India, and Fir. The execution readiness of applications at such sites is more relevant as, without a matching MPI implementation, the other sites have no potential for successful execution. However, in choosing to focus on this subset, we ignore accurate

TABLE III. ACCURACY OF PREDICTION MODEL

Basic Prediction		Extended Prediction	
NAS	SPEC	NAS	SPEC
94%	92%	99%	93%

TABLE IV. IMPACT OF RESOLUTION MODEL

Actual Execution Result				Increase in Successful Executions due to Resolution	
Before Resolution		After Resolution			
NAS	SPEC	NAS	SPEC	NAS	SPEC
58%	47%	78%	66%	33%	39%

predictions and, as a result, present lower prediction accuracy results than if all target sites had been considered.

Our evaluation of the effectiveness of our prediction model also distinguishes where information is gathered to form a prediction. We present *basic prediction* results formed by only running FEAM's required target phase at a target site. We present *extended prediction* results formed by including information from running FEAM's optional source phase at a guaranteed execution site. In this way, our evaluation differentiates the effectiveness of our methods for instances when users do not have access to or do not want to access guaranteed execution sites. Such a situation could, in particular, be relevant to users of community application binaries. Table III presents the accuracy results of our prediction model for basic and extended predictions on our test set, broken down by benchmark suite.

To evaluate the impact of our resolution scheme, we calculated the increase in successful executions enabled by FEAM. As with our prediction model evaluation, our resolution scheme evaluation focused on target sites with matching MPI implementations. We measured the number of successful executions before applying FEAM (E_{pre}) and the number of successful executions after applying the resolution related configurations composed by FEAM (E_{post}). Finally, we calculated the increase in successful executions ($E_{increase}$) in relation to the initial number of successful executions:

$$E_{increase} = (E_{post} - E_{pre}) / E_{pre}$$

Table IV presents these three sets of calculations for each benchmark suite as a percentage of the total number of executions.

C. Results Analysis

To put the evaluation results of our prediction model and resolution scheme into perspective, we first consider the execution results that we gathered without applying FEAM. We found that only around half (58% of the NAS binaries and 47% of the SPEC binaries) of the applications ran when the only configuration performed was the selection of a matching MPI implementation. These results underline the importance of considering more than just available MPI implementations when choosing execution sites. Again, these percentages only consider target sites where a matching MPI implementation was available. If all possible target sites had been considered, the percentage of successful executions would be much lower. Our results also confirm the importance of dealing with missing shared libraries to ensure the execution of an application binary without recompilation. We found that missing shared libraries caused more than half of the execution failures. (The remaining failures were due to C library version requirements, floating point exceptions, and system errors.)

Upon analysis of our evaluation results, we found that our model was able to recognize and correctly predict the vast majority of execution failures. FEAM's predictions were over 90% accurate. Due to additional compatibility tests incorporated from information gathered during FEAM's source phases, our extended predictions were even more accurate than our basic predictions (99% vs. 94% for NAS binaries and 93% vs. 92% for SPEC binaries). For example, by running MPI test programs compiled at guaranteed execution sites, we were able to detect floating point errors and application binary interface (ABI) incompatibilities in shared libraries. We found that less than 10% of the time our model incorrectly predicted success as it was unable to recognize failures due to system errors, such as failed MPI daemon spawning or time-outs due to communication errors.

As for our resolution scheme, our analysis revealed that it enabled execution for about half of the binaries that would have otherwise failed due to missing shared libraries. In other words, using FEAM resulted in around a third more successful executions overall (33% for the NAS binaries and 39% for the SPEC binaries). The other half of the missing library failures could not be resolved mainly due to incompatibility issues. For example, the shared libraries copies gathered at guaranteed execution sites required incompatible C library versions and used incompatible ABIs. The remaining resolution attempts failed due to system errors.

To additionally asses the cost of using FEAM, we measured how many CPU hours were required to run our prediction model as well as how much extra space was required to apply our resolution scheme. These quantities are relevant for computing sites that charge for compute nodes usage and limit storage space. We found that FEAM's source and target phases completed in less than five minutes. We measured that FEAM used on average 45MB of disk space when counting the size of a bundle of all shared libraries used by our test set. Overall, we found that executing FEAM for our evaluation test set required a small amount of time and disk space.

VII. FUTURE WORK

The general direction of our future work will be to develop more methods for efficiently migrating MPI applications to new environments. Our next focus is on techniques to aid the migration of MPI application source code. We are also in the process of carrying out a study that measures the amount of time users spend and the types of tasks users perform when migrating to new environments [25]. Eventually, we will evaluate the effectiveness of our techniques on a mixture of non-benchmark codes developed by individuals and communities.

VIII. CONCLUSIONS

This paper described our efforts to help make computational infrastructures more usable and accessible to any scientists who want to employ computation in doing their research. We described a model that predicts whether MPI application binaries will execute at new sites and a scheme that increases the likelihood binaries will execute by resolving shared library requirements. We presented a Linux-based implementation of our techniques called FEAM, a Framework for Efficient Application Migration. We evaluated FEAM across five computing sites with applications from NAS Parallel and MPI SPEC2007 benchmark suites compiled using three open-source MPI implementations and three compilers. Our evaluation found that our prediction model was more than 90% accurate, and that our resolution scheme enabled about a third more successful executions.

