An Open Standard Software Library for High-Performance Parallel Signal Processing: the Parallel VSIPL++ Library

James Lebak, Jeremy Kepner, Henry Hoffmann, and Edward Rutledge
MIT Lincoln Laboratory, Lexington, MA
e-mail: {jlebak,kepner,hoffmann,rutledge}@ll.mit.edu

Abstract

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 hardware but software. Specifically, the key technical hurdle lies in allowing the user to write programs at high level, while still achieving performance and preserving the portability of the code across parallel computing hardware platforms. The Parallel Vector, Signal, and Image Processing Library (Parallel VSIPL++) addresses this hurdle by providing high level C++ array constructs, a simple mechanism for mapping data and functions onto parallel hardware, and a community-defined portable interface. This paper presents an overview of the Parallel VSIPL++ standard as well as a deeper description of the technical foundations and expected performance of the library. Parallel VSIPL++ supports adaptive optimization at many levels. The C++ arrays are designed to support automatic hardware specialization by the compiler. The computation objects (e.g. Fast Fourier Transforms) are built with explicit setup and run stages to allow for run-time optimization. Parallel arrays and functions in VSIPL++ support these same features, which are used to accelerate communication operations. The parallel mapping mechanism provides an external interface that allows optimal mappings to be generated off-line and read into the system at run-time. Finally, the standard has been developed in collaboration with high performance embedded computing vendors and is compatible with their proprietary approaches to achieving performance.

 This work is sponsored by the High Performance Computing Modernization Office under Air Force Contract F19628-00-C-0002. Opinions, interpretations, conclusions, and recommendations are those of the authors and are not necessarily endorsed by the United States Government.

I. INTRODUCTION

Real-time signal processing is critical to a wide variety of applications: examples include 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 only be met through the use of multiple processors and fast, low-latency networks. The need for a balance between computation and communication is one of the key characteristics of this type of processing. Advanced software techniques are required 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 (see Figure 1). 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.

The usual 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 high performance. Unfortunately, this approach usually leads to software that is difficult to write, does not run on other platforms, and is dependent on the number of processors on which the application is deployed. Performance has been achieved, but at a high cost in programmer effort and software portability. This approach has led to a situation where, in current programs, the cost of software development outweighs the cost of hardware development by a significant and widening gap. These rising costs are driven by the lack of portability in current code and the effort required to achieve performance. Thus, there is a need to increase the productivity of high-performance embedded system software development, allowing performance to be achieved at a lower cost.

The High-Performance Embedded Computing Software Initiative (HPEC-SI) is designed to improve the state of affairs in developing high-performance software for embedded signal processing platforms. The goals of the effort are:

- to increase productivity of distributed signal and image processing application development;
- to make application level code more portable; and

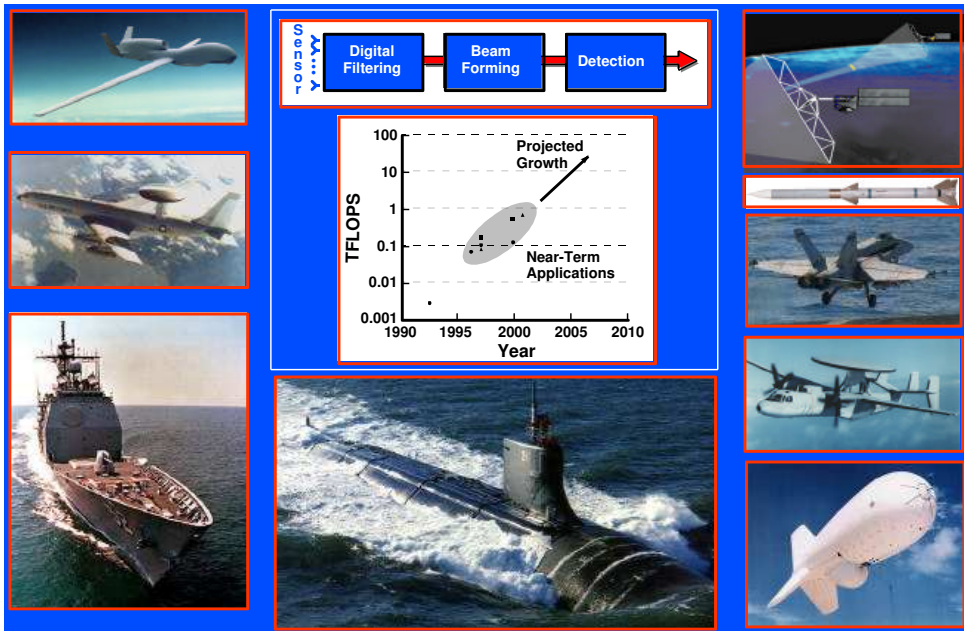

Fig. 1. **Military Processing Requirements.** Military platforms are among the largest drivers of embedded signal processing. By 2010, DoD systems are projected to require tens of teraflops of computation.

- to make it easier to achieve high performance on many different parallel platforms.

The primary mechanism for achieving these goals is the development of a new open standard for embedded signal processing, the Parallel VSIPL++ library. This new library builds on the functionality defined in the existing open standard Vector, Signal, and Image Processing Library (VSIPL, [1], [2]). It adds to this the concept of *processor map independence* taken from the MIT Lincoln Lab Space-Time Adaptive Processing Library (STAPL, [3]) and Parallel Vector Library (PVL, [4]), which orthogonalizes the tasks of writing an application and mapping it onto a parallel processor. Finally, it uses the object-oriented capabilities of C++ to achieve high performance while maintaining a high level of abstraction, similar to previous work done at Los Alamos National Laboratory [5], [6].

We begin by describing the application domain, target architectures, and related work in more detail. We move on to describing functionality included in PVL, which serves as a basis for the functionality of Parallel VSIPL++. We close by describing experiments in self-optimization for parallel programs.

II. Application Domain

The application domain for Parallel VSIPL++ is embedded real-time signal processing. By embedded, we mean that the applications are closely coupled to the platform in some way. In this section, we

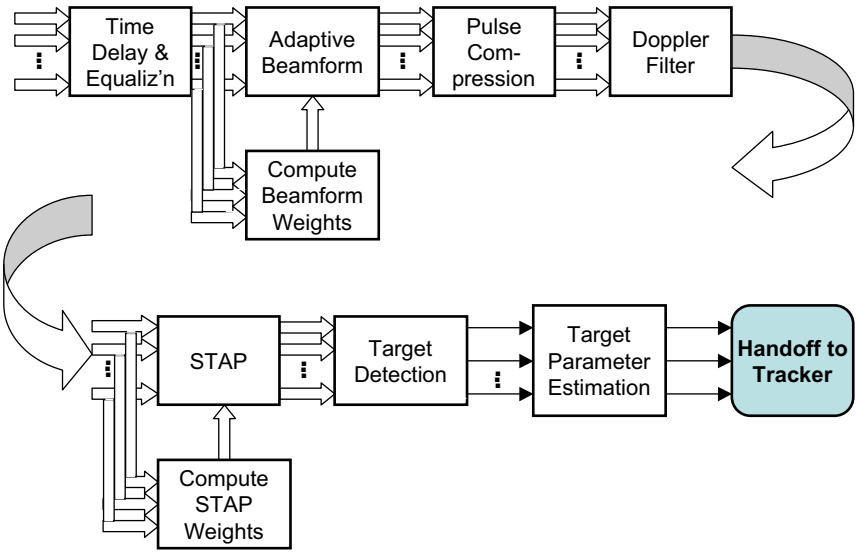

Fig. 2. Example radar signal processing chain for a ground moving-target indicator (GMTI) radar, taken from Reuther [7].

provide examples of the applications and platforms that are involved. We derive characteristics of the Parallel VSIPL++ library from the domain, describe the projects that are the antecedents of the library, and survey related work.

