Software Portability and the CM-5

James D. Mooney
Dept. of Statistics and Computer Science
West Virginia University
PO BOX 6330
MORGANTOWN WV 26506-6330
jdm@cs.wvu.edu

Abstract

Portability is a desirable attribute of most programs. If programs can be easily adapted to various contemporary computing environments and to the new, improved environments of the future, their usefulness is broadened and software costs can be substantially reduced. This problem is especially acute for high-performance parallel software, because architectures evolve rapidly and are often radically different; because performance is critical; and because there is a huge body of existing software to be ported.

This talk provides an overview of portability issues and strategies as they relate to high-performance computing (HPC) software in general and to the CM-5 in particular. Portability can be greatly enhanced by following suitable practices both when adapting existing software and when developing new software. We will examine some issues and strategies for porting code from other environments to parallel architectures such as the CM-5, and for making CM-5 programs more easily adaptable to other platforms in the future.

1. Introduction

Portability is an attribute for a software unit (e.g., program, component, system) which enables it to be “easily” adapted and moved to a new computing environment. If a software unit is portable, it should be possible to implement it in the new environment with limited (though rarely zero) cost and effort. Furthermore, the new implementation should provide equivalent behavior to the original (though not always identical) with satisfactory performance.

In principle, this attribute is almost always desirable. The investment represented by a software unit, especially a large and complex system, is most effectively returned if the software will have long and widespread use. There are many popular computing environments, both general and specialized, and these environments evolve rapidly. Long and widespread use, then, may

---

1 This work was funded in part by the National Science Foundation under grant CCR-9204461, and in part by the National Science Foundation and the State of West Virginia under the NSF WV EPSCoR program.
require adapting a program to a variety of environments, and continuing to adapt as these environments evolve. Portability is the property which enables this adaptation.

Despite the acknowledged need for better portability, published research on portability issues has been sparse. A variety of porting experiences have been reported, leading to many types of anecdotal advice (e.g. [LeCarme et al 89]). There is, however, no systematic framework to guide developers in maximizing portability in the general software development process. To help fill this void, the Portability Research Group at West Virginia University is exploring a variety of issues related to portability. This work has been funded both by a regular NSF grant and by the Computational Materials Science project of the NSF/WV EPSCoR program. The focus for the EPSCoR work has been primarily on portability of high-performance computing (HPC) software across parallel environments such as the CM-5. This presentation is based on that work.

Along with the benefits, there are costs associated with achieving a high degree of portability in a software unit. These costs take two principal forms:

- Increased development costs for the original software;
- Reduced performance and functionality compared to implementations tailored to a single environment.

Because of the tradeoff, the emphasis placed on achieving portability may vary for different software categories and application domains. If the value of multiple implementations is considered high, and the costs of portability are acceptable, then portability may receive priority. If on the other hand there are a limited number of likely target environments, and performance is critical, portability may not be judged desirable.

In the domain of high-performance computational software, there have been a number of traditional barriers to portability. Performance is indeed critical, and no strategy which substantially reduces performance can be tolerated. Many HPC programs were developed decades ago, before the advent of parallel processors. Since the 1970s the CRAY architecture has dominated high-performance computing; most programs not designed for conventional sequential machines have been designed to run on the CRAY.

The CRAY architecture provides high performance principally by means of high speed vector pipelining. Adaptation of older programs was still desirable to take advantage of this architecture. However, this adaptation was primarily carried out by use of specialized vectorizing compilers. Even though hand-tuning could have further improved performance, most scientists were not prepared to expend the effort to fully analyze and revise their programs.

This situation has changed considerably in the last 15 years. Many highly parallel architectures have come into use, challenging the CRAY as effective hosts for high-performance computing. These machines fall into a number of diverse categories, e.g., SIMD, MIMD shared memory, MIMD distributed memory. Moreover, each one incorporates many unique features to create a novel architectural concept. To achieve high performance, software must be tuned to the characteristics of each specific system.

Although there has been some convergence among these architectures, no ideal model has been found. New machines appear frequently, and the best environment for an application two years from now may be very different than the one in use today.
Users and developers of HPC software are now faced with a dilemma. On one hand, the proliferation of architectures makes portability essential if software is to have a long lifetime. On the other hand, the diversity of architectures makes the challenge of portability especially difficult.

The next section of this paper briefly overviews some general concepts for portability. Following this we discuss the problems and possible strategies for portability in the context of HPC software for parallel environments. The final section offers some particular guidelines for software which includes the CM-5 among its target environments.

2. Portability Concepts

This section briefly reviews some concepts related to portability, and introduces some preferred terminology. Most of the ideas in this section are discussed more fully in [Mooney 90].

