Addressing the Multicore Trend with Automatic Parallelization

Nadya Bliss

The slowdown of Moore's Law and the need to process increasingly large data sets are driving the computer hardware community to develop multicore chips in addition to the already prevalent commodity cluster systems and multiprocessor embedded systems. As parallel processors become ubiquitous, reducing the complexity of parallel programming becomes increasingly important. Lincoln Laboratory has developed an automatic parallelization framework, called pMapper, which is general with regard to programming languages and computer architectures and which focuses on distributing operations common in signal processing.

Parallel computing is becoming increasingly prevalent in both commercial and military applications. The current computer industry trend is toward creating processor chips that contain multiple computation cores—from the recent desktop Intel processors containing four processing cores [1] to the IBM/Sony/Toshiba Cell processor, which has eight high-performance computation cores and one general-purpose processor [2]. These multicore architectures are becoming dominant in all areas of computing, from personal computers to real-time signal processing. In the past, programming to exploit the architectural properties of a parallel machine was limited to a narrow field of experts. Because of the hardware industry's shift towards parallel computing, however, a wider field of programmers and algorithm developers must take advantage of these computer architectures.

Two key factors have contributed to this paradigm shift: the slowdown of Moore's Law and the need to process increasingly large data sets. Over the past 40 years, the number of transistors on a microchip doubled every 18 to 24 months [3]. Increasing the number of transistors in a constant area generally yields proportional increase in a single processor's clock speed and, therefore, its computation speed. But as the industry approaches the physical limits of transistor dimensions, the demand for continued improvements in computation speed is driving hardware manufacturers to produce multicore processors, with multiple processing units on a single chip.

The need for parallel processing is evident in both scientific and real-time computing. Scientific computing in high-level languages such as MATLAB requires high-fidelity simulations and computations that can be achieved by increasing the number of parameters and the size of the datasets. Similarly, in the field of embedded real-time computing the replacement of analog receiver technology (in sensor arrays) by digital technology requires faster digital processing capabilities at the sensor front end. Additionally, the migration of more and more post-processing, such as tracking and target recognition, to the sensor front end necessitates increasingly powerful processing architectures. Indeed, the need to process large quantities of data at high speeds is making parallel programming important at all levels of programming expertise. This need to parallelize programs often reduces productivity and prolongs algorithm development.

While the architecture community has been addressing the need for more computing power, techniques for parallel programming have been advancing relatively

Amdahl's Law

A common misconception in parallel programming is that the more processors are used, the faster the program will run. In reality, there is a maximum speedup that a program can achieve, as stated by Amdahl's law [a]. In its most general form, Amdahl's law provides the means to compute the maximum expected improvement in running time, given that only part of the program can be improved. Consider a program that can be broken up into k parts and each part can be sped up by a factor of S_k. The maximum achievable speedup for the entire program, S_{max}, is

$$S_{max} = \frac{1}{\sum_{k=0}^{n} \frac{P_k}{S_k}},$$

where

- P_k = percentage of the program
- S_k = speedup (1 equals no speedup)
- k = label for each corresponding percentage and speedup
- n = number of parts the program is broken up into.

To compute the maximum speedup of parallel programs, Amdahl's law reduces to

$$S_{max} = \frac{1}{(1-P) + \frac{P}{S}},$$

where P is the percentage of the program that can be parallelized and S is the speedup achievable for the parallelizable part of the program. In the optimal case, S equals the number of processors; usually, however, S will be lower than that. Figure A shows the relationship between speedup and processor count. If 75% of the program is parallelizable (blue curve) and S = number of processors, then the maximum speedup achievable, as S approaches infinity, is

$$\frac{1}{1-0.75} = 4.$$

Thus no matter how many processors are used, this program can never run more than four times faster.

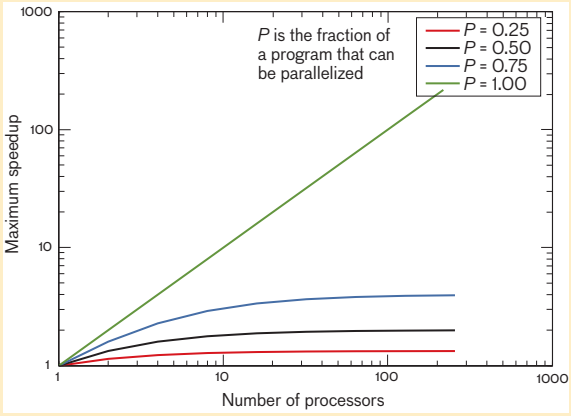

FIGURE A. Maximum speedups achievable for programs with various fractional parallel parts.

Reference

a. G.M. Amdahl, "Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities," in *AFIPS Conf. Proc.*, 1967, pp. 483–485.

slowly. Programming paradigms that emerged in the 1980s and 1990s, such as message passing, are still popular. Although significant advances have been made in raising the level of abstraction in parallel programming [4–7], parallel compilers [8], and program optimization tools [9], parallel programming has remained the province of a small number of specialists. Determining how to program a parallel processor efficiently is a difficult task that requires the programmer to understand many details about the computer architecture and parallel algorithms. Computer scientists have been developing various techniques for both detecting and utilizing parallelism, and have made significant progress in the area of instruction-level parallelism—that is, the ability to execute multiple low-level instructions at the same time. General program parallelism, however, is still an active area of research with many unanswered questions.