A. Application Examples

As an example of the embedded real-time signal processing systems that Parallel VSIPL++ is targeting, consider the signal processing performed by a ground moving-target indicator (GMTI) radar. This type of radar system is used to find moving targets on the ground. The signal flow for a typical GMTI radar is shown in Figure 2. This particular flow is described in more detail in a technical report by A. Reuther [7]. However, without dwelling excessively on the details of the computations performed, the processing in each of the seven stages is summarized below (see [7, pp.2–3]).

1) The *time delay and equalization stage* compensates for the differences in the transfer function between channel sensors.
2) The *adaptive beamforming stage* transforms the filtered data to allow detection of target signals coming from a particular set of directions of interest while filtering out spatially-localized interference.
3) The *pulse compression stage* filters the data to concentrate the signal energy of a relatively long transmitted radar pulse into a relatively short pulse response.

4) The *Doppler filter stage* processes the data so that the radial velocity of targets relative to the platform can be determined.
5) The *space-time adaptive processing or STAP stage* is a second beamforming stage which removes further interference and ground clutter interference.
6) The *detection stage* compares a radar signal response to its surrounding signal responses to determine whether a target is present.
7) The *estimation stage* estimates target positions in order to pass them to the tracking algorithms.

In terms of computation requirements the first five stages are the most important. Stages 1 and 3 involve several hundred applications of a set of finite impulse-response (FIR) filters. Stages 2 and 5 involve 10-100 solutions of an overdetermined linear system via least-squares, and a corresponding number of matrix-matrix multiplications. Stage 4 involves several hundred fast Fourier transformations (FFTs). All of these computations must be performed rapidly, because a new data set is generated by the sensor at a rate on the order of 10's of milliseconds. In 2004, the overall throughput requirement for the processing stream shown would be on the order of hundreds of gigaflops per second.

A second example application is provided by the standard missile (SM) program, whose signal processing chain was described by Rabinkin and Rutledge [8]. The signal processing chain is reproduced in Figure 3. This is an image processing application, where the operations performed on each pixel require a certain number of near-neighbor pixels. The part of the chain marked "front-end video processing" amounted to about 300 Mflop/s in 2002, with advanced processing predicted to scale to about 7 Gflop/s. While this processing load is not great by the standards of commercial workstations, Figure 3 has to be implemented on a computer that can fit on a single board in the 34.3 cm diameter provided by the missile. Again, a brief description of the front-end video processing is given here, summarized from Rabinkin *et al.* [9].

1) The *adaptive non-uniformity compensation stage* corrects for the non-uniform nature of the focal plane's pixel response.
2) The *dither processing stage* stabilizes the image by subtracting the effects of platform motion.
3) The *multi-frame processing stage* integrates multiple corrected frames to improve response from target objects.
4) The *constant false-alarm rate (CFAR) detection stage* performs local thresholding to detect target objects.
5) The *sub-pixel processing stage* improves estimation of target position.

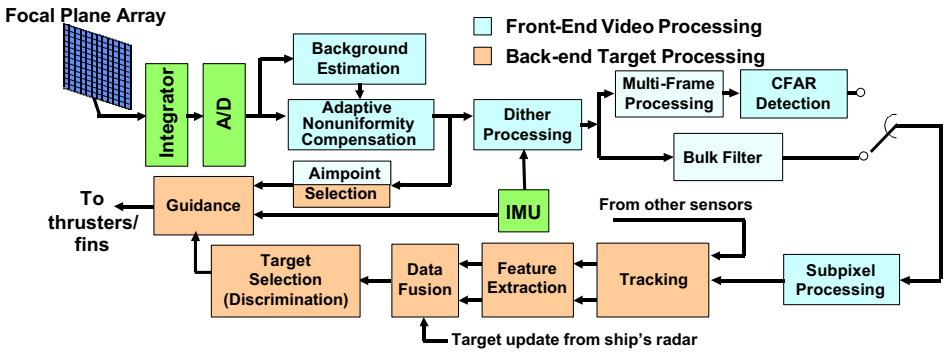

Fig. 3. Example processing flow for the Navy's standard missile (SM), taken from Rabinkin *et al.* [9].

These applications exhibit multiple opportunities to exploit parallelism. First, stages may be executed in a pipelined fashion, allowing the processing of multiple data sets to be overlapped. Second, the multiple problems in each stage may be computed by different processors. Finally, data-parallel processing may be performed: significant linear algebra operations, such as the least-squares solution and matrix multiplication in the GMTI application, or the image processing operations in the standard missile application, may be computed by multiple processors. The Parallel VSIPL++ library is required to provide support for all of these opportunities for parallelism.

Unlike scientific computing programs that may run for hours to compute a solution to a single problem, programs in this domain typically process a continuous stream of data sets that arrive at very short time intervals. The sequence of operations is generally known long before execution, and the platform is generally dedicated to the application. Therefore, overall system execution time can be reduced by setting up for particular operations ahead of time. A well-known example of such setup is the pre-computation of weighting factors for a particular size of FFT (see Van Loan [10] for a discussion of techniques for computing the FFT).

A further characteristic of these applications is that operations are performed across different dimensions of the data set. In these cases, the primary challenge is not performing the computations but moving data so that it is in a position to be acted upon. One of the most common examples of this 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 (radar, sonar, communications, etc.): typically, processing within a particular channel is followed by processing that cuts across channels. For example, in the GMTI application (Figure 2), the data set can be viewed as a three-dimensional entity whose dimensions correspond to sensor channels, range samples, and pulses transmitted. The

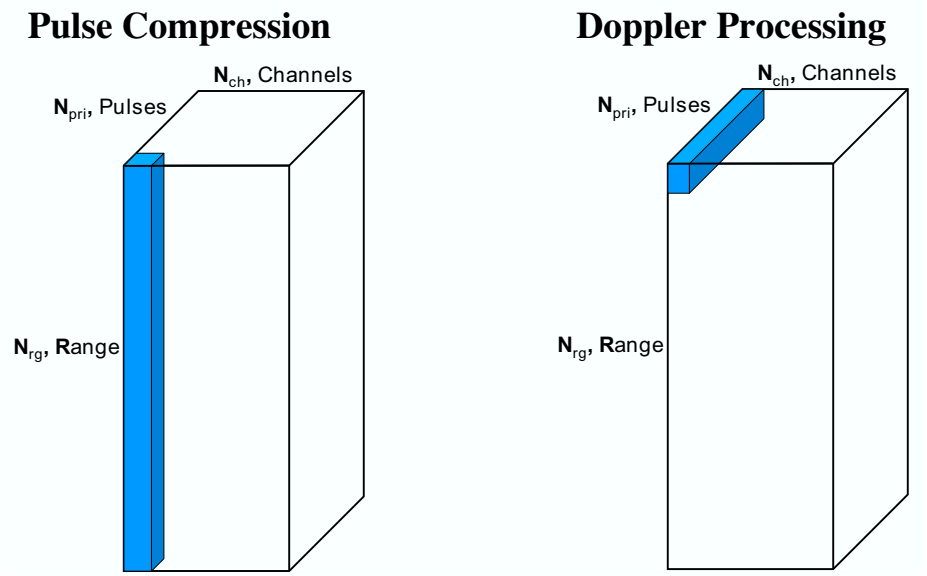

Fig. 4. **Corner Turn.** The pulse compression stage operates on all the range data for each pulse and channel; the Doppler filtering stage operates on all the pulse data for each range cell and channel. This may necessitate a data re-arrangement or corner turn between these two stages.

pulse compression stage (stage 3) operates on a set of range samples corresponding to a particular pulse and channel. The following stage, the Doppler processing stage (stage 4), operates on all the pulses for each range and channel (Figure 4). 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 is merely moved in memory. In a *distributed* corner turn, all-to-all communication between processor groups is required in addition to local data movement.