There are various use cases for employing FEAM. FEAM can be a preliminary probe for evaluating multiple target sites by analyzing sites for execution readiness quickly without requiring the application to be present at each site. FEAM can gather and document basic site characteristics, such as available MPI stacks. FEAM can also detect and document basic execution issues. When scientists are dealing with applications they did not create, FEAM can identify basic requirements, such as the MPI stack and shared library dependencies. FEAM can quickly enable execution that would otherwise be blocked by missing shared libraries without requiring scientists to be familiar with the management of shared objects. Additionally, FEAM, by providing documentation of the analysis process, can teach interested scientists what to look for at new computing sites when assessing execution readiness.

FEAM executes with minimal input from the user, runs quickly, and requires little space. It relieves the user of manually parsing various environment configurations to find out if a shared computing resource is a good match for execution. For scientists who do not have much experience, time, or support to explore new computing sites or compile MPI applications, FEAM provides an efficient automated tool for facilitating the migration process.

ACKNOWLEDGMENT

This work used the Extreme Science and Engineering Discovery Environment (XSEDE), which is supported by National Science Foundation grant number OCI-1053575. This work also used resources supported in part by the National Science Foundation under Grant No. 0910812 to Indiana University for “FutureGrid: An Experimental, High-Performance Grid Testbed.”

REFERENCES

[1] XSEDE: Extreme Science and Engineering Discovery Environment. https://www.xsede.org/.

[2] L. Bayucan, *et al*, “Portable Batch System External Reference Specification”, MRJ Technology Solutions, May 1999.

[3] W. Gentzsch, “Sun Grid Engine: towards creating a compute power grid”, Proceedings of the First IEEE/ACM International Symposium on Cluster Computing and the Grid, 2001, pp.35-36.

[4] SLURM: Simple Linux Utility for Resource Management, http://www.llnl.gov/linux/slurm/slurm_design.pdf

[5] J. McCarthy and B. Miller. Solution Deployment Descriptor (SDD), Part 1: An emerging standard for deployment artifacts. IBM DeveloperWorks. 2008.

[6] A. Dantas, et al. “Using web services for configuration and deployment according to the CDDLM standard,” International Conference on Web Services, 2006, pp. 951-954.

[7] M. Siddiqui, et al. “GLARE: A grid activity registration, deployment and provisioning framework,” Proceedings of the ACM/IEEE SC 2005 Conference, 2005, pp. 52 - 64.

[8] W. Goscinski and D. Abramson. “Distributed Ant: a system to support application deployment in the grid,” Proceedings of the Fifth IEEE/ACM International Workshop on Grid Computing, 2004, pp. 436-443.

[9] Autoconf - GNU Project. http://www.gnu.org/software/autoconf/.

[10] E.C. Bailey. Maximum RPM: Taking the Red Hat Package Manager to the Limit. Red Hat, Inc., 2000.

[11] Yum: Yellow Dog Update Modifier. http://yum.baseurl.org/.

[12] Smart Package Manager. http://labix.org/smart/.

[13] C. Xu, Y. Bai, and C. Luo. “Performance envalutation of parallel programming in virtual machine environment,” Sixth IFIP International Conference on Network and Parallel Computing, 2009, pp. 140-147.

[14] GNU Binutils. http://www.gnu.org/software/binutils/.

[15] OpenMPI: Open Source High Performance Computing. http://www.open-mpi.org/.

[16] MPICH2: High Performance and Highly Portable MPI. http://www.mcs.anl.gov/research/projects/mpich2/.

[17] MVAPICH2: MPI-2 over InfiniBand, 10GigE/iWARP and RoCE. http://mvapich.cse.ohio-state.edu/overview/mvapich2/.

[18] Modules – Software Environment Management. http://modules.sourceforge .net/.

[19] Msys - The MCS Systems Administration Toolkit. http://www.mcs.anl.gov/hs/software /systems/msys/.

[20] R.F. Van der Wijngaart. *NAS Parallel Benchmarks Version 2.4.* 2002.

[21] M.S. Müller, et al. "SPEC MPI2007 - an application benchmark suite for parallel systems using MPI." *Concurrency and Computation: Practice and Experience* Vol. 22, Issue 2 (2010): 191-205.

[22] XSEDE Resources Overview. https://www.xsede.org/resources/overview

[23] FutureGrid: A Distributed Testbed for Clouds, Grids, and HPC. https://portal.futuregrid.org/.

[24] University of Virginia Alliance for Computation Science and Engineering: Resources. http://www.uvacse.virginia.edu/resources/.

[25] Quantifying User Effort to Migrate MPI Applications: A Research Study. http://www.cs.virginia.edu/~kas9ud/study/.