Why is parallel programming so difficult? There are a number of reasons. First, decomposing, or mapping, data structures and tasks in a serial program into parallel parts is a challenge. Second, writing parallel code that runs correctly requires synchronization and coordination among processors. Finally, the most difficult task is writing efficient parallel programs, a task that requires careful balancing of a program's communication and computation parts. When dealing with a single-processor architecture, the programmers try to minimize the number of operations the processor must perform. Counting the number of operations in a serial application is generally straightforward, and well-understood performance bounds exist for many algorithms.

When the program is moved to a parallel machine, minimizing the number of computations does not guarantee optimal performance. The same is true of serial algorithms on today's complex processors because of multi-level memory hierarchies. The problem is compounded when moving to a parallel system. First, the theoretical speedup of the entire program is limited by the longest serial path, as quantified by Amdahl's law (see the sidebar "Amdahl's Law") [10]. Second, the speedup of the parallelizable part of the program is highly dependent on interprocessor communication. Efficiently mapping the application, or distributing parts of the application between multiple processing elements, becomes increasingly important. The performance bounds of parallel algorithms depend strongly on how the data and computation are distributed or mapped; this dependence makes it more difficult to estimate and optimize performance. The situation is further complicated because the algorithmic details might change as the algorithm is moved from a single processor to a multiprocessor machine. Often the most efficient serial algorithm is not the most efficient parallel algorithm.

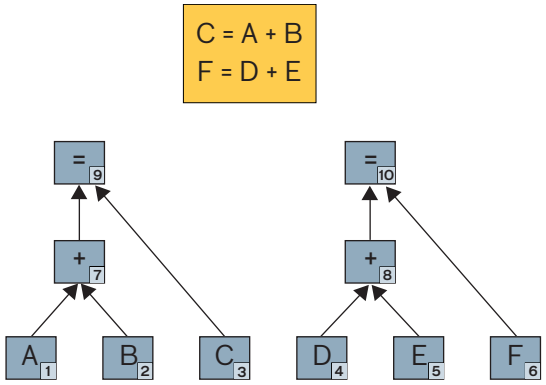

FIGURE 1. A simple program (top) is represented by a parse tree, or signal flow graph. Note that there are no dependencies between computation of C and F: nodes 1, 2, 3, 7, and 9 are not connected to nodes 4, 5, 6, 8, and 10. Thus the compiler can execute the two addition operations in parallel or out of order.

Signal and image processing have always been key mission areas at Lincoln Laboratory. The algorithms developed for these applications are computationally intensive, thus pushing the Laboratory to the forefront of developing parallel algorithms and libraries. Achieving high-performance, functionally correct parallel programming is particularly important and has led to the development of an automatic mapping framework, pMapper.

Automatic Parallelization versus Instruction Level Parallelism

Parallel computers, as well as research on parallel languages, compilers, and distribution techniques, first emerged in the late 1960s. One of the most successful research areas has been instruction-level parallelism [11], which refers to identifying instructions in the program that can be executed in parallel or out of order and scheduling them to reduce the computation time.

Figure 1 illustrates a simple program and an associated signal flow graph, or parse tree. Note that there are no dependencies between the computation of C and F. As a matter of fact, the two sub-trees are completely disjoint:

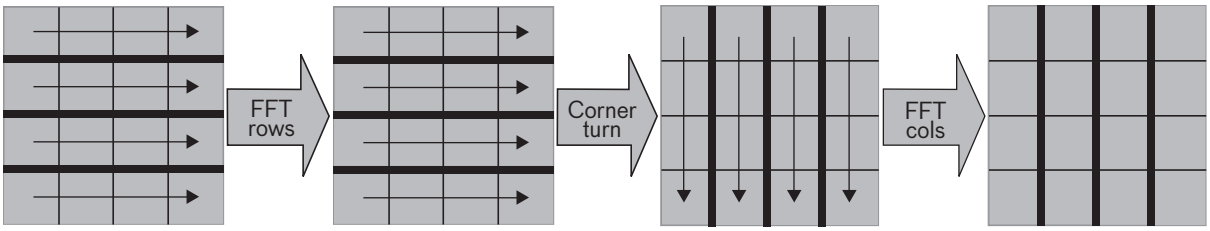

FIGURE 2. A common algorithm for implementing a parallel fast Fourier transform (FFT) is via two one-dimensional FFTs. The first FFT is performed along the rows; then, after the data are re-organized with a corner turn, the second FFT is performed along the columns.

nodes 1, 2, 3, 7, and 9 are not connected to nodes 4, 5, 6, 8, and 10. This lack of dependencies indicates to the compiler that the two addition operations can be executed in parallel or out of order (for example, if changing the order speeds up a subsequent computation). If the architecture allows for multiple instructions to be executed at once, this approach can greatly speed up program execution. Such instruction-level optimization has been incorporated into a number of mainstream compilers.

At a higher level, specifically at the kernel and function levels, this technique is referred to as concurrency analysis: analyzing parse trees of programs and determining which functions and operations can be performed in parallel. This area has also been researched extensively and incorporated into parallel languages and compilers [12].