Distributed corner turns are a primary bandwidth driver for embedded multicomputers. Fortunately, the continuous data processing requirement of these applications allows the computation to be pipelined so that the communication latency of the corner turn operation can be hidden. Hiding communication latency is one of the primary reasons pipelines are used in these applications.

B. Target Architectures

The major target hardware class for the application domain mentioned here is the embedded multicomputer. Included in this class are single-board shared-memory processors that can support the processing shown in Figure 3. The particular board used by Rabinkin *et al.* for their application is made by DY4 and includes 4 PowerPC G4 processors which share memory among them [8]. Also included in the target class are the hundreds-of-processor distributed-memory multicomputers required to support applications

such as that shown in Figure 2. The key characteristic that machines in this class exhibit is that they are chosen to fit the *form factor* constraints of the intended platform, that is, to maximize the available floating-point operations per unit size, weight, and/or power.

Constrained in size, weight, and power, systems in this domain sometimes require the use of digital signal processors, such as the Analog Devices TigerSHARC. For examples of circumstances under which such devices might be reasonable choices, see Mirod [11]. Such processors typically include on-chip memory that must be managed effectively in order to obtain high performance. Parallel VSIPL++ must provide mechanisms to allow the proper memory management.

Parallel application development and testing in this domain is often performed on low-cost commodity cluster hardware. These clusters may have a different number of nodes than the eventual target. Hence, Parallel VSIPL++ must support testing on workstations and clusters, and deployment on embedded multicomputers.

C. Evolution of Standard Signal Processing Interfaces

The ten-year period of 1995-2004 has seen a substantial change in the software provided by embedded multicomputer manufacturers. In 1995, major multicomputer vendors, including CSPI, Mercury, and Sky, all provided different math and communication libraries that were essentially incompatible with one another. This meant that applications written for one platform were not portable to other platforms. By contrast, in 2004, embedded multicomputer manufacturers typically provided an implementation of the vector, signal, and image processing standard library (VSIPL) for computation. This is true for the vendors mentioned above, as well as others listed at the VSIPL Forum web site: `<http://www.vsipl.org>`.

The VSIPL standard grew out of a desire to have a portable interface to vector and signal processing functions. The VSIPL forum, including representatives from multicomputer vendors, application developers (for example, Lockheed-Martin and Raytheon), and research laboratories (for example, Georgia Tech Research Institute and the Air Force Research Laboratory), defined and maintains the standard. The VSIPL standard defines ANSI C function prototypes and requirements for more than 800 functions, ranging from simple vector addition to signal processing (for example, FFT, filtering, and convolution) and complicated linear algebra operations (for example, QR factorization and singular value decomposition).[1] VSIPL is an *object-based* library: users specify computations as operations on vector and matrix objects.

Application builder experiences have shown that VSIPL provides portability without sacrificing performance [12], [13]. Three major characteristics of VSIPL, listed below, are designed to allow performance

[1]VSIPL provides limited image processing functions. Additional functions are a proposed extension to the standard.

on embedded multicomputers.

- *Early binding*, meaning that a complex computation is separated into a "setup" phase and a "compute" phase, where it is assumed that the compute phase is executed many more times than the setup phase. During the setup phase, an object associated with the operation is created that may contain memory areas required for the operation and pre-computed constants, such as the weights associated with an FFT. We refer to such objects as *computation objects*.
- Separation of storage and mathematical objects, to allow application writers to reference data multiple times or in multiple ways without performing copies. In VSIPL, data storage is represented by a *block*, an abstraction of a contiguous memory area. Users perform operations using *views*, which allow data in a block to be operated on as if it were a vector, matrix, or three-dimensional tensor. Multiple views may reference the same block and order the data differently.
- Library ownership of data, to allow vendor memory optimizations. Before VSIPL will compute on the data in a block, the user must call the *admit* function. After admit has been called, the user is not allowed to alter the data outside of VSIPL (for example, using a pointer to the data) before calling the *release* function. This allows the vendor freedom to optimize for embedded processors, such as the Analog Devices SHARC series, that include fast on-chip memory that must be explicitly managed.

These three items are the primary way in which the VSIPL standard enables platform adaptation. Early binding allows the application programmer to tell the library to speed up certain operations. Separation of storage and mathematical objects allows the application writer to reduce the number of copies in an application. Library ownership of data allow the library writer to optimize for his platform.

D. Multi-Processor Systems

Though VSIPL does not specifically address multi-processor systems, it can be used in such systems, in combination with communication mechanisms. The message-passing interface (MPI, [14]) is the *de facto* standard for communicating among processors in most distributed architectures, including cluster hardware. However, some embedded multi-processor vendors also provide additional communication libraries with different functions and performance characteristics: it may be desirable to use these libraries in certain circumstances. Furthermore, single-board embedded computers may not provide MPI, but instead use a custom library to perform direct memory access (DMA) transfers of data among processors. The DY4 board used in the Standard Missile program is an example [8].

Programming a multiprocessor system using a combination of VSIPL and a communication library has

```
/* Create a VSIPL block with a public data space */
vsip_scalar_d *buffer = (vsip_scalar_d*) malloc(L*sizeof(vsip_scalar_d));
vsip_block_d *block = vsip_blockbind_d(buffer, L VSIP_MEM_NONE);

/* Admit the block to VSIPL to operate on it
vsip_blockadmit_d(block, VSIP_FALSE);

/* Bind the public data to a vector view: offset=0, stride = 1, length=L */
vsip_vview_d *data=vsip_vbind_d(block, 0, 1, L);

/* VSIPL can compute with the view here */

/* Release the block to send data */
vsip_blockrelease_d(block, VSIP_TRUE);

/* Send the vector */
MPI_Send(buffer, L, MPI_DOUBLE, dest, tag, comm);

vsip_blockadmit_d(block, VSIP_TRUE);
vsip_vdestroy_d(data);
free(buffer);
```

Fig. 5. Example of interfacing VSIPL and MPI. Adapted from Skjellum and Bangalore [15].

two drawbacks. The first stems from VSIPL's requirement for ownership of memory. This requirement often means that a user must explicitly allocate and manage separate data structures for the computation and communication libraries. VSIPL's `admit` and `release` functions must be called to inform VSIPL of when it can and cannot compute on the data. This process is at best tedious and at worst can be a source of programming errors. The required protocol is illustrated by the code example in Figure 5 (adapted from Skjellum and Bangalore [15]).

A second, and more serious, drawback of programming a multicomputer using VSIPL and a communication library is that it is hard to make the application support modifications to the number of hardware elements used. Such modifications may be required, for example, when moving the program from a single-processor prototyping environment to a full-scale deployed parallel system, or when moving an application from one generation of technology to another. Even if the communication library used is a portable interface such as MPI, the programmer must expend additional effort to ensure that application code can scale to support such scenarios.

Portability between parallel software platforms requires more than just a portable interface: it requires a new concept, that of *processor map independence*, which allows the application code to become independent of its layout on a parallel processor. Map independence allows an application's functionality to be independent of the optimizations made to deploy it on a new parallel platform. It also allows a program's functionality to be tested before scaling, which provides obvious advantages for decreasing the time required to test and deploy a system. Map independence has been explored by MIT Lincoln

Laboratory in previous parallel library developments, referred to as the parallel vector library (PVL, [4]) and the space-time adaptive processing library (STAPL, [3]). STAPL was used to field a 1000-CPU embedded signal processor. PVL incorporated the lessons learned from STAPL into an object-oriented library written in C++. Among other applications, PVL has been used to achieve the 60 frames per second real-time throughput requirement for the standard missile program, mapping the application described in Figure 3 to the DY4 board described in Section II-B [8].