Porting is the act of producing an executable version of a software unit in a new environment, based on an existing version. The term environment refers to the complete range of elements in an installation that interact with the ported software. This typically includes a processor and operating system; it may also involve I/O devices, libraries, networks, or a larger human or physical system. Many authors use the term platform with a similar (but possibly more restrictive) meaning.

A program is portable if and to the degree that the cost of porting is less than the cost of redevelopment. A software unit would be perfectly portable if it could be ported at zero cost; this is never possible in practice. Instead, a software unit may be characterized by its degree of portability, which is a function of the porting and development costs. The degree of portability is clearly dependent on the target; it is only meaningful to speak of the portability of a program with respect to a certain environment or class of environments.

The principal types of portability usually considered are binary portability (porting the executable form) and source portability (porting the source language representation). Binary portability is clearly desirable, but is possible only across strongly similar environments. Source portability assumes availability of source code, but provides opportunities to adapt a software unit to a wide range of environments. Most portability research assumes that the goal is source portability. This is virtually always the case for HPC software.

The porting process has two principal components which may be called transportation and adaptation. Transportation is physical movement; this may not be trivial since compatible media must be used and various types of representation conversion may be required. Adaptation is any modification that must be performed on the original version; we take this to mean both mechanical translation such as by language processors, and manual modification by humans.

Costs may be associated with the use of a portability-based strategy for software development. The usual costs have been outlined in the previous section. The corresponding benefits take the form of reduced costs to produce and maintain future implementations, as well as possible quality improvements in factors such as reliability.

There are several distinct portability problems that may be faced by software developers. The two principal ones are:

- How to port an existing software unit to a new environment
- How to design a software unit to be as portable as possible
The first problem includes the need to determine whether porting or complete reimplementaton is the more desirable strategy. Either approach may require significant effort for program understanding before proceeding.

The second problem requires consideration of the potential value of portability and the costs that can be tolerated. Suitable target environments must be characterized, to avoid expending efforts on unneeded portability. A wide range of strategies is then available to help achieve the desired portability level.

A software unit interacts with its environment through a set of interfaces. This situation is illustrated in Figure 1. The principal interfaces for a typical program are those to the processor, operating system, and support libraries. If the interfaces can be made to “look the same” over a set of environments, then the program can be easily ported. Note that it is not necessary that the underlying elements such as processor, operating system, etc., be identical; all that is required is that the interfaces appear the same. This can often be achieved through the use of appropriate standards.

There are many strategies for portable software design, including: develop a portable specification, use a standard language, use a portable programming discipline, isolate system dependencies, use standard libraries and interfaces, use “bridge software” for non-standard interfaces, etc. Most of these can be summed up by the following principles:

- **Use suitable standards**
- **Think portable**

![Figure 1 -- Common Program Interfaces](attachment:image.png)
3. Portability for HPC software

As noted above, portability is becoming increasingly desirable for many high-performance programs. Users are recognizing that if good software is to be usable for a long time, and continue to achieve state-of-the-art performance, it must be adapted to a range of different architectures.

The difficulties of achieving portability over a full range of parallel environments without sacrificing performance are severe. However, today’s users are often more willing to invest the effort to work toward this goal. Users of scientific HPC codes are accepting the need to revise these codes to take advantage of state-of-the-art processors, but the rule of thumb now states that most users are willing to revise their software once. Therefore, it is important to do a good job the first time.

In some application domains, the portability problem is raised mainly for new software, or for recently developed, well documented software. This is not the case for HPC codes; there is a huge collection of decades-old FORTRAN software which is still considered useful but needs to be adapted to modern environments. To exploit the modest architectural innovations of vector processors like the CRAY series, much of this adaptation could be performed automatically by vectorizing compilers. Often it was not necessary for those carrying out the adaptation to fully understand the structure of the original program.

To exploit highly parallel architectures, though, smart compilers are inadequate. Specific environments may provide the best performance only with particular data organizations and task partitioning that can impact data structures and algorithms throughout the program. For the adaptor who would do the best job, it is absolutely necessary to understand the program before adapting it. In recent years the fields of reverse engineering and program understanding have received great attention in software engineering research. A variety of analysis methods and tools have been created to extract and document the structure of “legacy” software. A full discussion of these methods is outside the scope of this paper (see [Waters & Chikofsky 94] for an introduction to the area), but they must be used to establish a clear understanding of the programs to be adapted.

The rest of this discussion assumes that we are considering either new program development or the revision of older programs that are well understood.

Principles for “portable parallel programming” (often called “architecture-independent programming,” or AIP) should be based on general portability strategies, but there are some significant differences. Key areas of consideration for HPC portability include: specification, architecture models, program models, standard interfaces, and algorithm adaptation. We will examine each of these briefly.