Unfortunately, instruction-level parallelism and concurrency-analysis techniques do not solve the automatic parallelization problem, or the problem of taking a serial code and determining the best way to break it up among multiple processors. It is not sufficient to build a signal flow graph or parse tree of a program, determine what nodes can be executed in parallel, and then break up the program accordingly.

To explain why that is the case, let us consider another simple example. A common implementation of a parallel fast Fourier transform (FFT) on a large vector is performed via two one-dimensional FFTs with a multiplication by twiddle factors. For the purpose of this discussion, let us ignore the multiplication step and simply consider the two FFTs. First, the vector is reshaped into a matrix. Then one FFT is performed along rows and the second along columns, as illustrated by Figure 2. The FFT of each row or column of length N is computed according to

$$X(k) = \sum_{j=1}^{N} x(j)\,\omega_N^{(j-1)(k-1)},$$

where x is the input vector (row or column), X is the output vector, N is the vector length, and

$$\omega_N = e^{(-2\pi i)/N}.$$

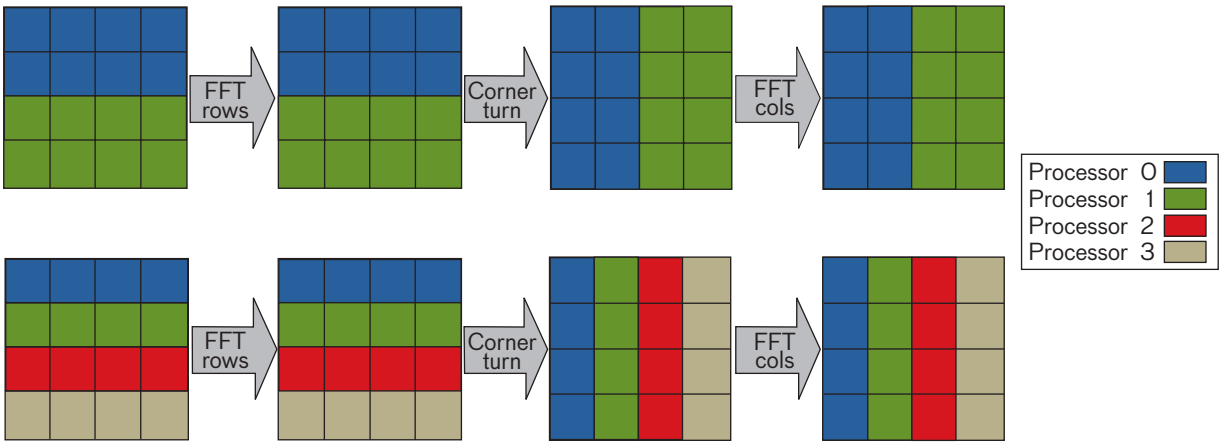

FIGURE 3. The top row illustrates mapping the FFT computation onto two processors. The bottom row is the same computation distributed onto four processors. Different colors indicate different processors.

First, consider the details of the serial implementation. The time complexity of the operation is simply the computational complexity of the two FFTs, which is $5N\log_2(N)$* multiplied by the number of rows (columns), where N is the number of elements in each row (column). The computational complexity of an operation is a count of arithmetic operations that take place during the computation, while the total time complexity could also include the time needed for communication in a parallel implementation. The complexity of the computation or communication operation is dependent on N, the number of elements involved in the operation.

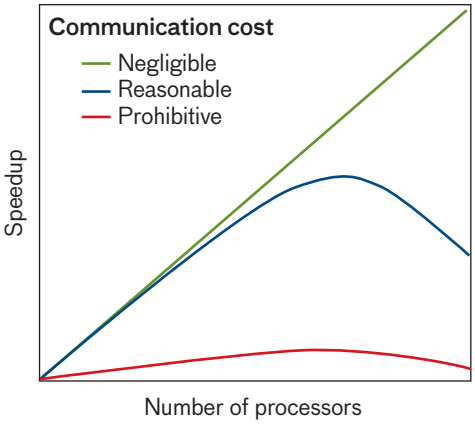

FIGURE 4. The speed benefit of parallization depends on the cost of communicating among processors. The blue curve is characteristic of most algorithms and is where finding communication/computation balance is most important.

Second, let us consider the details of the parallel implementation mapped onto two processors, as illustrated in Figure 3. Here, the time complexity is equivalent to computational complexity divided by two (the number of processors) plus the additional time needed to redistribute the data from rows to columns. Third, consider the same operation but using four processors. Although the computation time is reduced by a factor of four, there is now additional communication cost because the four processors need to exchange information with one another. Thus, as the number of processors increases, the work each processor must perform decreases, leading to speedup. At the same time, the need for interprocessor communication increases. Figure 4 illustrates the relationship between number of processors and speedup for varying communication costs.

This delicate balance of computation and communication is not captured through concurrency analysis. In particular, the signal flow graph of a serial program provides insufficient information to determine the optimal processor breakdown because the computation is no longer the only component and the network architecture and topology influence the execution time significantly. For example, on a slow network it might be beneficial to split the computation up between only a few nodes, while on a faster network the addition of more processors would provide greater speedup.

