Software Technologies for High-Performance Parallel Signal Processing

Jeremy Kepner and James Lebak

Real-time signal processing consumes the majority of the world's computing power. Increasingly, programmable parallel processors are used to address a wide variety of signal processing applications (e.g., scientific, video, wireless, medical, communication, encoding, radar, sonar, and imaging). In programmable systems the major challenge is no longer the speed of the hardware but the complexity of optimized software. Specifically, the key technical hurdle lies in mapping an algorithm onto a parallel computer in a general manner that preserves performance while providing software portability. We have developed the Parallel Vector Library (PVL) to allow signal processing algorithms to be written with high-level mathematical constructs that are independent of the underlying parallel mapping. Programs written using PVL can be ported to a wide range of parallel computers without sacrificing performance. Furthermore, the mapping concepts in PVL provide the infrastructure for enabling new capabilities such as fault tolerance and self-optimization. This article discusses PVL with a particular emphasis on quantitative comparisons with standard parallel signal programming practices.

Real-time signal processing is critical to a wide variety of applications, including radar signal processing, sonar signal processing, digital encoding, wireless communication, video compression, medical imaging, and scientific data processing. These applications feature large and growing computation and communication requirements that can be met only through the use of multiple processors and fast low-latency networks. A balance between computation and communication is one of the key characteristics of this type of processing. Advanced software techniques are needed to manage the large number of processors and complex networks required and achieve the desired performance.

Military sensing platforms employ a variety of signal processing systems at all stages: initial target detection, tracking, target discrimination, intercept, and engagement assessment. Although high-performance embedded computing is widely used throughout commercial enterprises, the Department of Defense (DoD) often has the most demanding requirements, particularly for radar, sonar, and imaging sensor platforms. Figure 1 shows many of these platforms, along with a graph that illustrates the growth in computational requirements for future real-time signal processing applications. The challenge for these systems is the cost-effective implementation of complex algorithms on complex hardware. This challenge is made all the more difficult by the need to stay abreast of rapidly changing commercial off-the-shelf (COTS) hardware. The key to meeting this challenge lies in utilizing advanced software techniques that allow new hardware to be inserted while preserving the software investment.

FIGURE 1. Military platforms are among the largest drivers of embedded real-time signal processing applications. Department of Defense (DoD) systems in the 2010 time frame are projected to require multiple teraflops of computation.

The standard approach to writing high-performance signal processing applications has been to use vendor-supplied computation and communication libraries that are highly optimized to a specific vendor platform and guaranteed to deliver the highest possible performance. Unfortunately, this approach usually leads to software that is difficult to write, is not compatible with other platforms, and is dependent on the number of processors on which the application is deployed.

The Parallel Vector Library

Our answer to the challenge of writing reusable software has been to develop the Parallel Vector Library (PVL). The major goals of the library are (1) to make the job of writing distributed signal processing software easier, (2) to provide for portability of application level code, (3) to separate the job of mapping distributed data and computations from the job of signal processor development, and (4) to achieve high performance on many different parallel platforms.

The primary technical challenge in deploying parallel processing systems is the assignment of different parts of the algorithm to processors in the parallel computing hardware. This process, illustrated in Figure 2, is referred to as *mapping*. The mapping of an application needs to be optimized for each system the application runs on. Thus, for a program to run on a variety of systems, the algorithm description must not depend on details of the parallel hardware; that is, it must be *map independent*.

Map independence is an important element of portability that goes beyond the simple, standards-based portability of recent efforts. Standards such as the Message-Passing Interface (MPI) [1] and the Vector, Signal, and Image Processing Library (VSIPL) [2] provide a *portable interface* to parallel applications, that is, one that is the same for different platforms. However, unless special care is taken with the design of a program using MPI and VSIPL, the application code must be modified every time the number of hardware elements used changes, as illustrated in Figure 3. Such modifications may be required, for example, when moving the application program from a single-processor prototyping environment to a full-scale deployed parallel system, or when moving an ap-

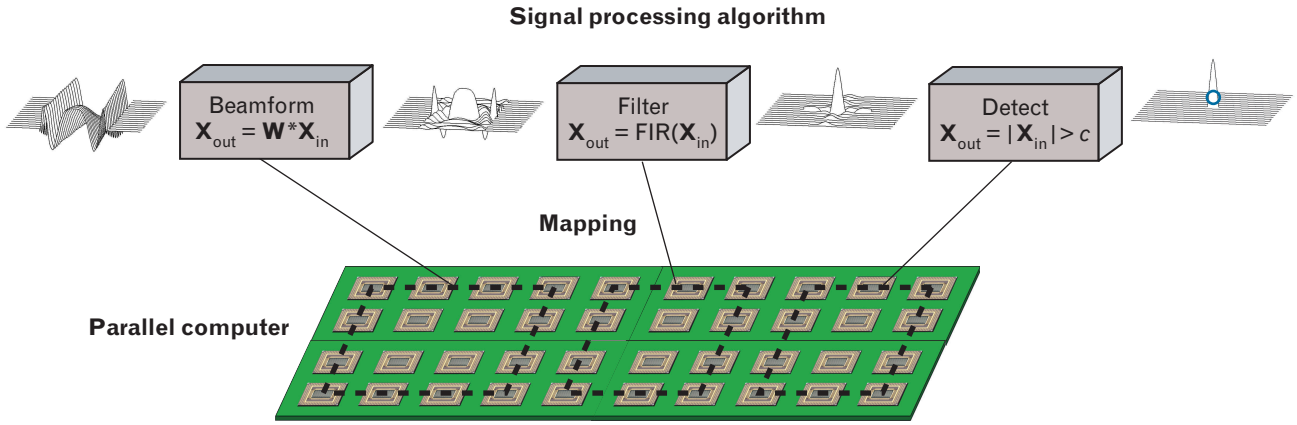


FIGURE 2. Parallel mapping is the process of assigning different parts of an algorithm to separate processing elements in the parallel-computing hardware. Mapping needs to be optimized for each system the application runs on. Here a three-stage pipeline, consisting of a beamformer, a finite impulse response (FIR) filter, and a detector, is mapped to distinct sets of processors in a parallel computer. A program that runs on a variety of systems, and doesn't depend on the specific details of the parallel hardware implementation, is said to be *map independent*.

plication from one generation of technology to another. PVL avoids the need for such modifications through the concept of map independence.

Map independence is achieved by raising the level of abstraction at which signal processing applications are written. In current practice, it is common for the application writer to have to deal with multiple objects describing the same memory area. For example, a buffer containing a vector may need an MPI object for communication operations and a VSIPL object

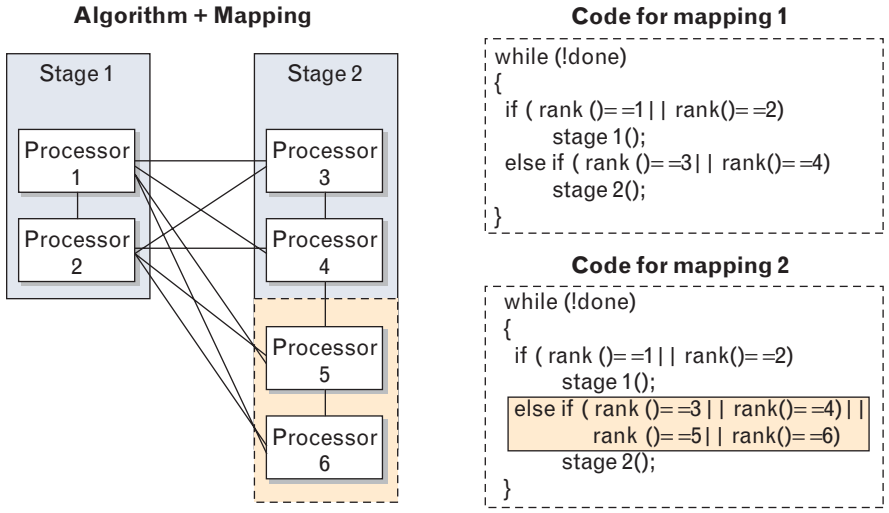


FIGURE 3. The challenge of portability and scalability in parallel systems. Consider a basic algorithm and mapping, where Stage 1 is mapped to processor nodes 1 and 2 and sends data to Stage 2, which is mapped onto nodes 3 and 4. Existing software approaches typically hard-code the processor mapping information directly into the application, and thus require significant changes to run the application on a different number of processors. When processor nodes 5 and 6 are added to Stage 2, the code for the new mapping must be modified, as shown above. The challenge is to create algorithms that are map independent, and don't need modifications when moved to different processing systems.

