Abstract—We outline our recent efforts in developing MIT Lincoln Laboratory’s Mapping and Optimization Runtime Environment (LLMORE). The LLMORE framework consists of several components that together estimate and optimize performance critical sections of an application. This framework can be used to improve the performance of parallel applications and as an important tool for analyzing different hardware architectures. In this paper, we describe the use cases that have driven the development of LLMORE. We also give two concrete examples of how LLMORE can be used to improve the parallel performance of a numerical operation and characterize the power efficiency of numerical algorithms and computer architectures.

I. INTRODUCTION

This paper is intended to serve as an introduction to MIT Lincoln Laboratory’s Mapping and Optimization Runtime Environment (LLMORE) and overview of the LLMORE software stack. LLMORE consists of several components, which together quantify and optimize the parallel performance of some performance critical section (e.g., a set of kernels) in a parallel application. LLMORE can be used to improve the performance of parallel applications and as an important tool for analyzing new hardware architectures. LLMORE is currently being developed as part of the Photonically Optimized Embedded Microprocessors (POEM) project. As part of the POEM project, LLMORE will be used to study chip-scale photonics and its impact on applications.

In this document, we first give an overview of the LLMORE project and describe the goals and general design of the current software effort. Next, we briefly describe a set of five use cases that outlines the ways in which we envision LLMORE being used. We more fully expand upon the runtime mapping and performance evaluation use cases, which are currently well supported by the LLMORE software. Next, we demonstrate through two examples how LLMORE supports these two use cases. Finally, we briefly summarize and discuss our future plans for developing LLMORE.

A. LLMORE Overview

LLMORE is a framework for optimizing the mapping of parallel data objects in parallel applications, simulating and optimizing new (and existing) architectures, generating performance data (runtime, power, etc.), and generating/executing optimized code on target architectures. It is important to note that LLMORE will not generate optimized parallel code for any set of instructions. It attempts to optimize parallel computation in the context where the computation uses parallel objects whose data distributions are described by maps, structures that contain information needed to specify how the data in the objects is mapped onto the processors of the parallel system. Maps (and the functions that use them to move data) encapsulate much of the complexity of the data movement necessary in parallel computations. By writing algorithms in a “map oblivious” (oblivious to the specific map) manner, users can focus more on algorithmic development and less on parallel programming issues. Constraining the LLMORE optimization problem to map-based objects, makes the problem of automatic parallelism tractable.

Figure 1 depicts a high level view of LLMORE, what it receives as input and what it produces as output. As input, LLMORE takes user code, a model of the system architecture, and a set of LLMORE specific parameters. LLMORE’s output consists of one or more of the following five items: a complete set of optimized maps (describing the data distribution for all parallel objects in the user code), performance data, a set of optimized architectures (from a larger parametrized set of architectures) for the user code, optimized generated code, or results from a run on target architectures.

Fig. 1. Overview of LLMORE.

User code consists of a sequence of numerical kernels and simple program flow directives (loops, branches, etc.) that
LLMORE can parse. This user code may or may not be an entire program or application. A user typically would want to input code that encompasses the most performance critical sections of their application. LLMORE is designed to support user code in multiple languages and libraries. LLMORE can also accept as input a set of maps (or rules describing map behavior) that enforces how some or all of the data objects in the user code are to be distributed. This allows some flexibility when applications have a priori knowledge regarding what makes a good distribution for particular data objects. LLMORE can also accept as input data pattern information about the objects used in the user code. For example, a user may want to provide the nonzero information for a sparse matrix to help LLMORE better optimize the map for that matrix. The architectural model consists of a machine model, which describes the target hardware for the application, and a programming model, which stipulates how the parallelism in the computation will be expressed on this target hardware. This machine model does not necessarily correspond to one specific architecture but could be parametrized with the model corresponding to a set of architectures that are specified by the parameter values. The final input is a set of LLMORE parameters, which controls various aspects of an LLMORE run.

B. LLMORE Software Design

In order to support the various uses of LLMORE with different potential inputs and outputs, we are designing LLMORE to be a flexible, modular, and component based infrastructure. Figure 2 gives an overview of the LLMORE software design. The arrows entering and exiting the LLMORE box represent the input and output to LLMORE (also shown in Figure 1). Everything else in the figure is part of the LLMORE software infrastructure. The LLMORE software is divided into three different layers: the language interface, the core functionality, and the machine interface.

Fig. 2. Overview of LLMORE Design. AST=abstract syntax tree.