It is important to address the issue of computation/communication balance in a global manner. Techniques already exist for optimizing performance of single functions. FFTW (Fastest Fourier Transform in the West) [13] optimizes the performance of FFT routines, for example; ATLAS (Automatically Tuned Linear Algebra Software) [14] does the same for linear algebra routines. But determining what is the best mapping, or distribution, for groups of operations makes the problem more difficult—what might be an optimal processor breakdown or map for a single function might not be optimal for a chain of computations. Our research tackles the two issues described above: balancing communication and computation in an application, and doing so in a global manner, not just on a per function basis.

Taxonomy

It is helpful to classify existing approaches along with pMapper by using a common set of characteristics. The first characteristic is concurrency—that is, whether the approach is meant to optimize a program for a serial architecture or parallel architecture. If the concurrency is serial, then the approach entails finding an efficient mapping into the memory hierarchy of a single-processor machine, such as determining the optimal strategy for cache utilization. Commonly used compilers optimize performance at serial concurrency. If the concurrency is parallel, the approach needs to optimize the code for a parallel or distributed memory hierarchy. pMapper is a parallel-concurrency optimization framework.

The second characteristic is support layer—that is, in which layer of software the automatic distribution and optimization is implemented. Optimization approaches tend to be implemented in either the compiler or middleware layer. If the parallelization approach is implemented in the compiler, it does not have access to runtime infor-

* The complexity given is for an FFT of a complex vector of length N, where N is a power of 2.

mation—information that could significantly influence the chosen mapping. On the other hand, if the approach is implemented in middleware and is invoked at runtime, it could incur a significant overhead because of the extra effort required to collect the runtime information. pMapper is implemented in middleware.

The third characteristic is code analysis—that is, how the approach finds instances of parallelism. Code analysis can be static or dynamic. Static code analysis involves looking at the code as text and trying to extract inherent parallelism on the basis of how the program is written. Dynamic code analysis involves analyzing the behavior of the code as it is running, thus allowing access to runtime information. pMapper uses dynamic code analysis.

```
%Define maps
mapA = map([4 1],{},[0:3]); %distribute rows
mapB = map([1 4],{},[0:3]); %distribute cols
%Create arrays
A = array(N,M,mapA);
B = array(N,M,mapB);
C = array(N,M,mapB);
%Perform FFT along the 2nd dimension (row)
A(:,:) = fft(A,[],2);
%Corner-turn the data
B(:,:) = A;
%Perform FFT along the 1st dimension (col)
C(:,:) = FFT (B,[],1);
```

FIGURE 5. Sample pMatlab code for the parallel FFT. Mapping the code is independent from algorithm development. mapA and mapB could change arbitrarily, without changing the code's functionality or correctness.

The fourth characteristic of the parallelization taxonomy is the optimization window, or at what scope the approach applies optimizations. Approaches could be local (peephole) or global (program flow). Local optimization approaches, which find optimal distributions for individual functions, have had the most success and are used by many parallel programmers. Consider the FFT example from Figures 2 and 3. A local optimization approach would consider a single FFT computation at a time and would use all of the available resources. Locally, this is the optimal solution. However, it is often true that the best way to distribute individual functions is not the best way to distribute the entire program or even a portion of the program. Global optimization addresses this issue by analyzing either the whole program or a sub-program. For the FFT example, a global optimization approach would consider the cost of redistribution between row and column operations and would produce different results, depending on the underlying architecture. pMapper performs global optimization.

pMapper tackles a challenging problem space as specified by this taxonomy and is unique in that it performs both dynamic code analysis and global optimization. Table 1 summarizes the taxonomy and pMapper's defining characteristics.

Lincoln Laboratory Parallel Software

The pMapper framework globally optimizes performance of parallel programs at runtime. To do this, pMapper requires a presence of an underlying parallel library. The Embedded Digital Systems Group at Lincoln Laboratory has been developing parallel libraries for more than 10 years. The libraries developed here include STAPL (Space-Time Adaptive Processing Library) [15], PVL (Parallel Vector Library) [16], and pMatlab [17]. All of these libraries have increased the level of abstraction by implementing a map layer that insulates the algorithm developer from writing complicated message-passing code. These libraries introduce the concept of map independence—that is, the task of mapping the program onto a processing architecture is independent

Table 1: Taxonomy of Automatic Program Optimization
(pMapper characteristics in **bold**)

Concurrency	Serial	**Parallel**
Support layer	Compiler	**Middleware**
Code analysis	Static	**Dynamic**
Optimization window	Local (peephole)	**Global (program flow)**

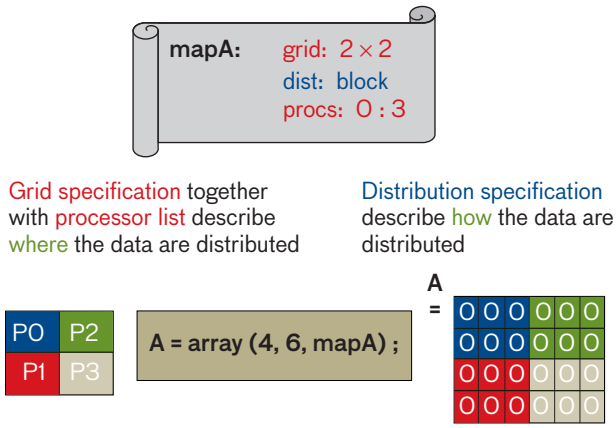

FIGURE 6. A map consists of the grid specification, distribution description, and processor list. Shown here is a 4 × 6 array mapped onto a 2 × 2 grid using processors 0, 1, 2, and 3.

from the task of algorithm development. Once the algorithm has been specified, the user can simply define maps for the program without having to change the high-level algorithm, as illustrated in the pMatlab example in Figure 5. Note that the maps can be changed without having to change any of the program details. The key idea behind map independence is that a parallel programming expert can define the maps, while a domain expert can specify the algorithm.