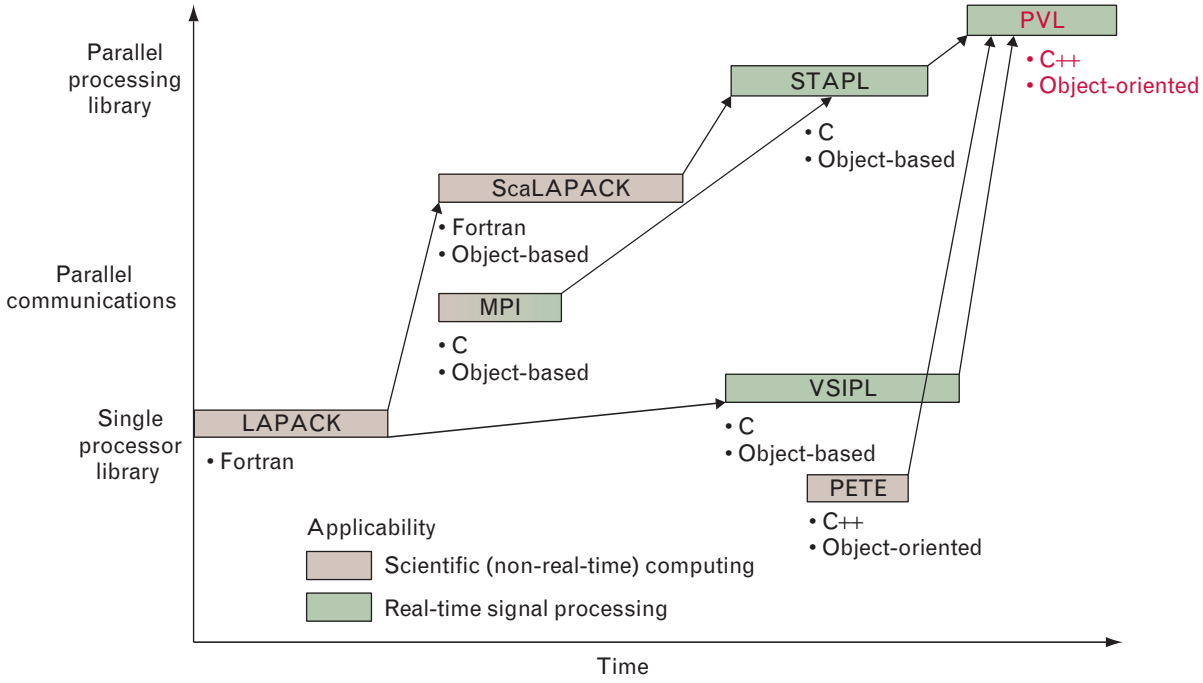


FIGURE 4. Evolution of the parallel vector library (PVL). Over the past decade, there has been a continuous transition of computing technology from non-real-time scientific computing to the real-time domain, and the development focus has shifted from procedural languages such as Fortran to object-oriented languages such as C++. Thus much of the functionality of the Fortran-based linear algebra package (LAPACK) was made suitable for embedded computing in the Vector, Signal, and Image Processing Library (VSIPL). The deployment of the Message-Passing Interface (MPI) made possible the development of the Fortran-based scalable version of LAPACK, called ScaLAPACK. PVL, which evolved from the earlier C-based parallel Space-Time Adaptive Processing Library (STAPL), builds on MPI and VSIPL in a way analogous to the way ScaLAPACK builds on MPI and LAPACK, also incorporating new object-oriented technologies such as those in the portable expression template engine (PETE) that makes mathematical programming easier and more efficient.

for computation operations. Coordinating these descriptions can be a tedious and error-prone task. PVL provides objects useful for both computation and communication, and handles the coordination of these operations without explicit direction from the application programmer. This coordination has the added benefit of making the job of writing distributed signal processing software easier.

PVL is based on earlier work on the Space-Time Adaptive Processing Library (STAPL), which was used to field a 1000-CPU embedded signal processor [3]. PVL incorporates the lessons learned from the STAPL project into a new object-oriented library written in the C++ programming language. C++ was selected because of its growing acceptance in the embedded community and its powerful abstraction capabilities that allow more to be done with fewer lines of code. Figure 4 shows how PVL's capabilities build on industry standards and previous libraries. The base version of PVL is actually built on the open standards of VSIPL and MPI to provide an added level of portability. PVL has been successfully implemented on a wide range of workstation, cluster, and embedded architectures, as illustrated in Figure 5.

In summary, PVL achieves its goals of increasing productivity and portability by allowing the independence of map and application and increasing the level of abstraction. We discuss the implementation of these concepts in more detail in the following section on the PVL programming model. The subsequent section on PVL performance shows how we achieve these goals without sacrificing our primary goal of high performance. A later section describes ongoing research involving PVL.

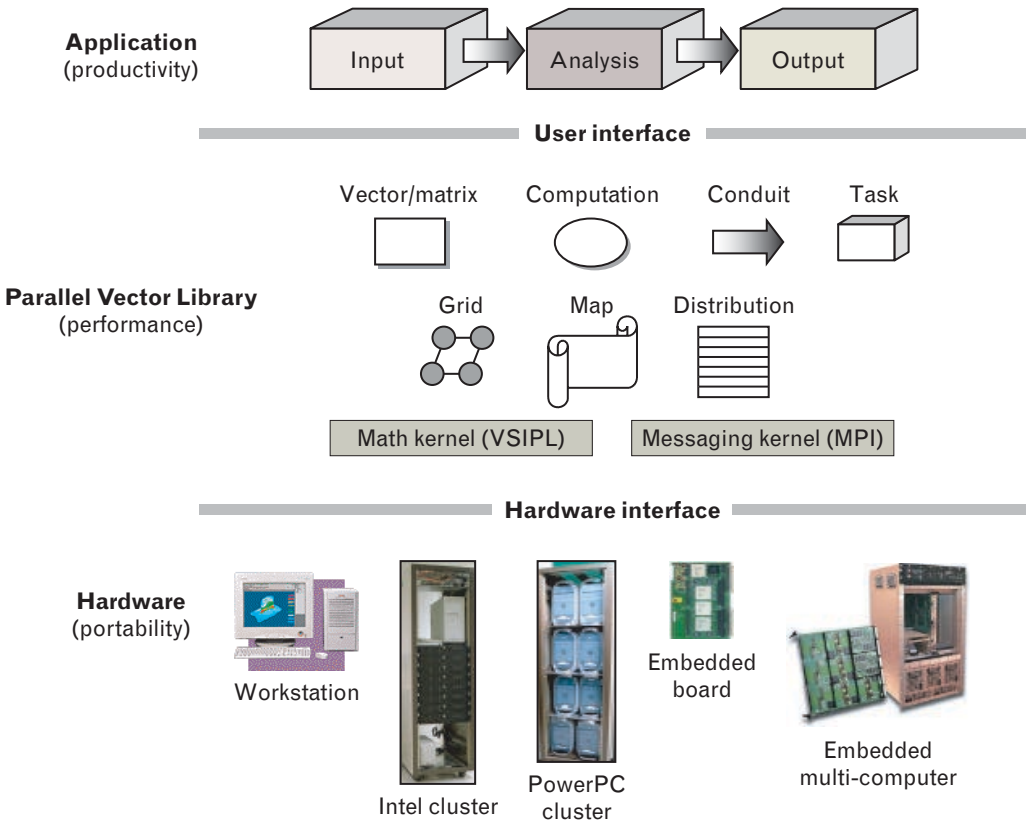


FIGURE 5. PVL layered architecture. The main purpose of a software library as middleware is to insulate the application from the hardware. The user writes an application in the high-productivity layer. The middleware implements constructs necessary to deliver high performance on parallel computers. The portability of the middleware is achieved by isolating the hardware-specific details to a math kernel such as VSIPL and a messaging kernel such as MPI.

The PVL Programming Model

Real-time signal processing applications generally make use of two types of parallelism. The first is data-parallel operations on vectors and matrices, which means operating on a vector or matrix that is itself distributed. The second is task-parallel operation, which may be used to construct pipelines or to process incoming data sets in a round-robin fashion. A PVL program can make use of both of these types of parallelism.

The core capability of PVL is that of assigning, or mapping, the portions of a parallel program to be executed on or stored in specific components of the machine. The assignment information is contained in a map object. PVL programs are written using four user-level signal processing and control objects: data objects (i.e., vectors and matrices), computations, tasks, and conduits. These objects are illustrated in Figure 6. The mapping of the user-level objects to processors is controlled by the mapping objects shown in the same figure.