The LLMORE language interface (shown in red in Figure 2) is the set of components through which the user interacts with LLMORE. The language interface provides language/library specific instructions for users to set up, run, and obtain output for an LLMORE optimization problem. The language interface primarily consists of two components: the Parse Manager and the Map Manager. The Parse Manager provides several interfaces to parse user code for different languages and libraries. The Parse Manager parses user source code in its native language and encodes the source code into an abstract syntax tree (AST) by calling the appropriate functions in the AST Builder, which is part of the core functionality layer. The Map Manager translates user defined maps to an internal LLMORE map representation in conjunction with the Map Builder (core functionality layer). Currently, LLMORE supports only the C++ language through the LLMORE language interface. However, we plan to support Matlab and Python as well in the future.

The LLMORE core functionality (shown in yellow in Figure 2) is a set of components for setting up and solving the optimization problem (optimizing maps, generated code, or architecture parameters) and providing performance information. One key feature is that the behavior of LLMORE core functionality is independent of the user code language. The LLMORE core functionality accomplishes this by using an AST representation internally. The common input interface to the LLMORE core is through the AST Builder component, which provides hooks for converting the language/library specific user code into the language neutral AST. The AST serves as a common representation of the user code operations that LLMORE can utilize in its map optimization. The Map Builder component serves a similar purpose in providing hooks for converting the language/library specific maps into the internal map representation. Another key feature is that the LLMORE core functionality is architecture independent and can solve the optimization problem without executing on the actual target architecture. By themselves, the LLMORE core components are sufficient to evaluate the optimization problem’s objective and find a solution. Optionally, these components can interface with LLMORE machine interface layer or third party libraries to improve this optimization. In addition to the AST Builder and Map Builder components, the LLMORE core functionality layer contains the Analyzer and Optimizer component. The Analyzer and Optimizer is the crux of the LLMORE framework, being responsible for setting up and solving the map and architecture optimization problems.

The final layer is the machine interface layer (shown in green in Figure 2). This layer is architecture dependent and provides an interface to various architectures. It contains two primary components: the Code Generator and the Runtime Engine. The Code Generator component generates optimized code for different architectures. The Runtime Engine is the component that controls the execution, launch, or emulation of generated code on particular architectures. Currently, the machine layer interface for LLMORE has yet to be developed and is left as future work.

II. USE CASES

The development of the LLMORE framework has been driven primarily by five major use cases: static mapping, run-
time mapping, performance evaluation, architecture optimization, and execution of target architectures. Here, we briefly summarize these use cases. In the subsequent subsections, we will describe the runtime mapping and performance evaluation use cases in more depth, since they are particularly well supported by the current LLMore software stack. In static mapping, the user inputs the code (or representation of the code) to be analyzed and an architectural model describing the target system. LLMore uses these inputs to calculate a complete set of optimized maps for the parallel objects represented in the user code that will be used when the user’s application is run on the actual parallel computation. The resulting LLMore maps are stored (e.g., in a file), so that they may be subsequently used for distributing the parallel objects when the application is run in parallel. In runtime mapping, a parallel application with previously distributed data calls LLMore to calculate a new distribution that can be used to improve the parallel performance of the application. The performance evaluation use cases focuses on evaluating some performance metric (e.g., runtime or power) for the specified user code and architecture model. The architectural optimization use case attempts to find an optimal hardware architecture (within a range of hardware architectures) for the specified user code. This is useful for gaining insight into what architectures work well for particular important computational kernels. The use case of execution on target architectures focuses on generating optimized code for and/or executing on specific target architectures. These use cases should be viewed as composable in that a user’s objective may utilize more than one of these.

A. Runtime Mapping

As described above, the runtime mapping use case focuses on optimizing the data mapping for currently mapped data in order to improve the parallel performance of the specified operations. Applications that internally generate poorly distributed data objects might find this usage of LLMore particularly useful for improving the distribution of these data objects. Applications that input large sparse matrices also could benefit greatly from the runtime mapping capabilities of LLMore. These applications initially distribute the sparse matrices in a naive fashion as these matrices are loaded into memory (since the data is too large to duplicate across processors). Once loaded, LLMore can determine more optimized maps for the matrices, which can improve the parallel performance of the application.

The LLMore overview (Figure 1) gives an accurate overview of the runtime mapping use case. For this use case, LLMore accepts as input user code, a set of LLMore parameters, and an architectural model. As part of the user code, maps may be also inputted for the parallel objects, either implicitly through the incoming distribution of the data or explicitly if the user wishes to tie a specific map to the variable. As optional input, the user code may be augmented by providing the data pattern information about particular objects. As previously mentioned, the architecture model contains a machine model, which describes the target architecture on which the code will run. Typically for the runtime mapping case this would be the architecture that LLMore is running on as well. The architecture model also contains a programming model, which describes how the parallel programming will be realized. The output for the runtime mapping use case is a complete set of optimized maps (item 1 in Figure 1).