Let us quickly review the concept of a map. For the purposes of this article, a map is an assignment of blocks of data to processing elements. A map can be defined with three pieces of information: (1) grid specification, (2) distribution description, and (3) processor list. Figure 6 illustrates how a map distributes an array across processing elements. (For a detailed discussion of maps, see Reference 18.) Since the task of mapping the program is separated from the task of developing the algorithm, the entity that determines the maps for the program could be another layer of software. That is exactly the approach that our research explores.

Automatic Parallelization with pMapper

pMapper is an automatic mapping engine originally designed to distribute MATLAB programs onto parallel computers, specifically clusters such as the 1500-processor Lincoln Laboratory Grid (LLGrid), shown in Figure 7) [19]. Although the examples discussed here are written in MATLAB, the concepts are general with respect to programming languages and environments.

Consider the MATLAB code example in Figure 5. Although introducing the mapping layer significantly simplifies parallel programming, specifying the details of a map object is nontrivial. As discussed previously, choosing the wrong number of processors can slow down the program. Additionally, choosing an inefficient set of map components, such as grid, processor list, or distribution, could yield poor performance. A program coded to be optimized by pMapper has maps replaced with tags that indicate to the system that the arrays should be considered for distribution (as illustrated in Figure 8). The automatic mapping system finds efficient maps for all tagged arrays. As the tag indicates that the arrays should be considered for distribution and not necessarily distributed, some of the maps may contain only one processor.

Two-Phase Architecture

To provide accurate mappings, we need to collect benchmark performance data of the parallel library on the target parallel architecture. Specifically, pMapper needs

FIGURE 7. pMapper was originally developed and tested on Lincoln Laboratory's 1500-processor LLGrid system.

to have access to timings of various <*map, function*> pairs on the architecture. For the FFT function, for example, pMapper will collect information on the amount of time required to execute the FFT with maps with varying numbers of processors and different distributions on various grids. pMapper requires the presence of an underlying parallel library, such as pMatlab. This library provides parallel versions of the functions. The performance of these functions with various maps needs to be assessed prior to making mapping decisions.

```
%Initialize arrays
A = array(N,M,ptag);
B = array(N,M,ptag);
C = array(N,M,ptag);
%Perform computation
B(:,:) = fft(A,2);
C(:,:) = fft(B,1);
```

FIGURE 8. This pMapper code is functionally equivalent to the pMatlab code in Figure 5. The automatic mapping/parallelization system finds efficient maps for the tagged arrays.

The task of benchmarking the library is computationally intensive, making it infeasible to collect sufficient timing data during program execution. Once the benchmarking data have been collected, pMapper uses them to generate maps in an efficient manner. This process naturally yields a two-phase mapping architecture.

The initialization phase occurs once, when pMapper is installed on the target architecture—or, if the architecture is simulated, when the architecture parameters are first specified. The idea of collecting performance data to later aid in optimization can also be found in profile-guided optimization approaches. A key difference in the pMapper initialization step is that the performance data are collected on individual functions and not on the full program. The initialization process is therefore independent from the program, and dependent only on the underlying parallel library and parallel architecture.

Once the timing data are collected and stored as a performance model, they are used to generate maps for the tagged numerical arrays. The mapping and execution phase illustrated in Figure 9 is performed once for each program at runtime. pMapper uses lazy evaluation—that is, it delays execution until necessary. This approach allows pMapper to have the greatest possible amount of information about the program to be mapped at mapping time. pMapper thus has an advantage over compiler approaches, which analyze the code before runtime and might not have access to as much information. Once execution is required, pMapper considers all of the functions up to the point of execution. In the example in Figure 8, pMapper would have access to both FFTs at the time of mapping and could therefore perform global optimization.

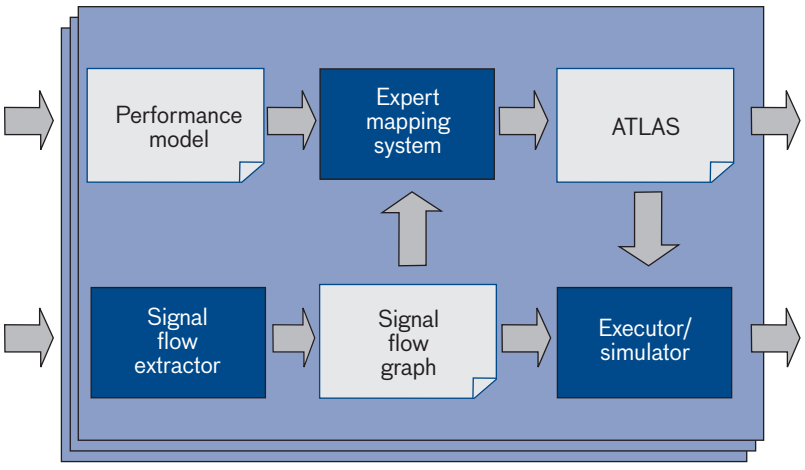

FIGURE 9. Block diagram shows pMapper mapping and execution. The performance model is created during initialization. The signal flow graph is extracted from the program, observing the lazy evaluation policy (i.e., delaying the computation until the result of the computation is needed). The signal flow graph is then mapped to produce an atlas—a collection of maps for the program. Finally, the program is executed (or simulated) on the underlying architecture.