E. Extending VSIPL to Multi-Processors: Parallel VSIPL++

The HPEC-SI effort is defining Parallel VSIPL++, a standard library for embedded parallel signal processing. It is doing so in two stages. First, it is defining VSIPL++, a object-oriented C++ API with the functionality of VSIPL. Second, it is defining Parallel VSIPL++, which extends the VSIPL++ API to allow data-parallelism and task-parallelism. This two-stage approach allows a clear migration path from VSIPL to VSIPL++ to Parallel VSIPL++. A code example for each of the three libraries is provided in Figure 6. VSIPL++ code and Parallel VSIPL++ code are shorter than VSIPL code because of the advantages of C++, such as default arguments and namespaces. Notice that the only difference between the VSIPL++ and Parallel VSIPL++ code is the addition of a map argument to the constructors for the matrices and vectors.

In Parallel VSIPL++, as in the STAPL and PVL prototype libraries, map independence is achieved by raising the level of abstraction at which parallel signal processing applications are written. The library provides operations on *high-level* distributed objects, useful for both computation and communication, and handles the coordination of these operations without explicit direction from the application programmer. This use of abstraction achieves map independence for the application. It also has the added benefit of making the job of writing distributed signal processing application software easier, eliminating the coordination between computation and communication libraries alluded to in the previous section.

Care must be taken, when developing software at a high level of abstraction, to ensure that the use of abstraction does not create additional overhead. For example, an operation involving multiple high-level objects may, in a simple implementation, result in the use of temporary storage and require extra copy operations. Practical high-performance libraries in C++ today use a technique referred to as *expression templates* to achieve high performance while taking advantage of abstraction [17]. Expression templates allow operations involving multiple high-level objects to be combined at a low level by the compiler, eliminating the need for temporary storage and copies. An example of this is provided by the POOMA library developed at Los Alamos National Laboratory [5], [6].

Single-processor C VSIPL

```
/* Setup phase */
vsip_vview_f* w = vsip_vcreate_f(M, VSIP_MEM_NONE);
vsip_mview_f* A = vsip_mcreate_f(M, N,
                                 VSIP_ROW,
                                 VSIP_MEM_NONE);
vsip_qr_f qrdObject = vsip_qrd_create_f(M, N,
                               VSIP_QRD_SAVEQ);
/* end of setup phase */
/* Generate or read A & w here */
/* Compute phase */
vsip_qrd_f(qrdObject, A);
vsip_qrd_prodq_f(qrdObject,
                 VSIP_MAT_HERM,
                 VSIP_MAT_LSIDE,
                 w);
/* end of compute phase */
/* calls to free objects not shown */
```

(a)

Single-processor VSIPL++

```
/* Setup phase */
Vector<> w(M);
Matrix<> A(M, N);
qrd qrdObject(M, N,
              QRD_SAVEQ);
/* end of setup phase */
/* Generate or read A & w here */
/* Compute phase */
qrdObject.decompose(A);
qrdObject.prodq<VSIP_MAT_HERM,
                VSIP_MAT_LSIDE>(w);
/* end of compute phase */
```

(b)

Parallel VSIPL++

```
/* Setup phase */
Vector<> w(M, wMap);
Matrix<> A(M, N, aMap);
qrd qrdObject(M, N,
              QRD_SAVEQ);
/* end of setup phase */
/* Generate or read A & w here */
/* Compute phase */
qrdObject.decompose(A);
qrdObject.prodq<VSIP_MAT_HERM,
                VSIP_MAT_LSIDE>(w);
/* end of compute phase */
```

(c)

Fig. 6. Sample VSIPL (a), VSIPL++ (b), and parallel VSIPL++ (c) code to factor a $M \times N$ matrix A into orthogonal factor Q and triangular factor R (see [16]), and compute the product $Q^H w$. In all three examples, a compute object called `qrdObject` is created to provide storage for the operation. In (c), the names `wMap` and `aMap` refer to maps defined in an outside file.

At the time of this writing, the VSIPL++ and Parallel VSIPL++ standards are under development by the HPEC-SI standards activity. CodeSourcery, LLC, is the primary developer of the reference implementation, which is in testing by early adopters. In order to provide a more concrete description of the desired functionality, we describe the implementation and results from the MIT Lincoln Lab Parallel Vector Library (PVL) in Section III. PVL is used in several deployed applications (for example, [8]), and is a prototype for the functionality included in the Parallel VSIPL++ reference implementation. Table I summarizes the characteristics of Parallel VSIPL++ and its antecedents.

Library Name	Signal Processing	Community-defined Standard	Object-Oriented	Parallel	Year
VSIPL	X	X			1999
POOMA			X	X	1999
PVL	X		X	X	2000
VSIPL++	X	X	X		2004
Parallel VSIPL++	X	X	X	X	2005

TABLE I

CHARACTERISTICS OF DIFFERENT LIBRARIES RELATED TO PARALLEL VSIPL++.

F. Related Work

The HPEC-SI effort is designed to make parallel processing more widely available to signal processing system developers by building on the success of VSIPL and on ideas that have been in the literature for a long time. Parallel VSIPL++ provides a portable, multi-vendor framework for separating the expression of application functionality from the specification of parallelism. It uses the expression template approach to allow the library to build efficient code by combining operations that would otherwise be separate function calls.

VSIPL++ and Parallel VSIPL++ use expression templates to produce optimized implementations of mathematical expressions involving high-level objects. Other techniques in the literature perform more detailed optimization in particular domains. The approach used by Morrow *et al.* is similar to the expression template approach: it uses a *delayed execution* model, which is similar to the expression template approach, to construct programs that it passes to an off-line optimizer. The optimizer in turn builds implementations to run on a "parallel co-processor" [18]. In another vein, the SPIRAL approach uses a learning technique to automatically construct fast implementations of signal processing transform routines adapted to a particular platform [19], [20]. FFTW [21] and ATLAS [22] perform similar optimizations for FFTs and for specific linear algebra routines. As VSIPL++ and Parallel VSIPL++ are API specifications, implementations are free to make use of these technologies to optimize for specific platforms.

The parallel-Horus library developed by Seinstra *et al.* is very similar to the Parallel VSIPL++ effort. There, the emphasis is primarily on applying parallelism to image processing research, while Parallel VSIPL++ is directed primarily on signal processing system implementation. Nonetheless, their list of requirements (paraphrased from [23]) is very similar to the Parallel VSIPL++ goals:

1) Extend an existing, extensive, sequential API, which should disclose as little as possible about the library's parallel processing capabilities.
2) Obtain high efficiency, both on single operations and aggregate operations.
3) Design the implementation to avoid unnecessary code redundancy and enhance reusability.
4) The user must be able to add types and operations.
5) The implementation must run on Beowulf clusters.
6) The implementation must be portable to different architectures, achieved through implementation in high-level languages like C or C++.

Requirement 1 is nearly identical to the concept of map independence. The other requirements are also objectives for Parallel VSIPL++, which is of course a C++ interface (only).

Many examples exist in the literature of systems that automatically parallelize an application. Such approaches usually provide a mechanism to allow the application to be independent of parallelism, and an optimization mechanism separate from the application. For example, Parallel-Horus also includes a framework that allows the implementation to self-optimize for different platforms [23]. An early example of automatic mapping of signal processing flow graphs to a parallel machine was provided by Printz [24]. Moore *et al.* demonstrated a graphical model-based approach that optimally maps a signal flow graph for an image processing application onto a parallel hardware architecture given a set of constraints and benchmarks of the key components [25]. Squyres *et al.* built software to transparently distribute processing operations, written in C, onto a cluster [26]. The implementation described uses "worker nodes" to perform the parallel image processing. In contrast to these approaches, the Parallel VSIPL++ standard does not specify a self-optimizing mechanism that would automatically adapt either the library or an application to a particular platform. However, it does provide a portable layer for the expression of parallelism that could be exploited by different self-optimizing systems. An example framework that accomplishes this is provided by the Self-optimizing Software for Signal Processing (S^3P) effort described in Section IV.