Figure 3 shows the data flow through the LLMore components for the runtime mapping use case. The first step is for LLMore Parse Manager (which is designed to support multiple programming languages and libraries) to parse the aforementioned user code and convert this code into an AST that effectively represents the operations. Next, the AST is passed to Analyzer and Optimizer component that will optimize the data distribution of the parallel objects in the user code. In particular, the AST is passed to the Mapper component, which create maps for each parallel object represented in the AST. The Mapper attaches these maps to the AST to create a “mapped AST.” This mapped AST is passed to the Performance Evaluator component, which calculates performance data and gives some indication of how well the user code would perform with the current set of maps on a particular architecture.

The Performance Evaluator can use either the Simulator component or the Runtime Engine component to obtain the performance data needed by the Mapper. In the former case, the Simulator uses the architectural model and the mapped AST to estimate the runtime for executing the user code on the target architecture. An optional component in the “Simulator path” is the Machine Independent (MI) Code Generator. The MI Code Generator takes as input the architectural model and mapped AST and generates machine independent code that is not specific to a particular architecture. Our current machine independent code representation is a dependency graph, which specifies the dependencies of different computation, memory, and network operations that occur in the parallel computation. The LLMore Simulator component is written to interface with multiple simulators, including internally developed and third party simulators. Currently, the LLMore Simulator interfaces with one simulator, an internal simulator that processes the architecture model and dependency graph to return a
count of the floating point operations and an estimate runtime
and power. The “Runtime Engine path” works in a similar
fashion to the “Simulator path”. The Runtime Engine compo-
nent also takes as input the architecture and the AST model.
However, instead of simulating the user code in software, the
Runtime Engine will execute the user code with the LLMORE
calculated maps on the actual target architecture. The Code
Generator is an optional machine dependent component that
can produce optimized user code for particular architectures
that can be used by the Runtime Engine. Currently, the
“Runtime Engine path” has not been implemented and is left
for future work.

Once the Performance Evaluator returns the performance
data, this data is then fed back to the Mapper to potentially
improve the maps. Once the analysis and optimization process
is complete, the final complete set of maps are returned to
the application.

For the runtime mapping use case, LLMORE is integrated as
part of the parallel application, being invoked through a library
call. Thus, LLMORE needs to optimize the maps quickly,
so that it does not become a bottleneck for the application.
This means that for the runtime mapping case, LLMORE will
most likely execute in parallel. It is important to note that
it is crucial that the cost of LLMORE optimization can be
amortized by the repeated use of the optimized portion of the
user code. For instance, the cost of optimizing a sparse matrix-
vector multiplication algorithm might be amortized through
repeatedly using that operation in an iterative method. Related
to this, it may be necessary to skip the Performance Evaluator
since this can be particularly time consuming. LLMORE
also needs to provide the information on its optimized data
mappings in a manner that is useful to the parallel applications.
One option is to simply return map objects that the parallel
application can use to remap its data. Alternatively, LLMORE
could provide a mechanism for redistributing the application
data. Initially, we have decided to implement the former option
although the latter option will be considered as future work.

B. Performance Evaluation

As described above, the performance evaluation use case
evaluates the specified code on the specified architecture to
calculate and return one or more performance metrics. As we
saw in the runtime mapping case, performance evaluation is
an important part of the Analyzer and Optimizer component
(as seen in Figure 3). However, performance evaluation can
also be the end goal of the user. The general outline of this
use case is basically the same as for the runtime mapping use
case as described in the previous subsection with the exception
of the output. The output for the performance evaluation use
case is performance data.

Figure 4 shows the data flow through the LLMORE com-
ponents for the performance evaluation use case. Many of the
steps are identical to the runtime mapping use case (shown in
Figure 3), so we only describe the important differences. One
difference is that the map conversion of input maps is now
optional since the parallel objects in the user code are not
necessarily initially mapped. Another difference is that the
Mapper component is also optional. LLMORE can perform
the performance evaluation without remapping the user data.
Finally, the output of this use case is the output resulting from
the Performance Evaluator component.

III. Runtime Mapping: Sparse Matrix-Vector
Product Example

In this section, we present an example that illustrates how
LLMORE can be used in the runtime mapping use case.
Figure 5 shows the C++ code for a simple sparse matrix-vector
product example. The example utilizes the Epetra parallel
numerical linear algebra library (a Trilinos package [1]) for
which LLMORE supports the parallel sparse matrix (Epetra_CrsMatrix) and vector (Epetra_Vector) datatypes. In this
example, LLMORE is used to improve the maps that specify
that distribution of the sparse matrix and the two vectors
involved in this operation.

The code fragment shown in Figure 5 assumes that the
Epetra sparse matrix and vectors have already been allocated.
The LLMORE specific code is shown in lines 5–17. Lines 5–7
are responsible for setting up the parameters that will guide the
LLMORE execution. First, the parameter “PARTITIONING
METHOD” is set to “DOMAIN,” which indicates that consecu-
tive elements in vectors should be assigned to the same process
and the nonzeros in consecutive rows should be assigned to the
same process. The second parameter “VERBOSITY LEVEL”
indicates how much output should be displayed during the
LLMORE execution. In Line 9, the LLMORE Framework is
initialized with the previously described parameters.