Results

To validate pMapper performance, we used the pMapper framework to determine efficient mappings of the High Performance Embedded Computing (HPEC) Challenge benchmarks [20]. This benchmark suite [21] was developed to provide quantitative evaluation of multiprocessor systems. It consists of signal-processing kernels, knowledge-processing kernels, and a compact application based on an implementation of synthetic aperture radar (SAR).

Table 2 summarizes the results. The kernel benchmarks can be divided into two categories:

1. Ones requiring no interprocessor communication (known as embarrassingly parallel): finite impulse response

(FIR), constant false-alarm rate (CFAR), and pattern match;

2. Ones that do require interprocessor communication: singular value decomposition (SVD), QR matrix factorization, genetic algorithm, and database operations.

All of the kernel benchmarks were mapped onto a simulated IBM Cell processor. The SAR compact application benchmark, for which the results are also shown, was mapped and executed on the LLGrid cluster.

Consider the CFAR benchmark. pMapper distributed the computation row-wise among eight processors. This benchmark, along with the FIR filter and the pattern match benchmark, is embarrassingly parallel and benefits from the use of the maximum number of processors. While these are straightforward to parallelize, they provide initial evidence that the pMapper approach finds efficient mappings.

Table 2: Results of the HPEC Challenge

BENCHMARK	SPEEDUP	NUMBER OF PROCESSORS
FIR (finite impulse response)	8	8
CFAR (constant false-alarm rate)	8	8
SVD (singular-value decomposition)	6.7	8
QR factorization	2.6	8
Pattern match	8	8
Genetic algorithm	2.8	8
Database operations	3.8	8
Application (synthetic aperture radar)	17	23

TABLE 2. The first seven High Performance Embedded Computing (HPEC) Challenge benchmarks are mapped onto a simulated IBM Cell processor with the maximum of eight processing elements. The last benchmark was mapped and executed on the LLGrid. Speedup is defined as serial execution time divided by parallel execution time.

On the other hand, QR factorization benchmark achieves speedup of only 2.6 and uses only six out of eight available processors. The parallel QR algorithm is communication intensive; even on a low-latency machine the communication takes a significant amount of time as compared with the computation. The traversal of the mapping space for various matrix sizes (Figure 10) for QR illustrates pMapper's capability to effectively navigate the search space and balance communication and computation. Comparison of pMapper maps to those which an expert would choose indicates that pMapper is able to generate efficient maps for this benchmark.

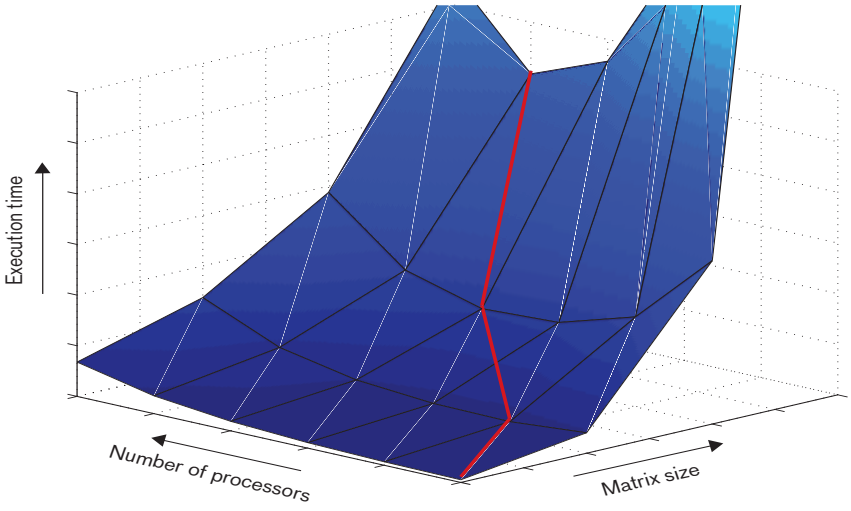

FIGURE 10. pMapper searches for the best mapping for various matrix sizes. The red line indicates the mappings chosen for a particular matrix size.

The results are similar for the genetic algorithm benchmark. This benchmark consists of a number of operations, some computing local results and others requiring communication among processors. pMapper performs global analysis and thus chooses to use only three out of the available eight processors. A mapping using more than three processors for this benchmark would more than offset the gain in computation time by additional communication time.

Architecture Modeling with pMapper

In addition to mapping computations onto architectures, the pMapper technology provides another valuable capability: processor sizing and architecture analysis. pMapper uses a machine model abstraction of the system to determine the mappings. The machine model can be of an existing architecture or of a new architecture design. The mappings and predicted performance of various kernels and applications onto the machine can determine optimal sets of architecture parameters. Parameters that can be analyzed by the framework include latency, bandwidth, number of interconnects, and CPU speed.