Specification

A precise written functional specification is an important starting point for any type of software development. This is equally true for HPC software. Even when adapting legacy software, a specification is essential, as it provides the evidence that program understanding has been successful.

It is generally advised that specifications concentrate on what rather than how. This is particularly important for portable HPC software. For example, the specifications should document formulas to be computed, and perhaps accuracy to be attained, but not specific methods to perform
the computation. Because of the diversity of architectures, the best implementation methods will inevitably differ.

For truly portable specifications, it is also desirable to identify the target class of architectures, and to avoid any unnecessary system-dependencies when specifying such elements as user interfaces, report formats, files, etc.

Architecture Models

To achieve portable software design, it is necessary to identify the intended class of target environments. For parallel environments the most significant variations will occur in the system architecture. A class of architectures can be characterized by an architecture model. It is then possible to design software to perform well on systems which conform to the chosen model.

Because of the great diversity of parallel architectures, it is difficult to capture all (or even most) of them in a single model. Moreover, such a model would be of limited use to us. A very generic model will represent few of the details of any specific system. If a program is designed based only on such a model, it will require considerable adaptation to suit any actual target.

Nonetheless there are broad-based models that can aid in the initial conceptual stages of HPC software design. One well known example is the PRAM (Parallel Random Access Machine) model [Fortune & Wylie 78] which can represent most architectures with a (logical or physical) shared memory. Another useful model is that proposed by Hillis [Hillis 85] on which the Connection Machine series (CM-1, 2, and 5) is based. In this model, all systems are logically viewed as a collection of an infinite number of processors, with a fully programmable interconnection network.

A fuller representation of the details of specific targets requires multiple models. These models further characterize the capabilities of individual processing nodes and the nature of the storage system and interconnection network. Examples may include SIMD, MIMD shared memory (with uniform or non-uniform memory access), MIMD distributed memory, etc. We rely here on the intuitive meaning of these models, although in practice formal definitions would be used. Space precludes describing these models, but they are discussed in many good texts on parallel architectures (e.g. [Almasi & Gottlieb 94, Hwang 93]).

A program design conceived around a generic model may be refined for a more specific one, while avoiding the temptation to design with a single precise architecture in mind. If the desired target class spans multiple models, then multiple designs may be required.

Programming Models

Most HPC software may be considered to conform to a particular parallel programming model (or paradigm). Such a model can characterize, among other things, the communication mechanisms to be used and the types and levels of parallelism to be supported. Examples of programming models include data parallel, shared memory, message passing, and variants of message passing such as host-node, hostless, and “single program” (SPMD).

The programming model is distinct from the architecture model, but a given architecture model can support some programming models more effectively than others. Therefore the choice of a target architecture class should help to determine the model to be used.
There are also more formal and less widely known models which have attractive properties for particular types of software. These are often captured in specific notations or languages. Two examples are the CHAMELEON language [Alverson & Notkin 93] which is well suited to a particular class of algorithms on shared-memory systems, and the Bird-Meertens formalism [Skillicorn 90] which is considered to be suitable for most of the principal architecture models.

A significant problem which has yet to be addressed is how to convert existing programs to a new and more desirable model. Limited tools exist to transform shared memory programs to message passing, for example, but in general this transformation requires substantial manual effort.

Standard Interfaces

The heart of any portability strategy is the use of appropriate standards for most if not all of the interfaces which the software unit will use to interact with its environment. Standards may appear at any level of representation ranging from the original form of the program (e.g. language standards) to the executable form (e.g. processor standards). The hope is that implementations of the chosen standards will already exist in the target environments; if not, implementations or bridge software may be developed during the porting process.

Suitable standards for HPC software include some that are commonly used for more general software, but also some specialized toward HPC requirements. It is desirable that HPC interfaces be fairly high-level, to allow greater flexibility of implementations at the other side of the interface.

The most common languages for HPC programming are FORTRAN and (more recently) C. Each of these have well-established ANSI standards. However, there is a need to extend these languages for parallel environments by allowing the specification of explicit parallelism and preferred data distribution. Ideally this should be done in a way which expresses the requirements of the program without assuming a specific system architecture.

The most recent version of FORTRAN, commonly called FORTRAN 90, has tried to address some of these needs but has not become widely accepted. Instead, a separate user-driven effort has produced specifications for “High-Performance Fortran” (HPF) [HPFF 94, Kennedy 95]. This specification extends standard FORTRAN with directives in the form of special comments that can be interpreted by a preprocessor. It is especially suitable for a data-parallel style of programming which is implementable on a wide variety of architectures. HPF seems likely to become widely used; an effort to develop “High-Performance C” is in an earlier stage.

Standard libraries play an important role in portable parallel programming. Important examples include the numerical libraries produced by the Numerical Algorithms Group (NAG), and the widely used linear algebra package LAPACK.