In broad terms, users obtain task parallelism by writing an application as a set of tasks and conduits. A task represents a scope for a particular computation stage; the effect of multiple tasks is to allow different processor groups to perform different computations at the same time. Within a task, operations are performed with vectors, matrices, and computations, using a single-program multiple-data paradigm that allows for data parallelism. Conduits allow communication of data objects between different scopes or tasks.

The overall effect of these objects is to make it possible to write multistage algorithms independent of the underlying hardware. For example, a simple data

	Class	Description	Parallelism
Signal processing and control	Vector/matrix	Used to perform matrix/vector algebra on data spanning multiple processors	Data
	Computation	Performs signal/image processing functions on matrices/vectors (e.g., FFT, FIR, QR decomposition)	Data and task
	Task	Supports algorithm decomposition (i.e., the boxes in a signal flow diagram)	Task and pipeline
	Conduit	Supports data movement between tasks (i.e., the arrows on a signal flow diagram)	Task and pipeline
Mapping	Map	Specifies how tasks, vectors/matrices, and computations are distributed on processors	Data, task, and pipeline
	Grid	Organizes processors into a two-dimensional layout	

FIGURE 6. PVL objects. PVL is based on four basic user-level signal processing and control objects (highlighted in color): vectors and matrices, computations (for example, a fast Fourier transform, or FFT), tasks, and conduits. Each of these is independently mappable onto a grid of processors.

parallel implementation of a basic frequency-domain filter can be constructed, as shown in Figure 7(a). A complete filtering system can then be constructed by inserting the basic filtering algorithm into a task and adding appropriate input tasks and output tasks connected by conduits, as shown in Figure 7(b). In each case, the application code does not include any references to the number of processors being used and can therefore be mapped to any parallel architecture.

In the following sections, we give more detail re-

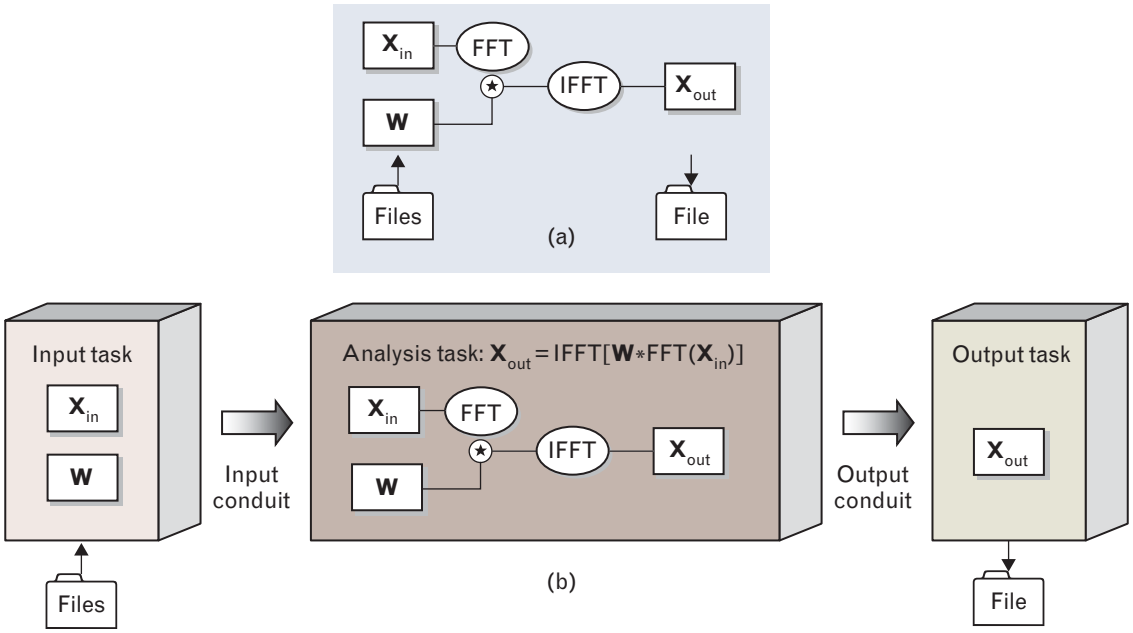


FIGURE 7. (a) Development of a basic frequency-domain filtering application, using an FFT and an inverse FFT (IFFT) on input data. PVL vector/matrix and computation objects allow signal processing algorithms to be implemented quickly by using high-level constructs. (b) PVL tasks and conduit objects allow complete filtering systems to be built.

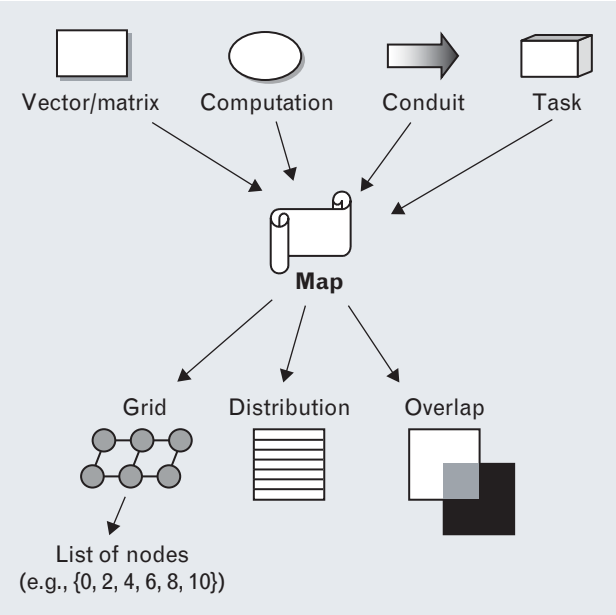


FIGURE 8. Structure of a PVL map object. All PVL user objects contain a map. Each map is composed of three components: a grid, a distribution description, and an overlap description. Within the grid is the list of physical nodes onto which the PVL object is mapped.

garding the mapping, data parallel features, and task parallel features of PVL.

Maps

Maps consist of a set of processor nodes and a description of how those nodes are to be used. Objects that can have a map are called *mappable*. There are three general classes of mappable type, and three corresponding types of maps. User functions, data, and calculations are represented in the library by tasks, distributed data objects, and distributed computation objects, respectively. Conduits are not themselves mappable types, but each endpoint of a conduit is a task object.

The map contains all information specific to how the object is placed onto the processors. Within each map is a two-dimensional grid with a specific list of nodes and a distribution description specific to the type of object. For example, a distribution description for a data object such as a matrix or vector determines how data are to be distributed among processors (block, cyclic, or block-cyclic) and includes an overlap description that describes whether elements should be duplicated on multiple processors (this may be done, for example, to support boundary conditions). Figure 8 shows the structure of a map object.

Data Parallel Operations

A PVL application program performs mathematical operations on distributed data objects (matrices and vectors). These operations are written at a high level, using mathematical expressions. Objects are assigned to particular nodes by using the map object. The distribution descriptions for data objects allow the user to specify an arbitrary block-cyclic distribution for each matrix or vector. (For a complete description of the block-cyclic data distribution, see the high-performance Fortran specification [4].)

As an example, Figure 9 shows three simple distributions of a vector of length eight: (a) a *block* distribution on three processors, (b) a block distribution over two processors, and (c) a *cyclic* distribution over four processors. The application program does not

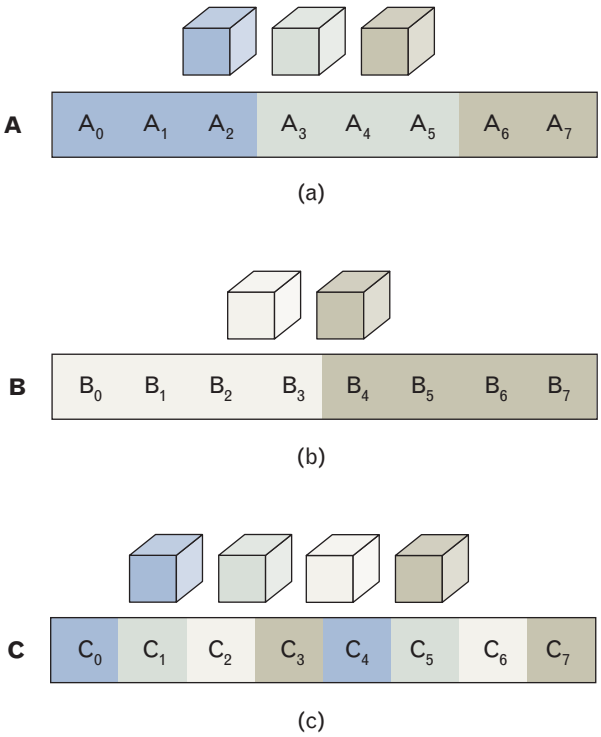


FIGURE 9. Example distributions of a vector of length eight: (a) block distribution on three processor nodes, (b) block distribution on two processor nodes, (c) cyclic distribution on four processor nodes.

explicitly specify the distribution, but merely includes a reference to the map, which is itself stored in an external file. The map can be changed without changing the application code. Also, vectors or matrices used in an expression are not required to have the same distribution. In the expression **C** = **A** + **B**, the vectors **A**, **B**, and **C** are not required to be distributed in the same way (though there may be compelling performance advantages to doing so).