Figure A illustrates the architecture analysis capability. The machine abstraction and the application specification are inputs into the pMapper framework. As output, pMapper generates maps for the application. Additionally, pMapper generates performance estimates of the application running on the architecture. This process is repeated for ranges of computer architecture parameters, allowing optimal parameters and suitability of processing architecture to be determined. One current research direction is to improve pMapper's ability to model and optimize architectures.

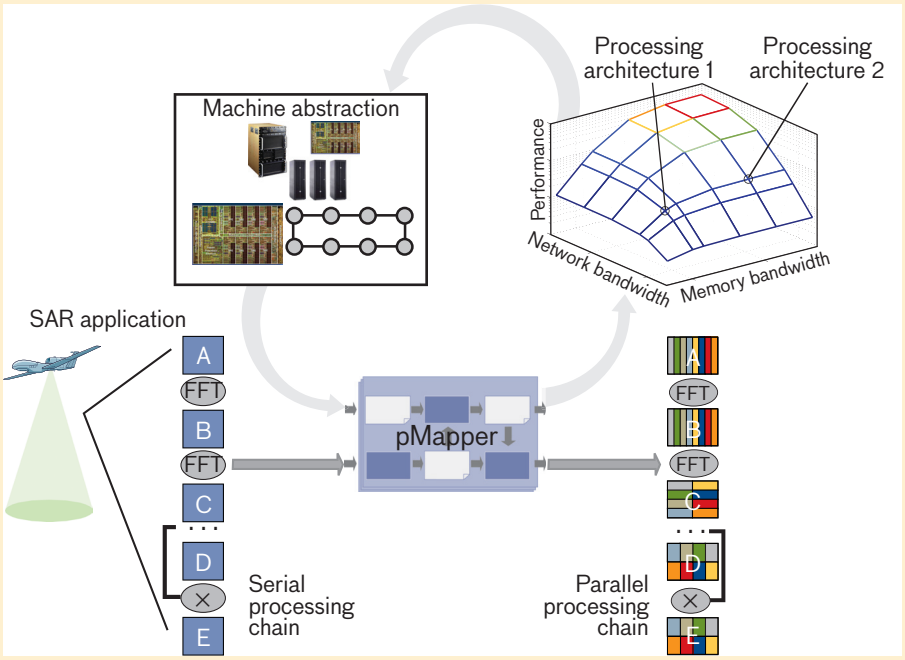

FIGURE A. pMapper's performance feedback allows for update of architecture parameters to better meet application space needs. Here, pMapper considers the effect of network and memory bandwidth on the fast Fourier transform (FFT) performance of a synthetic aperture radar (SAR) image processing application.

The SAR benchmark, unlike the others, was not simulated to run on the IBM Cell architecture; instead, it was mapped and executed on LLGrid, where with 23 processors it achieved a speedup of 17. This benchmark demonstrates pMapper's ability to handle large-scale applications. Reference 22 validates the pMapper result: the hand-coded implementation achieves similar speedup to the automatic parallelization.

Future Directions

Since the early 1980s, industry and academia have been developing techniques to ease the programming of parallel computing systems and allow a wide range of users to benefit from the power these systems provide. Nonetheless, parallel computing has remained the domain of specialized experts. That can no longer be the case. From the LLGrid cluster, which has more that 200 users at Lincoln Laboratory, to the IBM Cell processor, which promises 256 gigaflops of computing power with eight specialized cores, to Intel's desktop quad core processor, parallel computers are here. Lincoln Laboratory has made substantial contributions to research on parallel libraries and standards. Automatic program parallelization and optimization are the next steps. pMapper is an important

research development in both automatic program parallelization and analysis of parallel systems (see the sidebar "Architecture Modeling with pMapper").

Additional research directions include extending pMapper to architectures based on field-programmable gate arrays (FPGAs) and to knowledge-based computations. FPGAs can provide significant performance increase over general-purpose processors, yet they are expensive to program. FPGAs require finer-grained analysis of applications; specifically, algorithm kernels have to be broken down into individual operations (such as adds and multiplies), whereas general-purpose processors can use coarser-grained, kernel level analysis. Prototype pMapper capability has been developed to perform fine-grained program analysis; however, the current implementation is computationally intensive. If the computation-to-FPGA mappings will be reused many times, the high cost of finding the mapping is acceptable. On the other hand, if mappings need to be dynamic, the computational complexity of the approach must be reduced. One way to address this problem is to recognize similar parts of the computation during program analysis and reuse previously found mappings. We are exploring this approach.

Additionally, to accurately produce mappings for an FPGA architecture, we need a more detailed model of the underlying architecture. A pMapper prototype has been developed to not only allow detailed architecture analysis, but also to allow for automatic mapping onto heterogeneous architectures. Because FPGAs are often used as part of a larger system (as accelerator processors), an automatic mapping capability for heterogeneous systems is very valuable.

Another direction being pursued is program analysis and mapping of back-end, knowledge-based algorithms. Currently, signal processing is done at the front end of the sensor system. The processed data are then passed on to the back-end processor to perform further analysis, such as anomaly detection, target identification, and social network analysis. As the raw data set sizes become exceedingly large, however, real-time knowledge processing will become essential. Many of these algorithms are based on graph algorithms, which in turn can be cast as sparse matrix operations. Sparse algorithms often perform poorly on parallel machines because of many irregular data accesses. Automatic optimization of these algorithms would greatly improve performance. Additionally, it is worthwhile to explore architecture properties that allow for more efficient sparse computations. The pMapper automatic parallelization framework will assist in both of those research initiatives.