III. A Prototype for Parallel VSIPL++: the parallel vector library (PVL)

Parallel VSIPL++ follows PVL in achieving map independence by raising the level of abstraction at which applications are written. The high level of abstraction allows the library writer more opportunities for platform-specific optimization and makes application code more portable.

The core capability of PVL is that of assigning or *mapping* 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. Three categories of user-level objects can be mapped: data objects (that is, vectors and matrices), computation objects (such as the QR factorization object in Figure 6), and tasks.

In broad terms, users obtain task parallelism by writing an application as a set of connected and independent scopes or *tasks*. Each task represents a particular computation stage whose data and operations are independent of other tasks. Within a task, operations are performed using vectors, matrices, and computation objects, using a single-program multiple-data (SPMD) paradigm that allows for data parallelism. Tasks may be mapped to different processor groups to allow the system to perform different computations at the same time. Special communication objects called *conduits* allow communication of data objects between different tasks. These objects are summarized in Figure 7.

For example, a simple data-parallel implementation of a basic frequency-domain filter can be constructed as shown in Figure 8. A complete filtering system can then be constructed by inserting the basic

	Class	Description	Parallelism
Signal Processing & 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)	Data & Task
	Task	Supports algorithm decomposition (i.e. the boxes in a signal flow diagram)	Task & Pipeline
	Conduit	Supports data movement between tasks (i.e. the arrows on a signal flow diagram)	Task & Pipeline
Mapping	Map	Specifies how Tasks, Vectors/Matrices, and Computations are distributed on processor	Data, Task & Pipeline
	Grid	Organizes processors into a 2D layout	

Fig. 7. **PVL Library Objects.** PVL Library Components. PVL is based on four basic user level object types (highlighted in yellow): vectors and matrices, computations (for example, an FFT), tasks, and conduits. Each of these is independently mappable onto a grid of processors.

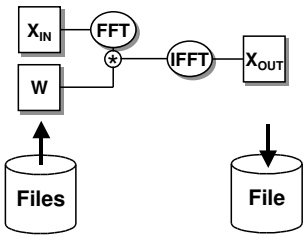

Fig. 8. **Basic Filtering Application.** PVL Vector/Matrix and Computation Objects allow signal processing algorithms to be implemented quickly using high level constructs. (The "*" symbol denotes vector elementwise multiplication.)

filtering algorithm into a task and adding appropriate input tasks and output tasks connected by conduits (see Figure 9). In each case, the application code does not include any references to the number of processors used and can therefore be mapped to any parallel architecture.

In subsequent sections, we give more detail of the mapping (section III-A), data parallel (section III-B), and task parallel (section III-C) features of PVL, comparing these to Parallel VSIPL++ where there are differences.

A. Maps

Both PVL and Parallel VSIPL++ achieve parallelism through the use of map objects. Maps consist of a set of nodes and a description of how those nodes are to be used. Objects that can have a map are called *mappable*. In PVL, 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. In Parallel VSIPL++, maps for matrices and vectors are

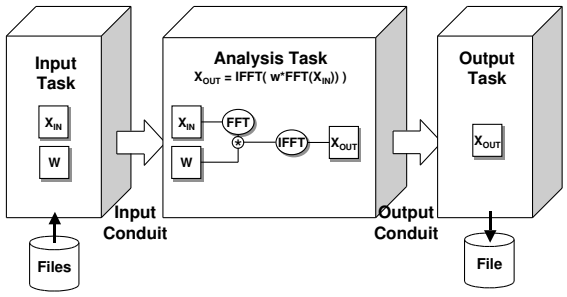

Fig. 9. **Basic Filtering System.** PVL Task and Conduit objects allow complete systems to be built. (The "*" symbol denotes vector elementwise multiplication.)

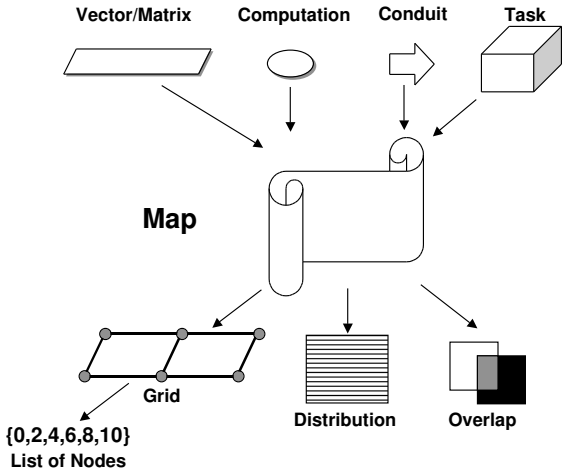

Fig. 10. **Structure of a PVL Map Object.** All PVL user objects contain a map. Each map is composed of three components: a grid, a distribution description, and an overlap description. Within the grid is the list of physical nodes onto which the object is mapped.

associated with the underlying blocks: this is consistent with the idea that blocks are storage mechanisms and views are used to operate on data.

A PVL map includes a two-dimensional grid with a specific list of nodes and a distribution description specific to the type of object. For example, the distribution description of a data object such as a matrix or vector determines how data are to be distributed among processors (block, cyclic, or block-cyclic). A distribution also includes an overlap description that indicates whether elements should be duplicated on multiple processors: this may be done, for example, to support boundary conditions. Figure 10 shows the structure of a PVL map object; the structure of a Parallel VSIPL++ map object is similar.

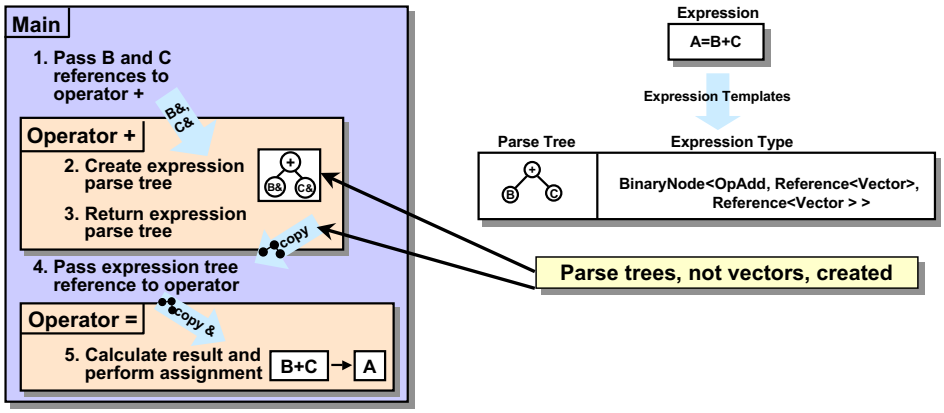

Fig. 11. **C++ Expression Templates.** Obtaining high performance from C++ requires technology that eliminates the normal creation of temporary variables in an expression. Los Alamos PETE technology makes this possible.

B. Data-parallel programming with PVL

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

$$A = B + C * D,$$

where A, B, C and D are all distributed vectors or matrices. Such expressions are enabled by the *operator overloading* feature of C++ [27]. A naive implementation of operator overloading in C++ will result in the creation of temporary data structures for each sub-step 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 (see Figure 11). In many instances it is possible to achieve better performance with expression templates than using standard C based libraries because the C++ expression-template code can achieve superior cache performance for long expressions [5].