PVL computation objects allow complex operations such as a fast Fourier transform (FFT) to be set up in advance and to be executed in a parallel manner. Advance setup, or *early binding*, of computation is common practice in embedded signal processing. As with computation objects (such as the FFT) in VSIPL, distributed computation objects provide a convenient place for the library to store constants and precalculated data without exposing them to the application programmer.

Furthermore, these objects allow computations to be mapped to special-purpose hardware, and they manage the communication into and out of that hardware. The map associated with the distributed computation object specifies where the computation takes place. It may also specify characteristics of the algorithm to be used to perform the computation; for example, the particular FFT algorithm to be used may be specified in the map of an FFT object.

Task Parallel Operations

A PVL program consists of a set of tasks connected by conduits. Within a task, an application is written using distributed matrix and vector objects as previously described. Operations within a task are written using a single-program multiple-data paradigm. Distributed data objects are communicated between different tasks by using conduits. An individual data object exists only in the scope of one particular task at a time; by using a conduit, the data contained in these items can be sent to and received from other tasks in a non-blocking way, possibly with multibuffering. The use of the conduit abstraction separates the application programmer from the details of the communication service being used, and provides more opportunity to optimize the communication operation.

Conduits are critical because, in many signal processing applications, the primary challenge is not performing the computations but moving data so that the data are in a position to be acted upon. One of the most common examples of this data movement is the corner turn operation, which, in its simplest form, can be described as a matrix transpose operation. This operation is performed on almost every system with multiple input channels (such as radar, sonar, and communications); typically, processing within a particular channel is followed by processing that cuts across channels.

For a two-dimensional data object, a corner turn is necessary when two consecutive operations require access to data in different ways, one by rows and one by columns. A corner turn between the two operations permits each operation to operate on data stored in a favorable access pattern, maximizing memory performance. In a local corner turn, data associated with an object are merely moved in memory. In a distributed corner turn, all-to-all communication between processor groups may be required in addition to local data movement. By using the conduit object, the corner turn is no more difficult for the programmer to orchestrate than any other data movement.

Efficient performance of a collective communication operation requires a considerable amount of setup or early binding of communication. At program setup time, a PVL application program specifies the tasks that each conduit connects. During this connection operation, the PVL conduit object can compute in advance the source and destination of all messages, set up all necessary buffers, create multiple buffers if necessary, and set up special hardware for communication.

PVL Benefits

PVL supports both task and data parallelism. Data parallelism is a well-understood way to achieve speedup for dense matrix calculation. The major benefit of PVL's task-parallel features is that the stages may be run on separate hardware (for example, separate processors of a parallel machine). By breaking the system up into multiple stages, the system designer gains an extra level of parallelism. This process, which is referred to as *pipelining*, allows customization of the system resources for each individual stage. An alterna-

tive approach would be to have all the system resources work on an individual data set in turn, before going on to the next one. The latency for each individual data set processed by the system will probably be greater in the pipelined scheme than in the uniform processing scheme. However, an increase in throughput is attained with the pipelined approach because several data sets are being processed at the same time. In addition, gains in efficiency may be realized because of the smaller number of processing resources assigned to each stage, and because communication may occur concurrently with computation.

Another major benefit of PVL is the portability and reusability of software modules. Beyond the obvious benefits for technology refresh and system upgrades, this eases system development, because application writers can gradually introduce parallelism. An application may be tested with single-processor maps and then retested with the maps for a parallel processor after the code has been verified. The separation of application and mapping also makes it easier to debug on inexpensive platforms such as networks of workstations, reducing contention for embedded target hardware.

From a software engineering perspective, the use of tasks and conduits in the PVL programming model provides an effective framework for integrating modules developed by different software engineers. Application developers can concentrate on the signal processing and linear algebra requirements of their piece of the algorithm independently of other pieces.

Performance Results

Parallel machines have various complex memory hierarchies, and the capabilities of the individual processors vary greatly from machine to machine. Ideally, mapping the application to the hardware should be entirely separate from the application code writing. As we have described, PVL uses abstractions to hide the details of single-processor computation and the transfer of data between processors, allowing the application to become scalable to different numbers of processors. These same abstractions therefore provide for portability of the application between different parallel machines or succeeding generations of the same parallel machine.

Use of a high level of abstraction may have two effects on performance. High-level objects may introduce overhead and reduce performance; however, a high level of abstraction may also enable library optimizations to be hidden from the application programmer. In this section, we describe the performance of PVL from both of these perspectives. First, in the following subsection, we demonstrate that PVL does not introduce an unacceptable amount of computational overhead for complex FFT operations. Then, in the subsequent subsection, we demonstrate that the level of abstraction introduced in PVL can result in improved performance in some cases.

Abstraction and Overhead

A major goal of PVL is that a reasonably mapped application should be able to achieve performance close to that which can be achieved by native mathematical and message-passing libraries on a given platform. To examine our success in this goal, we measured the amount of overhead PVL adds to an optimized FFT operation. This section describes these performance results in more detail.

Understanding the PVL FFT. The use of the PVL FFT can be divided into three phases: declaration, setup, and execution. The distinguishing characteristics of the PVL FFT are that the input vector and the output vector each have a map associated with them that describes how they are distributed, and that, in addition, a *computation map* is associated with the FFT object that can be different from either the input or the output vector map. The computation map describes how and where the computations take place.

Methodology. We measured computation times for out-of-place complex-to-complex vector FFT operations using vectors of length $L = 2^n$, $n \in \{8, 9, \ldots, 14\}$ (i.e., L ranged from 256 to 16,384). These numbers were picked because they reflect a good range of interest for the embedded processing space. Each function of interest was run a given number of iterations and the total time was recorded. The first iteration was not counted because cache effects were likely to be most pronounced the first time the function was called. Measured times considered only the computation portion of the program, and did not include time to create or construct the FFT object, allocate

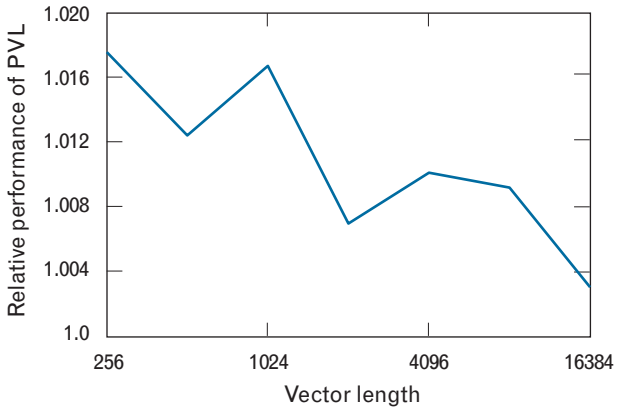


FIGURE 10. Overhead for PVL fast Fourier transform (FFT) execution time, relative to that of the Fastest Fourier Transform in the West (FFTW) library for vectors of various lengths. In all cases the overhead is less than 2%.

memory for the vectors, or perform other functions. Computation rates were calculated by assuming $5nL$ floating-point operations per complex FFT. In all cases we ran the measurement program six times, averaged the time over a thousand iterations each time, and used the minimum of the six results.

Performance Results. The PVL FFT object whose performance is measured here used the Fastest Fourier Transform in the West (FFTW) library [5]. Because this library is a collection of self-optimizing FFT routines that has been shown to outperform vendor-optimized libraries in some cases, we can be confident that it represents a well-performing FFT operation. Figure 10 shows the execution time of PVL relative to FFTW. It is clear that PVL adds very little overhead to the underlying FFTW call.