Acknowledgements

A number of individuals have contributed to this work. I particularly want to thank Hank Hoffmann, who was the initial collaborator on the pMapper project, and Sanjeev Mohindra, who contributed immensely to this work. Jeremy Kepner and Robert Bond provided valuable insight, guidance, and support. I also thank Ken Senne and Zach Lemnios for funding this work, and everyone on the LLGrid team for providing a tremendous resource for both algorithm development and pMapper testing. ■

REFERENCES

1. Intel CoreTM Microarchitecture, www.intel.com/technology/architecture-silicon/core.
2 The Cell project at IBM Research, www.research.ibm.com/cell.
3. G.E. Moore, "Cramming More Components onto Integrated Circuits," *Electronics,* vol. 38, no. 8, 1965.
4 N. Travinin Bliss and J. Kepner, "pMatlab Parallel Matlab Library," *Int. J. High Perform. Comput. Appl.,* vol. 21, no. 3, 2007 (special issue on high productivity programming languages and models), pp. 336–359, hpc.sagepub.com/cgi/reprint/21/3/336.
5. J. Lebak, J. Kepner, H. Hoffmann, and E. Rutledge, "Parallel VSIPL++: An Open Standard Software Library for High-Performance Parallel Signal Processing," *Proc. IEEE,* vol. 93, no. 2, 2005, pp. 313–335.
6. UPC Community Forum. www.upc.gwu.edu.
7. K. Yelick, P. Hilfinger, S. Graham, et al., "Parallel Languages and Compilers: Perspective from the Titanium Experience," *Int. J. High Perform. Comput. Appl.,* vol. 21, no. 3, 2007 (special issue on high productivity programming languages and models), pp. 266–290, hpc.sagepub.com/cgi/content/abstract/21/3/266.
8. M. Wolfe, *High Performance Compilers for Parallel Computing* (Benjamin/Cummings, Redwood City, Calif., 1996).
9. R. Gupta, E. Mehofer, and Y. Zhang, "Profile Guided Compiler Optimization," chap. 4 in *The Compiler Design Handbook: Optimization and Machine Code Generation,* Y.N. Srikant and P. Shankar, eds. (CRC Press, Boca Raton, Fl., 2002).
10. G.M. Amdahl, "Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities," in *AFIPS Conf. Proc.,* 1967, pp. 483–485.
11. J.L. Hennessy and D.A. Patterson, *Computer Architecture: A Quantitative Approach,* 4th ed. (Morgan Kaufman Publishers, San Francisco, 2006).

12. Yelick, "Parallel Languages and Compilers."
13. M. Frigo and S.G. Johnson, "FFTW," www.fftw.org.
14. A. Petitet, R.C. Whaley, and J.J. Dongarra, "Automated Empirical Optimizations of Software and the ATLAS Project," *Proc. High Performance Embedded Computing Workshop (HPEC 2000), Lexington, Mass., Sept. 20–22, 2000.*
15. C.M. DeLuca, C.W. Heisey, R.A. Bond, and J.M. Daly, "A Portable Object-Based Parallel Library and Layered Framework for Real-Time Radar Signal Processing," *Proc. 1st Conf. International Scientific Computing in Object-Oriented Parallel Environments (ISCOPE '97),* 1997, pp. 241–248.
16. Lebak, "Parallel VSIPL++."
17. J. Kepner and N. Travinin, "Parallel Matlab: The Next Generation," *Proc. High Performance Embedded Computing Workshop (HPEC 2003), Lexington, Mass., Sept. 23–25, 2003.*
18. N.T. Bliss, R. Bond, J. Kepner, H. Kim, and A. Reuther, "Interactive Grid Computing at Lincoln Laboratory," *Linc. Lab. J.,* vol. 16, no. 1, 2006, pp. 165–216.
19. Ibid.
20. N.T. Bliss, J. Dahlstrom, D. Jennings, and S. Mohindra, "Automatic Mapping of the HPEC Challenge Benchmarks," *Proc. High Performance Embedded Computing Workshop (HPEC 2006), Lexington, Mass., Sept. 19–21, 2006.*
21. R. Haney, T. Meuse, J. Kepner, and J. Lebak, "The HPEC Challenge Benchmark Suite," *Proc. High Performance Embedded Computing Workshop (HPEC 2005), Lexington, Mass., Sept. 20–22, 2005.*
22. J. Mullen, T. Meuse, and J. Kepner, "HPEC Challenge SAR Benchmark pMatlab Implementation and Performance," *Proc. High Performance Embedded Computing Workshop (HPEC 2006), Lexington, Mass., Sept. 23–25, 2006.*

ABOUT THE AUTHOR

Nadya Bliss is a staff member in the Embedded Digital Systems group. Since joining the Laboratory in 2002, she has been a principal developer of pMapper, the automated parallelization system, and pMatlab, the parallel MATLAB toolbox. Her research interests are in parallel and distributed computing and intelligent/cognitive algorithms. She has a master's degree in computer science from Cornell University.