In both PVL and Parallel VSIPL++, any block-cyclic distribution of data is supported (for a complete description of the block-cyclic data distribution, see [28]). The Parallel VSIPL++ library provides explicit support for the well-known special cases of *block distribution* and *cyclic distribution*, illustrated in Figure 12. In the block distribution, processors are assigned nearly equal, contiguous chunks of data. In the cyclic distribution case, processors alternately receive single data elements. Designating a data object

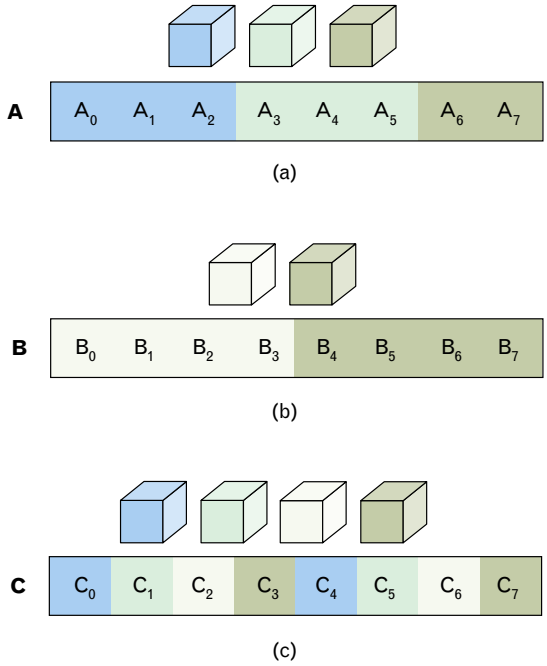

Fig. 12. **Distribution Example.** Example distributions of a vector of length eight: (a) block distribution on three processor nodes, (b) block distribution on two processor nodes, (c) cyclic distribution on four processor nodes.

as having one of these special case distributions allows the compiler to better optimize the data reference operations associated with the object. Similar designations were provided in POOMA [6].

In the case of Parallel VSIPL++, the application program may explicitly specify the distribution, in which case a recompile will be required to change the maps. Alternately, the map may be stored in an external file, which allows the map to be changed without changing the application code. This approach, which was followed by PVL, is also allowed in Parallel VSIPL++. 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).

Consider the PVL code to perform a simple add, shown in Figure 13. Intuitively, it is easy to understand that this code fragment adds the vectors `b` and `c` to produce `a`. However, the mechanism used to achieve this result is not obvious. We use the portable expression template engine (PETE, [5]) to define operations on vectors and matrices. PETE allows us to describe a mathematical operation in a *PETE expression* and defer execution of that operation until after all parts of the operation are known. This has the potential

```
void addVectors(const int vecLength)
    {
    Vector< Complex<Float> > a("a", vecLength, aMap);
    Vector< Complex<Float> > b("b", vecLength, bMap);
    Vector< Complex<Float> > a("c", vecLength, cMap);
    // Fill the vectors with data
    generateVectors(a,b,c);

    a=b+c;

    // Check results and end
    }
```

Fig. 13. PVL code to add two vectors. The distribution of each vector is described by its map object.

to allow optimized evaluation of expressions by eliminating temporary objects. The operations contained in a PETE expression are processed by an object called an *evaluator*, which stores the parse tree for the expression as seen in Figure 11.

Add, subtract, multiply, and divide operations in PVL are performed by a binary element-wise computation object (called a `BinaryNode` in Figure 11) internal to the evaluator. This object contains code to move the input data to the working locations, actually perform the add, and move the sum to the desired output location.

Thus, the simple statement `a=b+c` actually triggers the following sequence of events:

1) a PETE expression for the operation is created that references `b` and `c` and records that the operation is an add;
2) the assignment operator (`operator=`) for vector uses an evaluator to interpret the expression; and
3) the evaluator calls its internal binary element-wise computation object to perform the add and assign its results to `a`.

Creating the evaluator object is a time-intensive operation, and one that can be optimized using early binding. Therefore, the Parallel VSIPL++ and PVL libraries provide an optimization method, `setupAssign`, that creates the evaluator for the particular expression and stores it for future reference. When the assignment operator for a distributed view is called, it checks whether an evaluator has been stored for this particular expression, and if so, uses that evaluator to perform the expression rather than creating one. If the evaluator has not been stored, one is created, stored, and associated with the view for future use. This approach provides early binding for system deployment without requiring it in system prototyping.

Figures 14 shows the performance achieved using templated expressions on a single Linux PC. We compare expressions of increasing length ($A = B+C$, $A = B+C*D$, and $A = B+C*D/E+\text{FFT}(F)$),

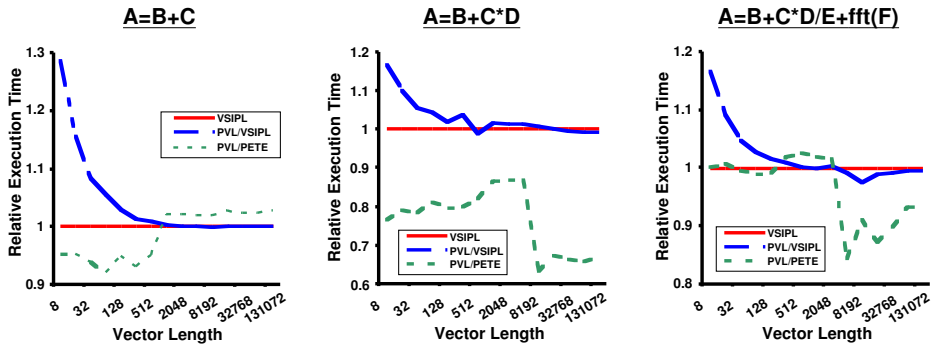

Fig. 14. **Single Processor Performance.** Comparison of the single processor performance VSIPL (C), PVL (C++) on top of VSIPL (C), and PVL (C++) on top of PETE (C++), for different expressions with different vector lengths. PVL with VSIPL or PETE is able to equal or improve upon the performance of VSIPL.

and examine the performance of three different approaches: the VSIPL reference implementation, PVL layered over the VSIPL reference implementation, and PVL implemented using expression templates (bypassing VSIPL altogether). Notice that layering PVL on of VSIPL can introduce considerable overhead for short vector lengths: this overhead is eliminated by the expression template approach. For long expressions, code that uses templated expressions is able to equal or exceed the performance of VSIPL.

Figures 15 and 16 show the performance achieved using templated expressions on a four-node cluster of workstations, connected using gigabit Ethernet. The expressions used are the same as in Figure 14, and the approaches are the same, except that the basic approach uses a combination of VSIPL and MPI in C. In Figure 15, all the vectors are identically distributed, so no communication needs to be performed *except* by the FFT operation. Therefore, the first two expressions show results similar to the single-processor results: for the third expression, the communication cost of the FFT dominates and all approaches are roughly comparable. In Figure 16, vector A is distributed over two nodes, and the remaining vectors are distributed over 4 nodes each. This introduces the requirement to communicate the results from one set of nodes to another, and this communication cost dominates the computation time so that all the approaches have similar performance.