Abstraction and Optimization

PVL is implemented by using the C++ programming language, which allows the user to write programs using high-level mathematical constructs such as

$$\mathbf{A} = \mathbf{B} + \mathbf{C} * \mathbf{D},$$

where **A**, **B**, **C**, and **D** are all distributed vectors or matrices. Such expressions are enabled by the operator overloading feature of C++ [6]. A naive implementation of operator overloading in C++ results in the creation of temporary data structures for each

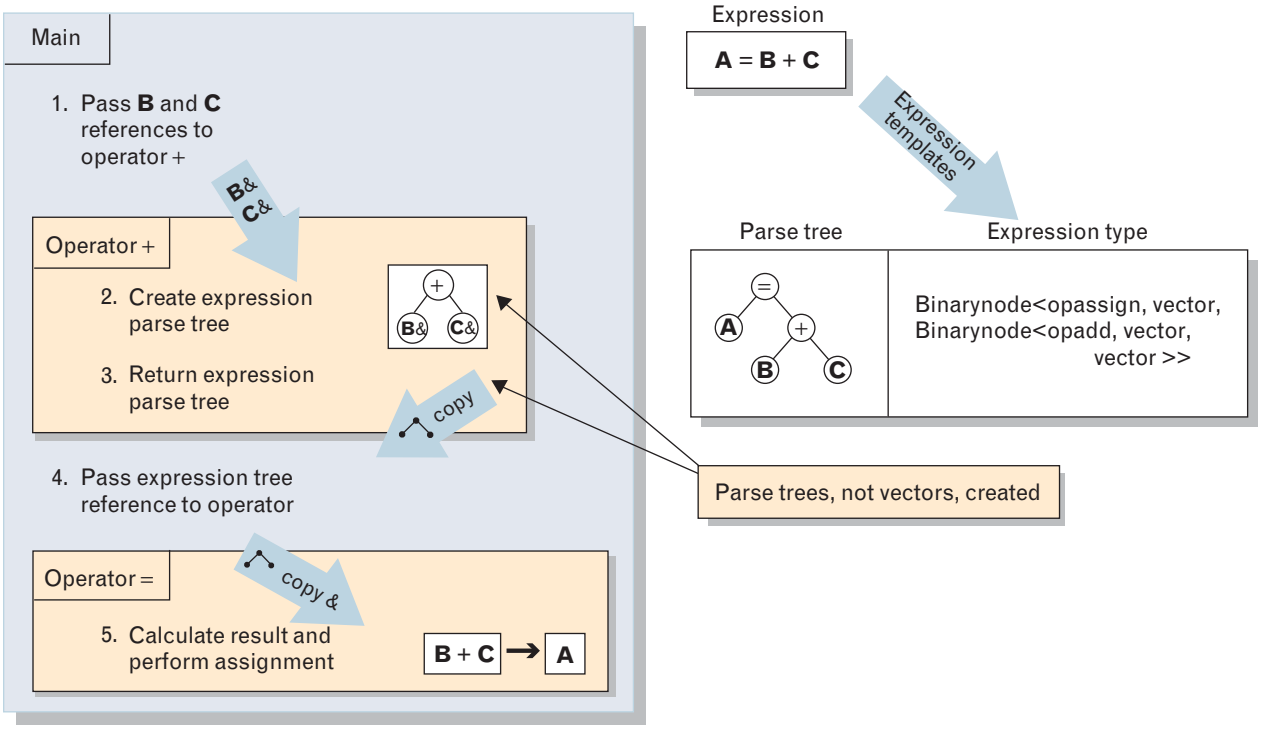


FIGURE 11. C++ expression templates. C++ typically requires the use of temporary variables in order to write high-level mathematical expressions. Obtaining high performance from C++ requires technology such as expression templates that eliminate the normal creation of temporary variables in an expression.

substep of the expression, such as the intermediate multiply **C** * **D**, which can result in a significant performance penalty. This penalty can be avoided by the use of expression templates, which allow the compiler to analyze a chained expression and eliminate the temporary variables. A tool called the Portable Expression Template Engine (PETE), developed by the Advanced Computing Laboratory at the Los Alamos National Laboratory, allows easy generation of expression template code for user-defined types. Figure 11 illustrates this process for the simpler expression **A** = **B** + **C**. In many instances it is possible to achieve better performance with expression templates than with standard C-based libraries because the C++ expression-template code can achieve superior cache performance for long expressions [7].

Figures 12, 13, and 14 show the performance PVL obtained with three different templated expressions on an eight-node processor cluster. For long expressions, PVL code that uses templated expressions is able to equal or exceed the performance of existing signal processing libraries (VSIPL and MPI).

Advanced Research and Future Work

This section describes some of the current research work being done to add new capabilities to PVL in the areas of fault tolerance, self-optimization, and dynamic parallelism.

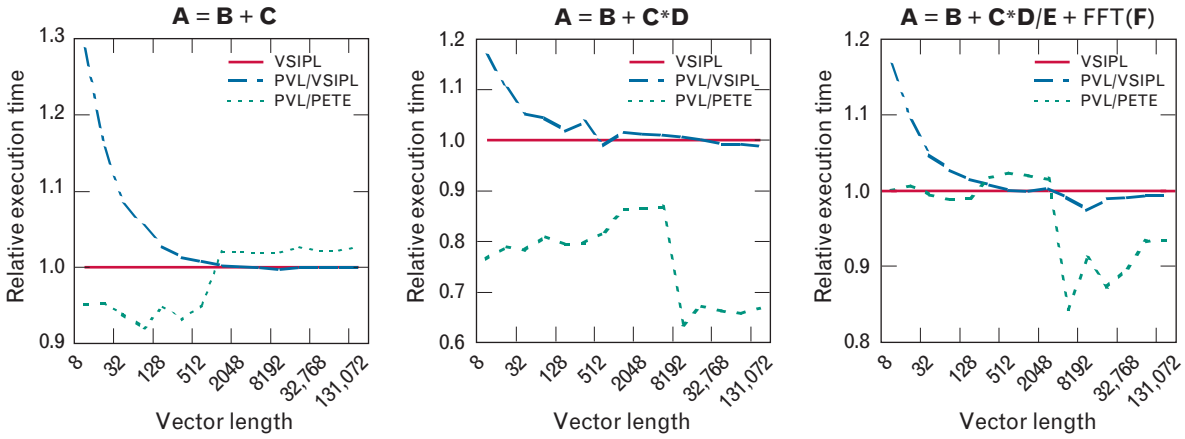


FIGURE 12. Comparison of single-processor performance of VSIPL (C), PVL (C++) on top of VSIPL (C), and PVL (C++) on top of the Portable Expression Template Engine (PETE) (C++) for three different expressions with different vector lengths. PVL with VSIPL or PETE is able to equal or improve upon the performance of VSIPL.

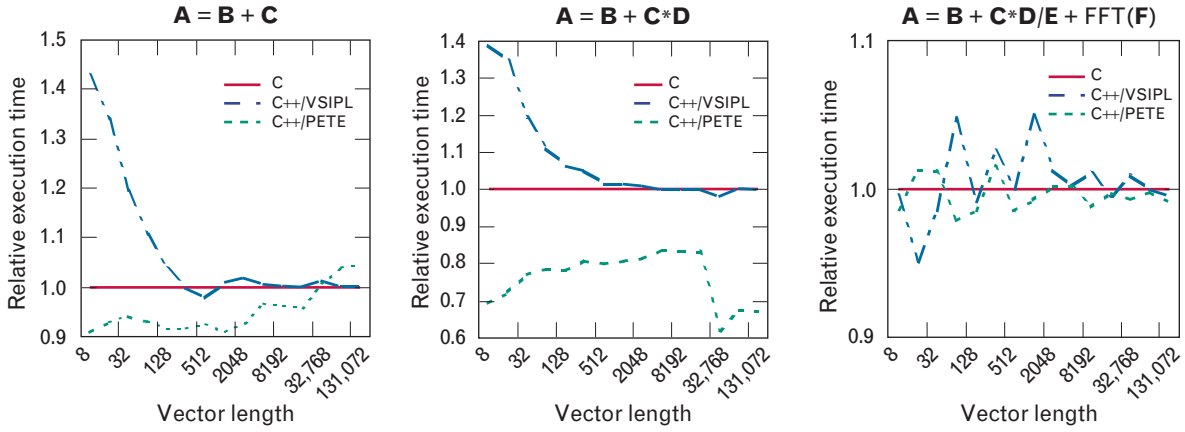


FIGURE 13. Comparison of multiprocessor (no communication) performance of VSIPL (C), PVL (C++) on top of VSIPL (C), and PVL (C++) on top of PETE (C++) for three different expressions with different vector lengths. PVL with VSIPL or PETE is comparable to or better than the performance of VSIPL.