Message passing is a critical function which arises only in explicitly parallel programs. Until recently no common agreement has existed on message passing models. Message passing programming models have been difficult to learn, and portability has been almost unattainable. This situation has led in part to the popularity of data parallel programming and simulated shared memory, even though message passing is potentially more powerful and more efficient in many cases.

This situation has improved with the introduction of two important user-driven standards, PVM and MPI. PVM (Parallel Virtual Machine) [Geist et al 95] provides a common
communication model designed especially for distributed networks of possibly dissimilar machines. MPI (Message Passing Interfaces) [Gropp 94] defines a common model for basic message passing operations. Implementations for both of these standards are available without cost for a wide variety of systems.

Input and Output is often a major bottleneck for parallel programs. Unfortunately, work has only begun on developing higher-performance I/O models, and no standards have yet emerged. Software with high-performance I/O requirements should isolate I/O components in distinct modules as far as possible, since these will probably have to be revised for each implementation.

Algorithm Adaptation

Because of the extreme diversity of architectures designed for high-performance computing, it may sometimes be necessary to use completely different algorithms when moving from one architecture to another. Differences in the basic architecture model or in parameters such as processor count, communication bandwidth, or floating point precision, may cause different methods to emerge as the best performers.

Since the need for distinct methods is most acute in lower-level algorithms, much of the problem can be addressed by use of relatively high-level portable libraries and flexible standards. For example, standards like LAPACK and MPI are designed to allow a wide latitude of underlying implementations, while the special comments of HPF provide hints and guidance which can be interpreted in various ways by specific implementations.

It is further desirable for portability that HPC programs be organized in such a way that critical algorithms not available from standard libraries be isolated in distinct modules so they may be more easily substituted. Where possible, alternate methods may be programmed in a single implementation, to be chosen at compile time or run time according to specific configuration parameters.

4. Portability and the CM-5

The final section of this paper applies the discussion of the preceding sections to the specific case of the CM-5. The problems to be addressed are moving software from other environments to the CM-5, and making CM-5 software as portable as possible so it can be easily adapted to other environments in the future.

The difficulty of adapting existing software to the CM-5 depends on the intrinsic portability of the software to be adapted, and on the level of understanding achievable by the adaptor. Understanding the existing program is the first step. There is no substitute for the program understanding process outlined above for legacy systems, guided by whatever analysis tools may be available. This process should result in both a written functional specification and good structural documentation of the existing program.

A good understanding of an old program leads invariably to many ideas on improving its structure, for reasons such as better understanding, improved performance, or enhanced portability. The developer who truly understands the program should not hesitate to follow these ideas.
Whether the task is to implement a new (portable) CM-5 program, or port an old one, a functional specification should be developed. Up to this point the fact that the CM-5 is the current target should have little relevance. Its only influence should be to ensure that distributed memory MIMD (or more generally, the Hillis model) is included among the identified target architecture models. The functional requirements of the program should be described without reference to a CM-5 implementation.

At the design stage some consideration may be given to a known target architecture. The CM-5 is particularly suited to a data parallel programming model, or to SPMD message passing. If these approaches suit the program, all is well. If not, a choice must be made between following a model which may be implemented less efficiently, or changing the model entirely.

A critical issue is the choice of standards, beginning with language standards. The CM-5 supports extended versions of FORTRAN and C called CM FORTRAN and C*. These languages may be the only available choice for now, but their extended features should be used cautiously. HPF is becoming available for the CM-5 and would probably be the language of choice for FORTRAN programs where possible. Meanwhile, the portable developer should be aware of HPF and develop CM FORTRAN programs in a way which can be converted to HPF with relatively little effort. In fact, HPF directives can be incorporated in CM FORTRAN programs at any time, since they will be interpreted as comments.

Notice that both CM FORTRAN and HPF feature extensions suitable primarily for data-parallel programming. If a different model is to be used, then standard FORTRAN (FORTRAN 77) should be adhered to as far as possible.

High-Performance C will not be available for some time. C programmers should use the extensions of C* cautiously, with awareness of the need to adapt these forms when porting.

Libraries like the NAG library and LAPACK are generally available for the CM-5 (but the NAG libraries are commercial). Implementations of PVM are also available, and an MPI implementation is being completed. When functional, this should be considered as a more portable alternative to CMMD.

Bibliography

[Almasi & Gottlieb 94]

[Alverson & Notkin 93]

[Fortune & Wylie 78]

[Geist et al 95]
URL: [http://www.epm.ornl.gov/pvm/]
[Gropp 94]

[Hillis 85]

[HPFF 94]

[Hwang 93]

[Kennedy 95]

[Lecarme et al 89]

[Mooney 90]

[MPI 95]

[Skillicorn 90]

[Waters & Chikofsky 94]