Figures 14 through 16 show comparison of expression templates with the unoptimized, VSIPL reference implementation as a proof-of-concept. However, it is obviously important for the library to be able to leverage all the features of a hardware architecture. An example of such a feature is the Altivec extensions provided by the PowerPC G4 processor: these are C-language extensions that provide short vector types and new operations, corresponding to low-level processor instructions, on those vector types [29]. Franchetti and Püschel have shown that SPIRAL can generate code that uses such extensions, and have

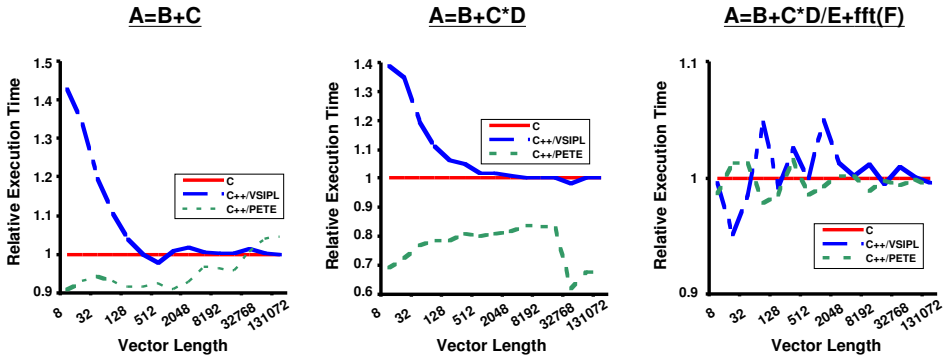

Fig. 15. **Multi-Processor (no communication).** Comparison of the multi processor (no communication) performance of VSIPL (C), PVL (C++) on top of VSIPL (C), and PVL (C++) on top of PETE (C++), for different expressions with different vector lengths.

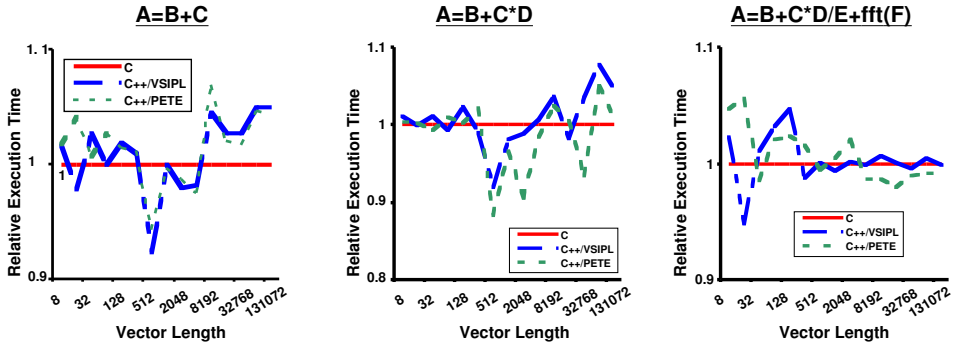

Fig. 16. **Multi-Processor (with communication).** Comparison of the multi processor (with communication) performance of VSIPL (C), PVL (C++) on top of VSIPL (C), and PVL (C++) on top of PETE (C++), for different expressions with different vector lengths.

implemented it using the similar SSE and SSE-2 extensions for Intel architectures [30]. Similarly, Rutledge demonstrated that PETE can make use of Altivec extensions directly and achieve comparable or better performance to optimized implementations of VSIPL by doing so [31]. A summary of his results is shown in Figure 17. He compared a hand-generated Altivec loop (assuming unit stride) with an Altivec-optimized VSIPL implementation provided by MPI Software Technology, Inc., and PVL using PETE (again assuming unit stride) to generate Altivec instructions, for a series of expression. The VSIPL implementation achieves lower performance on average. At least in part, this probably has to do with the inability of the VSIPL implementation to assume unit stride, but it has more to do with the necessity in C-based VSIPL to perform multiple function calls to evaluate long expressions. The most encouraging

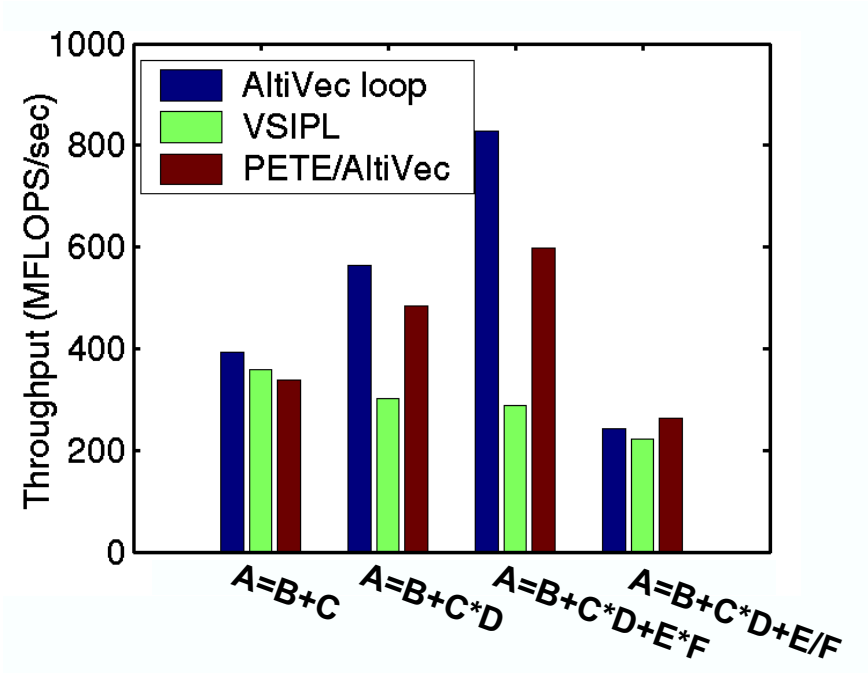

Fig. 17. **Combining PETE with Altivec.** Comparison of average throughput achieved by a hand-generated Altivec loop, an optimized VSIPL implementation, and PETE modified to use Altivec instructions.

result, however, is that PVL using PETE is able to achieve performance comparable to handwritten Altivec code.

Finally, Parallel VSIPL++ extends the block functionality of VSIPL in an important way, allowing the user to define blocks and thereby extend the library. Bergmann *et al.* used this capability in a novel way to allow portions of a Parallel VSIPL++ program to run on FPGAs [32]. The ability to have close interaction between FPGAs and programmable processors through the Parallel VSIPL++ library has the potential to improve performance and shorten development time for signal processing applications.

C. Task-parallel programming with PVL

Task-parallel programming is critical for embedded applications, which demand not only low latency (processing time for an individual data sets) but also high throughput (ability to keep up with a continuous stream of data sets). Because the best implementation of task parallelism is often application-dependent, Parallel VSIPL++ does not specify a standard set of task-parallel constructs. However, the concept of map independence exemplified by PVL and standardized in Parallel VSIPL++ can be applied to task-parallelism as well as data-parallelism. Further, the PVL task-parallel programming constructs are

reproducible using Parallel VSIPL++. In this section, we describe the task-parallel constructs of PVL in more detail.

A PVL program consists of a set of tasks connected by data transport objects called *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 using the conduits. An individual data object exists only in the scope of one particular task at a time; 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 multi-buffering. The use of the conduit abstraction separates the application programmer from the details of the communication service being used.

Conduits are critical because, in many signal processing applications, the primary challenge is not performing the computations but moving data so that it is in a position to be acted upon. As previously described (section II-A), one of the most common examples of this is the "corner turn" operation. 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, setup all necessary buffers, create multiple buffers if necessary, and set up special hardware for communication. Thus, the library is provided with more opportunities for optimization of the communication operation.

The major benefit of the PVL 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, pipelined stages, the system designer gains an extra level of parallelism. This allows customization of the system resources for each individual stage. An alternative implementation would have all the system resources work as a monolithic entity 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 approach versus the monolithic approach. However, an increase in throughput is obtained using the pipelined approach because several data sets are being processed at the same time. In addition, gains in efficiency may be realized due to the smaller number of processing resources assigned to each stage, and communication may occur concurrently with computation.

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 independent of other pieces.