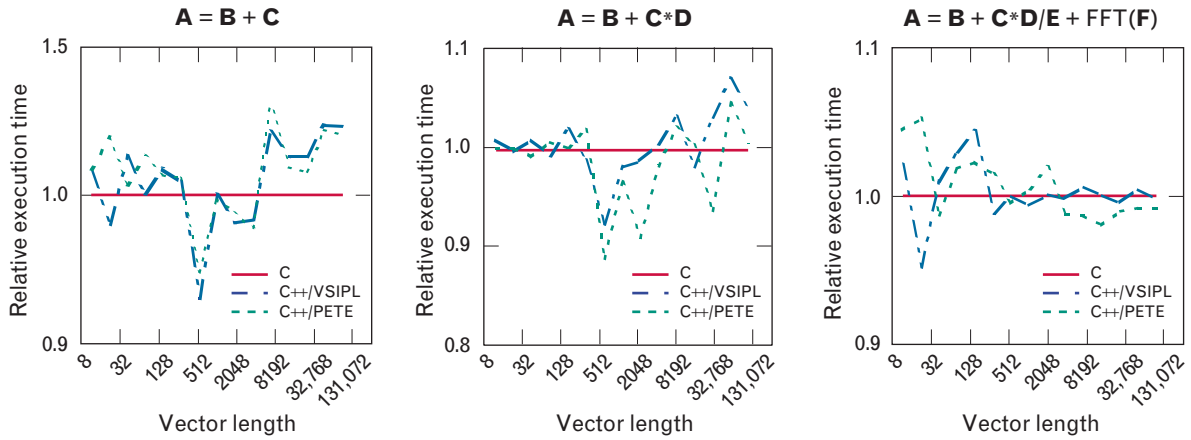


FIGURE 14. Comparison of multiprocessor (with communication) performance of VSIPL (C), PVL (C++) on top of VSIPL (C), and PVL (C++) on top of PETE (C++) for three different expressions with different vector lengths. PVL with VSIPL or PETE is comparable to the performance of VSIPL.

Fault Tolerance and Dynamic Mapping

One of the key challenges in bringing massively parallel computing to real-time signal processing is how to implement fault-tolerance capabilities in software. Signal processors need to be able to adjust quickly to processor failure so that they can continue to carry out their mission. The simplest approach is to double or triple the size of the computer and use a redundant voting scheme. Unfortunately, there are many situations in which this solution is too costly or the system is physically too large. In addition, this type of redundancy will not catch certain errors.

In theory, massively parallel systems present an opportunity for greatly increasing the fault tolerance of the system by adding only a few additional proces-

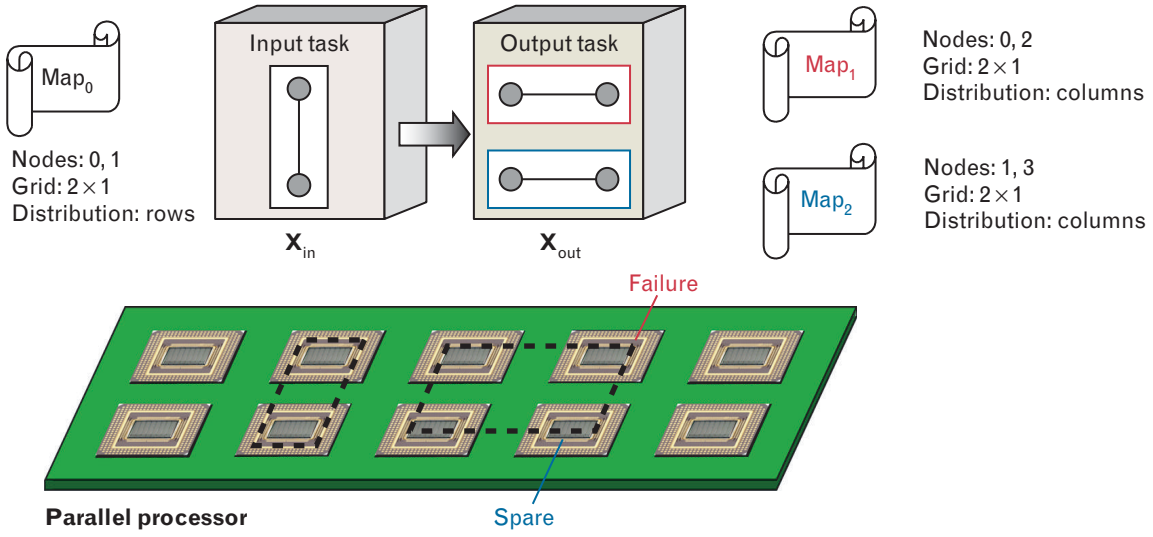


FIGURE 15. Dynamic mapping for fault tolerance. The separation of mapping from the algorithm allows the application to adapt more easily to changing hardware by modifying mapping at runtime. Remapping a matrix, for example, allows the program to react to failed processors and assign the work load to a spare processor, without requiring any fundamental change in the signal processing algorithm.

sors. Exploiting this capability requires software that can abstract the functionality of the program away from the hardware and allow processing tasks to be easily moved from one processor to another. In terms of the PVL library, the fault-recovery problem can be summarized as finding an efficient and dynamic way to give an object a new map after a failure has been detected, as illustrated in Figure 15.

We have implemented three fault-recovery strategies that solve this problem in different ways. In the first, referred to as *remapping*, the task object is destroyed and rebuilt from scratch on a new set of nodes. In the second strategy, referred to as *redundant objects*, a task object is built for each different possible failure scenario, and the appropriate object is used when a fault is detected. In the third strategy, the library allows a task to be mapped to a larger set of processors and only a subset is used at any given time. The set of nodes associated with the object can dynamically change to be a new subset of the larger set. This strategy is referred to as *rebinding*.

These three strategies have different effects on the complexity of the application (in terms of memory use and code size) and the performance of the application (in terms of samples processed per second). The remapping strategy is the least complicated of the three, but has the largest performance impact. The redundant object strategy has the best performance of the three, but is very complicated for the application programmer. The rebinding strategy puts the burden of complexity on the library developer and offers better performance than the remapping strategy. A more detailed description of our results in this area can be found elsewhere [8].

Self-Optimization

PVL allows algorithm concerns to be separated from mapping concerns. However, the issue of determining the mapping of an application has not changed. As a software application is moved from one piece of hardware to another, the optimal mapping needs to change, as illustrated in Figure 16. Currently, people

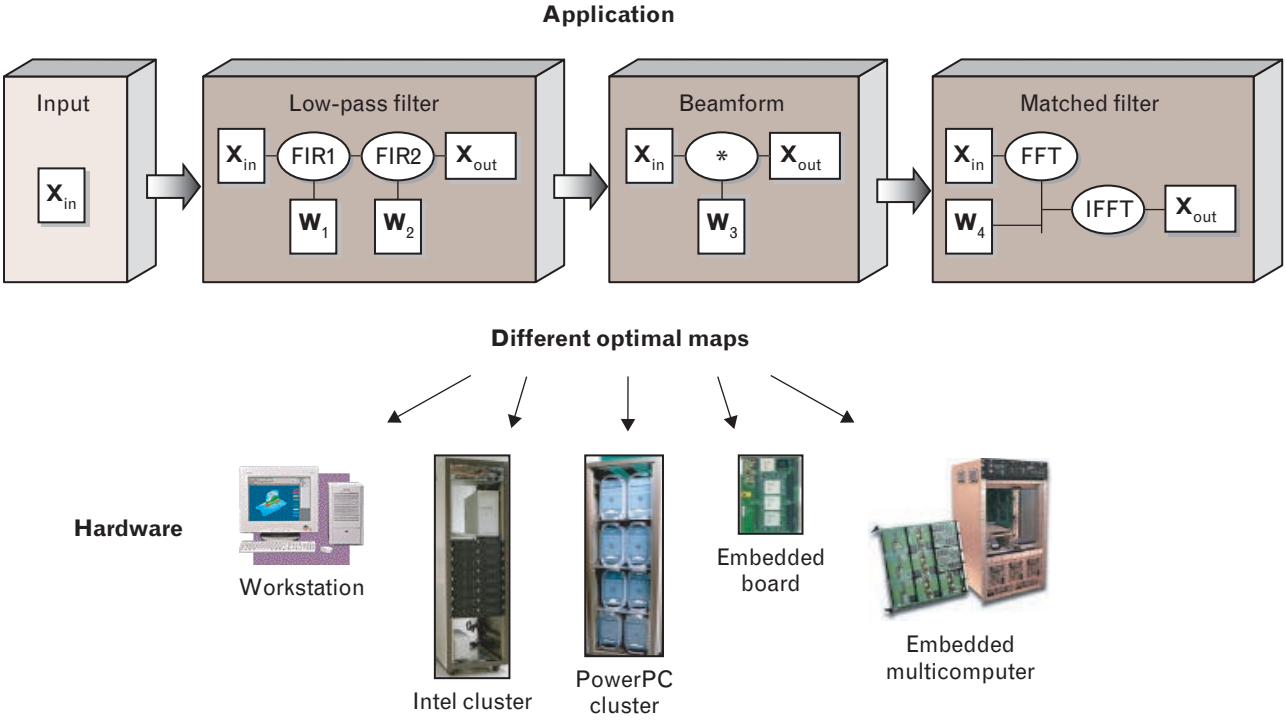


FIGURE 16. Optimal mapping challenge. Real applications can easily involve several complicated processing stages with potentially thousands of mapped vectors, matrices, and computations. The maps that produce the best performance will change, depending on the hardware. Automated capabilities are necessary to generate maps and to determine which maps are optimal.