Lines 11–15 shows the current approach for representing the
user code in LLMORE. Here, LLMORE specific commands
from the C++ version of the Parse Manager are used to encode
the operations from the user code (specifically the sparse
matrix-vector product that will be executed on line 26) into
the AST. In Line 11, the sparse matrix is declared in the
LLMORE language and the resulting variable is linked with the
corresponding Epetra sparse matrix by means of the pointer
that is passed into the constructor. This connection between
the LLMORE sparse matrix variable llmoreA and the original
Epetra matrix epetraA provides information to LLMORE
(e.g., number of rows, sparsity pattern) that is necessary to

Fig. 4. Data flow for Performance Evaluation Use Case. Dashed boxes
indicate optional components that may or may not be part of an LLMORE
run.
Fig. 5. LLMORE code example. Optimization of Epetra sparse matrix-vector product using LLMORE.

In this section, we describe a simple power mini-application that illustrates how LLMORE may be used in the performance evaluation use case. In this mini-application, LLMORE simulates user code on a simple mesh network, returning the power of the system, the simulated time of execution, and the number of floating point operations for the simulation.

Figure 6 outlines the code necessary to implement this power mini-application in the LLMORE framework. The first 10 lines are very similar to those shown for the sparse matrix-vector product example (Figure 5). Lines 1-3 set up the parameters needed to guide the LLMORE execution. Line 5 initializes the LLMORE framework with these parameters. Lines 7-10 encode the user code (which corresponds to a dense matrix-matrix product) into the LLMORE AST. It is important to note that unlike the sparse matrices, the LLMORE dense matrices need not be tied to native matrix objects since there is no sparsity information that is needed by LLMORE. Instead, the user simply provides the number of rows and columns to the constructor (N in this case).

For brevity, the performance evaluation mini-application. Illustrates how LLMORE can be used to obtain performance data (runtime, number of operations, power) for dense matrix-matrix product on a simple mesh network.
Line 12 calls a function `buildMeshNetworkArch` to build the mesh network architecture model. The architecture model would be typically be defined by the user through additional LLMORE code (or read from a file) but the details have been omitted here for brevity.

After the architecture model has been constructed, the maps are computed for the parallel objects contained in the AST (line 14). In line 16, a dependency graph is built from the architecture model and mapped AST (details also omitted for brevity). This dependency graph specifies what dependencies that each computation or communication operation has on previously completed operations. In line 18, the previously described internal simulator is initialized to process this dependency graph and architecture model. The simulator is run (line 19) and returns the runtime of the simulated operations, the number of floating point operations, and the static power of the system specified by the architecture model.

From the performance data that is returned, we can calculate the energy efficiency performance metric GFLOPS/W. We can use LLMORE to vary the architecture parameters (such as the static power of the processor and memory) in an attempt to satisfy some GFLOPS/W objective. We can also simulate different user code by replacing the dense matrix-matrix product LLMORE instructions (lines 7-10) with different operations. A key feature of LLMORE is the ability to easily simulate many different kernels and applications once the specified architecture has been encoded.

Figures 7 and 8 show the energy efficiency performance metric GFLOPS/W for the dense and sparse matrix-matrix product kernels, respectively, for a simple 16-by-16 mesh network architecture. These figures show the GFLOPS/W as the static power allocated to the processors and memory is varied. The GFLOPS/W values for the sparse kernel is significantly lower than the dense operation due to the poorer performance of this numerical kernel on this architecture. In this manner, LLMORE can be a useful tool for power optimization.

V. SUMMARY AND FUTURE WORK

In this paper, we introduced the MIT Lincoln Laboratory’s Mapping and Optimization Runtime Environment (LLMORE). We gave an overview of the LLMORE framework, which consists of several components that together estimate and optimize the parallel performance of some performance critical section in a parallel application. We outlined the five use cases that have driven our development of LLMORE and expanded further upon the runtime mapping and performance evaluation use cases. Finally, we presented two concrete examples that illustrated how to use LLMORE to accomplish different objectives: improving the parallel performance of a numerical operation and characterizing the power efficiency of numerical algorithms and computer architectures.

Much development work is still needed for LLMORE to reach its full potential. Currently, we support the C++ language with the Epetra parallel numerical linear algebra package, but we plan to support additional languages and libraries in the future, including Matlab, Python, and PVTOL. Although much of the language interface (at least for C++) and core functionality is in place, we have not begun work on the Machine Interface Layer. So in future work, we will develop the Code Generator and Runtime Engine components to support execution on target architectures.

REFERENCES