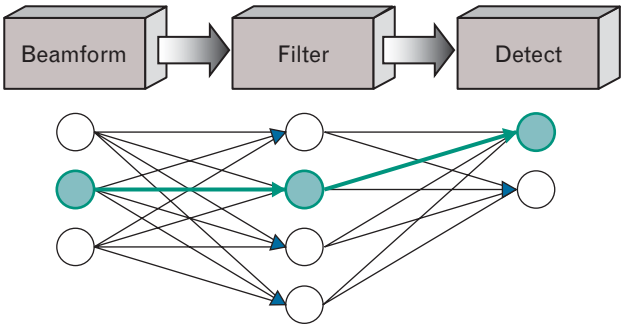


FIGURE 17. Self-optimizing Software for Signal Processing (S3P). This framework combines the dynamic mapping capabilities found in PVL with the self-optimizing software techniques developed in FFTW. The resulting framework allows an optimal mapping to be found automatically for any application on any parallel architecture. Each available path is a complete system mapping, and the best path (shown in green) is the optimal system mapping.

with knowledge of both the algorithm and the parallel computer use intuition and simple rules to try to estimate what the best mapping will be for a particular application on a particular hardware. Unfortunately, this method is time consuming, labor intensive, and inaccurate, and it does not in any way guarantee an optimal solution.

The generation of optimal mapping is an important problem because without this key piece of information it is not possible to write truly portable parallel software. To address this problem, we developed a framework called Self-optimizing Software for Signal Processing (S3P) [9].

The S3P framework, which is illustrated conceptually in Figure 17, requires certain capabilities from an application. First, the application must be made of

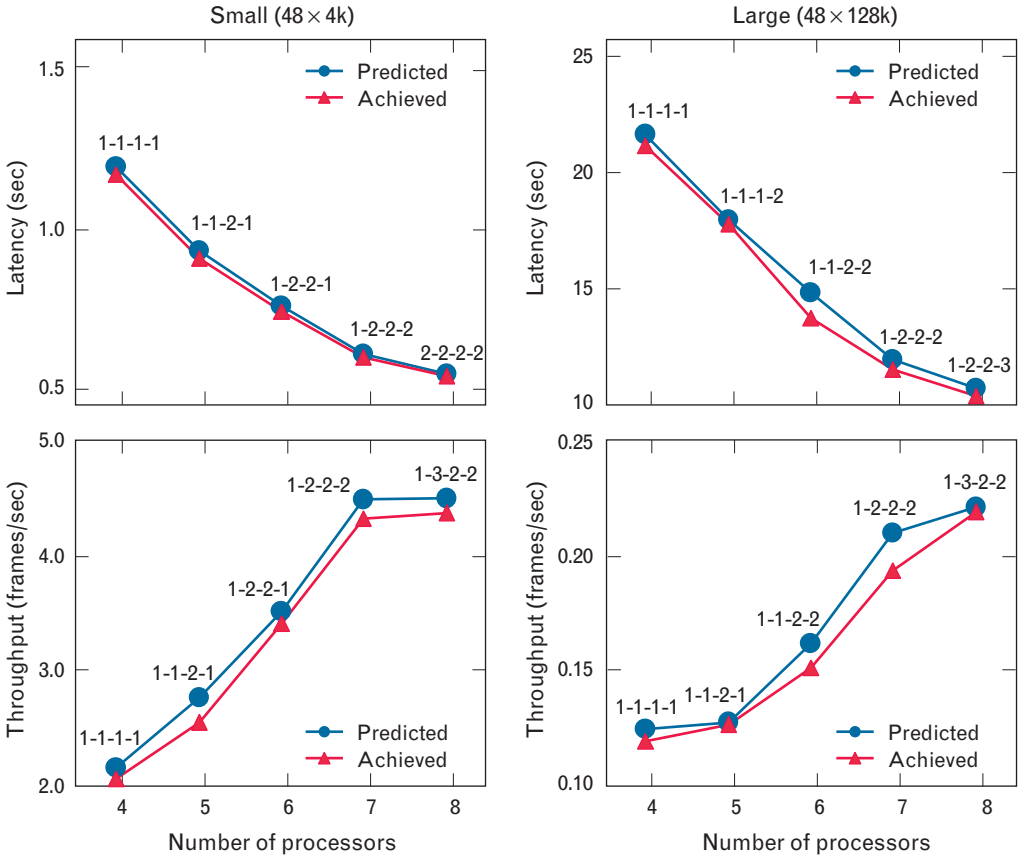


FIGURE 18. S3P performance. The S3P framework is tested on two different problem sizes in a four-stage application with two different criteria: minimize latency for a given number of processors and maximize the throughput for a given number of processors. In all cases S3P picks the correct mapping, as shown by the four-component processor assignment string, and predicts the performance of the best map to within a few percent.

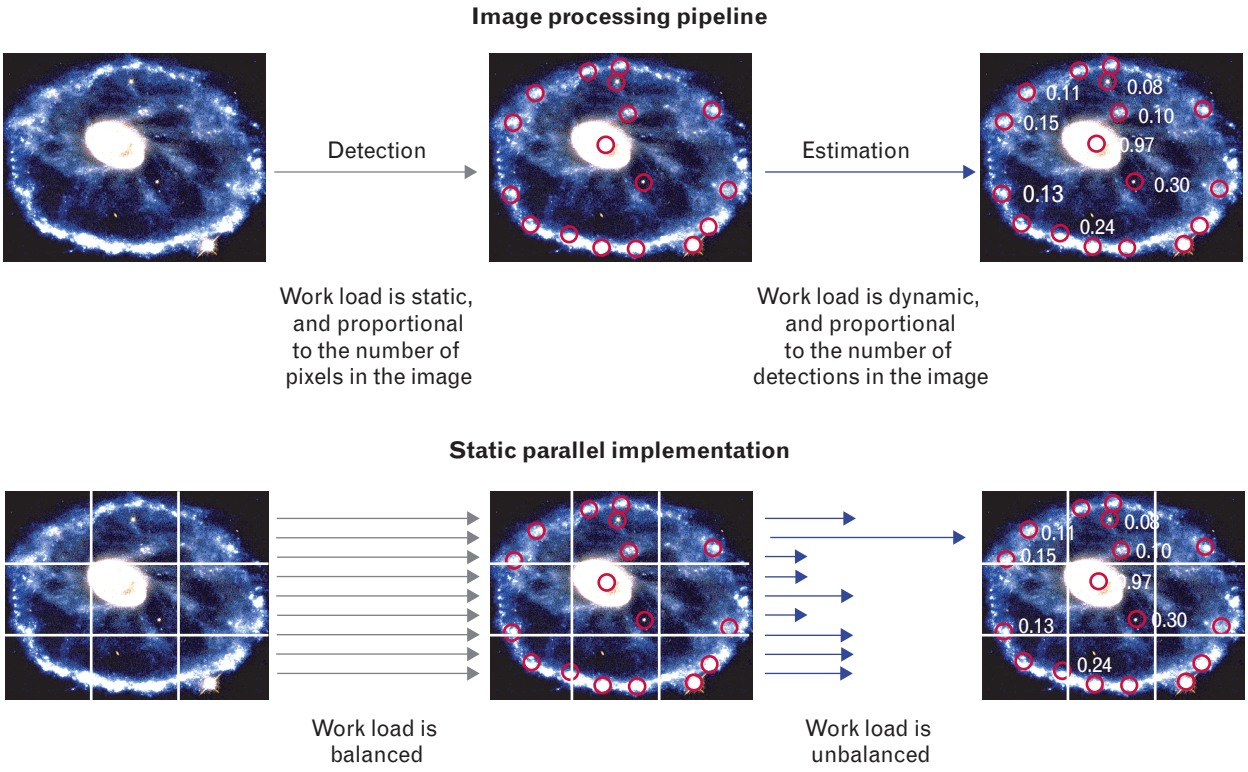


FIGURE 19. The problem of dynamic load balancing. A typical signal or image processing pipeline, shown in the upper row, begins by performing many regular operations such as detection that depend only upon the size of the input data, followed by additional operations such as estimation that depend upon the precise location of objects of interest within the data. The detection processing lends itself naturally to parallel computing by statically chopping up the data into equal regions, as shown in the lower row. This partition of the data results in a balanced work load among the parallel processors. The estimation processing, however, is dynamic in nature and a static distribution leads to unbalanced work loads on the parallel processors. Researchers are seeking solutions that dynamically distribute the work load among the processors on an as-needed basis.

tasks that are capable of being mapped to multiple configurations of hardware. Furthermore, each task needs to be able to measure its computing resources (e.g., number of processors, memory, execution time) in each configuration. Given these capabilities, S3P can assemble a system graph that contains all possible mappings and find the best path through this graph, which corresponds to the optimal mapping. Because the resource measurements are empirical there is no need for *ad hoc* modeling, and the entire process is automatic. This framework has been tested on different problem sizes and with different criteria, and it is able to pick the correct mapping and to predict the performance of the best map to within a few percent. Figure 18 shows S3P performance for two problem sizes and two performance criteria—latency and throughput. In each example, S3P finds the correct mapping and clearly predicts the achieved performance.

Dynamic Load Balancing

Another key area for continued development is the need to be able to address problems where the work load is dynamic. A typical example of this is post-detection processing in a signal processing system in which the amount of work done is proportional to the number of targets found. This situation is difficult to address with a parallel computer because it is not possible to predict how to distribute the data so that the workload is even.

The challenge presented by dynamic work loads is classically described by the "balls into bins" problem. Typically data in a signal processing system are dis-

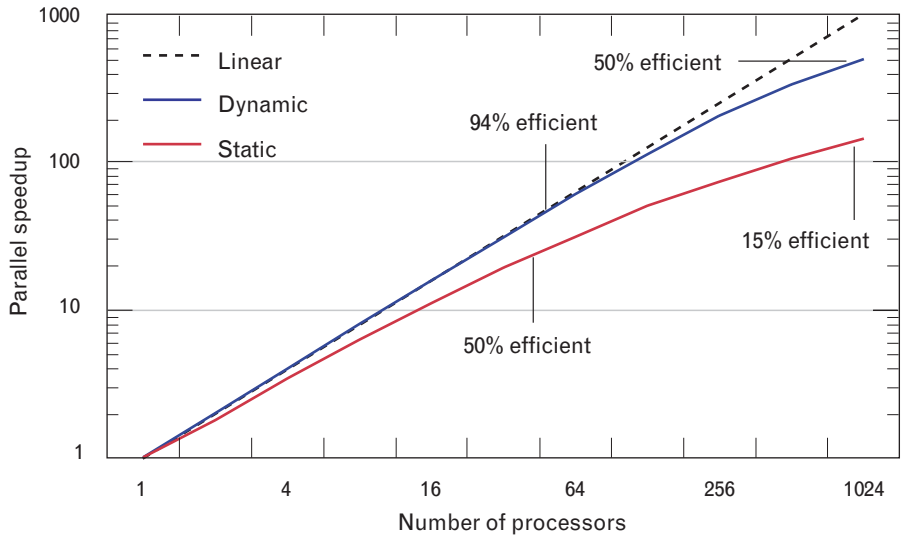


FIGURE 20. Static versus dynamic load balancing. Randomly distributed targets in data that are statically distributed on a parallel processor always result in an unbalanced processor distribution because of random fluctuations in the work load. This unbalance leads to a decrease in processing efficiency as the number of processors increases. If the work can be distributed dynamically among the processors on an as-needed basis, the efficiency can be much higher and is limited only by the granularity of the work.

tributed uniformly on a parallel processor, but targets or detections are randomly distributed in these data, which results in some processors having more work than others. The nonuniformity of the distribution of work can quickly lead to the vast majority of processors being idle while the application waits for one processor to finish working. Figure 19 illustrates this problem with an image processing pipeline, showing how a static parallel implementation results in an unbalanced work load. The solution to this problem lies in the ability to dynamically distribute the work on an as-needed basis. Figure 20 shows how a dynamic distribution can dramatically improve processing efficiency, compared to static distribution. This improvement becomes particularly significant as the number of processors increases. A variety of technologies are useful in making this type of solution feasible on a parallel computer, including shared memory, fast broadcast, and multithreading.

Summary

Exploiting parallel processing for real-time streaming applications presents unique software challenges. We have developed a software library to address many of these challenges. The library exploits advanced features of C++ to easily express data and task parallelism without making the application dependent on the underlying parallel hardware. This approach delivers high-performance execution comparable to or better than standard approaches.

Our future efforts will focus on adding to the features of this technology to exploit dynamic parallelism, integrate high-performance parallel software underneath mainstream programming environments, and use self-optimizing techniques to maintain performance.

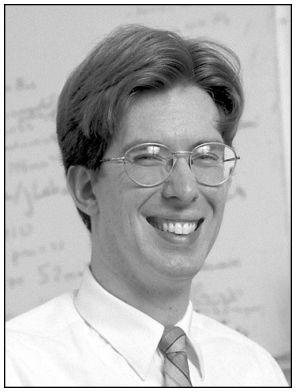
Acknowledgments

We would like to thank the entire PVL software development team, including Matt Anderson, Robert Bond, Hector Chan, Jim Daly, Mike DiMare, Ted Hall, Mike Harrison, Andy Heckerling, Paul Hergt, Hank Hoffmann, Michael Moore, Fran Rayson, Patrick Richardson, Ed Rutledge, and Glenn Schrader.

REFERENCES

1. "MPI: A Message-Passing Interface Standard," Message-Passing Interface Forum (University of Tennessee, Knoxville, Tenn., Apr. 1994), <http://www.mpi-forum.org>.
2. D.A. Schwartz, R.R. Judd, W.J. Harrod, and D.P. Manley, "Vector, Signal, and Image Processing Library (VSIPL) 1.0 Application Programmer's Interface," Technical Report (Georgia Tech Research Corporation, Atlanta, Ga., Mar. 2000), <http://www.vsipl.org>.
3. 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," *ISCOPE '97: First Conference on International Scientific Computing in Object-Oriented Parallel Environments*, *Marino del Rey, Calif., 8–11 Dec. 1997,* pp. 241–248.
4. "High-Performance Fortran Language Specification," High-Performance Fortran Forum, *Scientific Programming* **2** (1), 1993, pp. 1–170.
5. M. Frigo and S.G. Johnson, "An Adaptive Software Architecture for the FFT," *Proc 1998 IEEE International Conference on Acoustics, Speech, and Signal Processing* **3**, *Seattle, 12–15 May 1998,* pp. 1381–1384.
6. B. Stroustrup, *The C++ Programming Language*, 3rd ed. (Addison-Wesley, Reading, Mass., 1997).
7. S. Haney, J. Crotinger, S. Karmesin, and S. Smith, "Easy Expression Templates Using PETE, the Portable Expression Template Engine," *Dr. Dobb's J.* **24** (10), 1999.
8. J.M. Lebak, J.V. Kepner, and H. Hoffmann, "Software Fault Recovery for Real-Time Signal Processing on Massively Parallel Computers," *Proc. Tenth SIAM Conf. on Parallel Processing for Scientific Computing, Portsmouth, Va., 12–14 Mar. 2001.*
9. H. Hoffmann, J.V. Kepner, and R.A. Bond, "S3P: Automatic, Optimized Mapping of Signal Processing Applications to Parallel Architectures," *Proc. Fifth Annual High-Performance Embedded Computing (HPEC) Workshop*, *Nov. 2001.*

JEREMY KEPNER
is a staff member in the Embedded Digital Systems group. His research has addressed the development of parallel algorithms and tools, and the application of massively parallel computing to a variety of data-intensive problems. He received a B.A. degree in astrophysics from Pomona College, and a Ph.D. degree in astrophysics from Princeton University in 1998, after which he joined Lincoln Laboratory.



JAMES LEBAK
is a staff member in the Embedded Digital Systems group. He is interested in parallel and numerical algorithms for digital signal processing. He received B.S. degrees in mathematics and electrical engineering in 1989, and an M.S. degree in electrical engineering in 1991, from Kansas State University. He received a Ph.D. degree in electrical engineering from Cornell University in 1997.