D. Summary of Parallel VSIPL++ Functionality

At the completion of the Parallel VSIPL++ standard, it will be possible to use the library to write multi-stage algorithms that are independent of the underlying hardware. As described in previous sections, programs will be written using four user-level objects: data objects (vectors and matrices), computations, tasks, and conduits (see Figure 7). The mapping of the user-level objects to processors will be controlled by the mapping objects shown in the same figure. The standard will provide the functionality of VSIPL, and will include the key features of early binding, separation of memory and mathematical objects, and library ownership of data that are used to enable platform-specific adaptation for VSIPL. We have demonstrated that we can extend PETE to take advantage of platform-specific vector optimizations such as Altivec. Most importantly, the standard will provide the capability of adapting the same application code to different parallel platforms by changing the maps.

The major benefit of Parallel VSIPL++ is the portability and re-usability 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 a mapping of the entire application to a single processor, and then re-tested 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.

IV. Self-Optimization of PVL and Parallel VSIPL++ Programs

Like PVL, Parallel VSIPL++ allows algorithm concerns to be separated from mapping concerns. However, the problem of determining the mapping of an application has not changed. As software is moved from one piece of hardware to another, the mapping will need to change. The current standard approach is for humans with both knowledge of the algorithm and the parallel computer to use intuition and simple rules to estimate what the best mapping will be for a particular application on a particular hardware platform. This method is time consuming, labor intensive, inaccurate, and 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. Many of the prototype approaches in

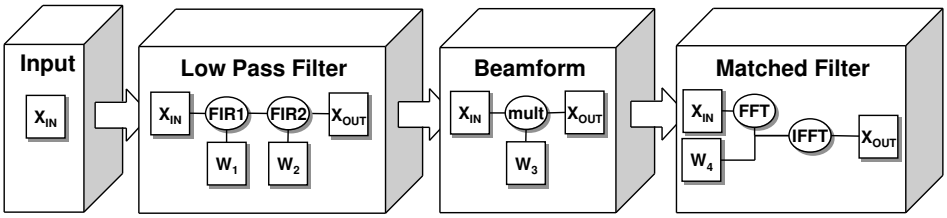

Fig. 18. **Multistage example application.** The example signal processing application used for S^3P.

the literature described in Section II-F include a demonstration of this type of approach. Hoffmann and Kepner developed a framework called Self-optimizing Software for Signal Processing (S^3P,[33]) which demonstrates the same ideas using PVL. S^3P is based on PVL, but the key principle of map abstraction that PVL shares with Parallel VSIPL++ enables similar constructs to be built using Parallel VSIPL++.

S^3P is designed to automate mapping of a single application onto a dedicated cluster resource. Figure 18 shows the example application that is used. The application is meant to exemplify a typical processing stream from radar, sonar, or communications. It is a directed, acyclic pipeline among four processing stages. The input stage creates the input data and sends them on to the low-pass filter stage. The low-pass filter stage reduces the bandwidth of the input data. The beamforming stage transforms the filtered data to allow detection of signals coming from a particular set of directions of interest. The detection stage determines whether targets are actually present in the beamformed data. A corner turn is required between the low-pass filter and beamforming stages.

The S^3P framework (Figure 19) is used to perform optimization on an application. It requires certain capabilities from an application. First, the application must be made of tasks which are capable of being mapped to multiple configurations of hardware. Furthermore, each task needs to be able to measure or estimate its computing resources (e.g. number of processors, memory, and execution time) in each configuration. These capabilities are provided by the task object in PVL.

Given these capabilities, S^3P's map generator can assemble a system graph that represents all candidate mappings of the problem. The graph has T columns, where T is the number of tasks to be mapped: column k consists of M_k nodes, where there are M_k possible mappings of task k. An edge in the graph has a value corresponding to the time taken to communicate data from one stage with a particular mapping to the following stage with another mapping. The map timer in S^3P is used to obtain from each task the times associated with each node and edge. An example system graph is shown in Figure 20.

Once the system graph containing all possible mappings has been constructed, the map selector in

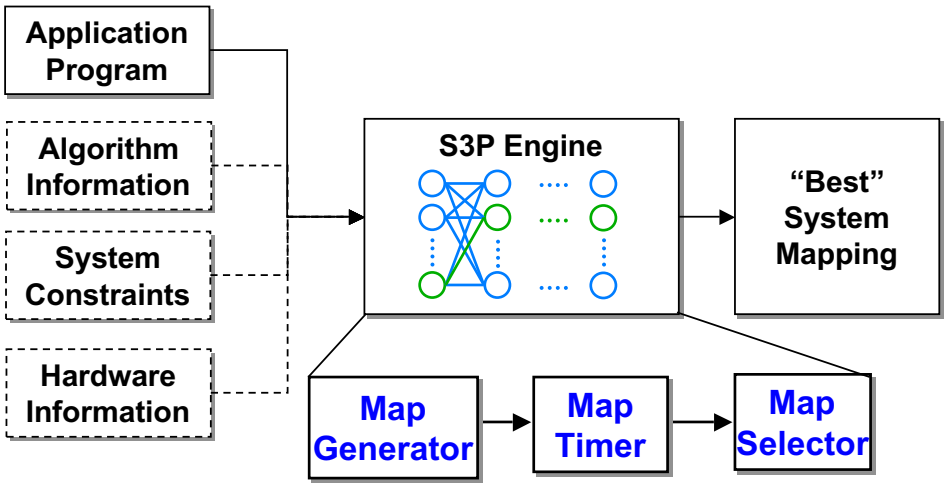

Fig. 19. **Self-optimizing Software for Signal Processing (S^3P).** This framework combines the dynamic mapping capabilities found in PVL with self-optimizing software techniques. The resulting framework allows the optimal mapping of an application to be found automatically for any parallel architecture.

S^3P searches for the optimal set of maps. It does so by using dynamic programming to find the "best" path through this graph, which corresponds to the optimal mapping. Thus the entire process of finding an optimal mapping can be completely automated.

The S^3P framework has been tested on different problem sizes and with different criteria, and it is able to pick out the correct mapping and to predict the performance of the best map to within a few percent. Figure 21 shows S^3P performance for two problem sizes and two performance criteria – latency and throughput. In each example, S^3P finds the correct mapping and predicts the achieved performance.

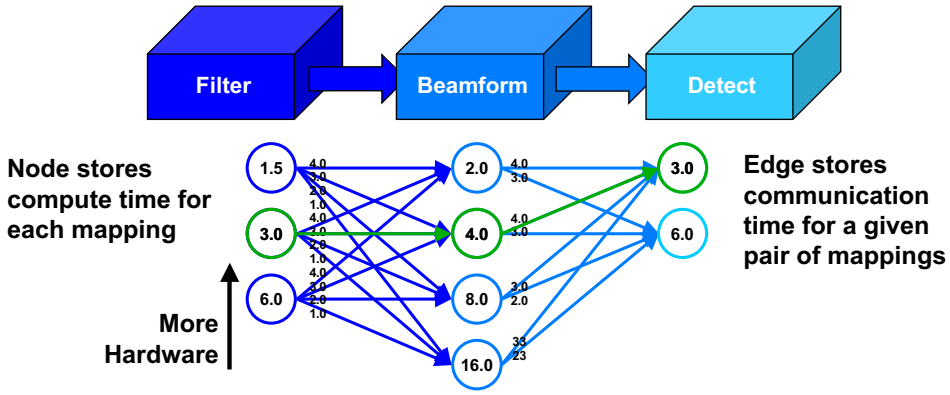

Fig. 20. **Example graph generated by S^3P.** This is the system graph associated with mapping the multistage application (Figure 18) onto a cluster. Nodes store compute time for each mapping, and edges store communication time for a given pair of mappings. The path through the graph that maximizes throughput given a hardware constraint is shown in green: this path is chosen because it has the smallest bottleneck among all paths that satisfy the constraint.

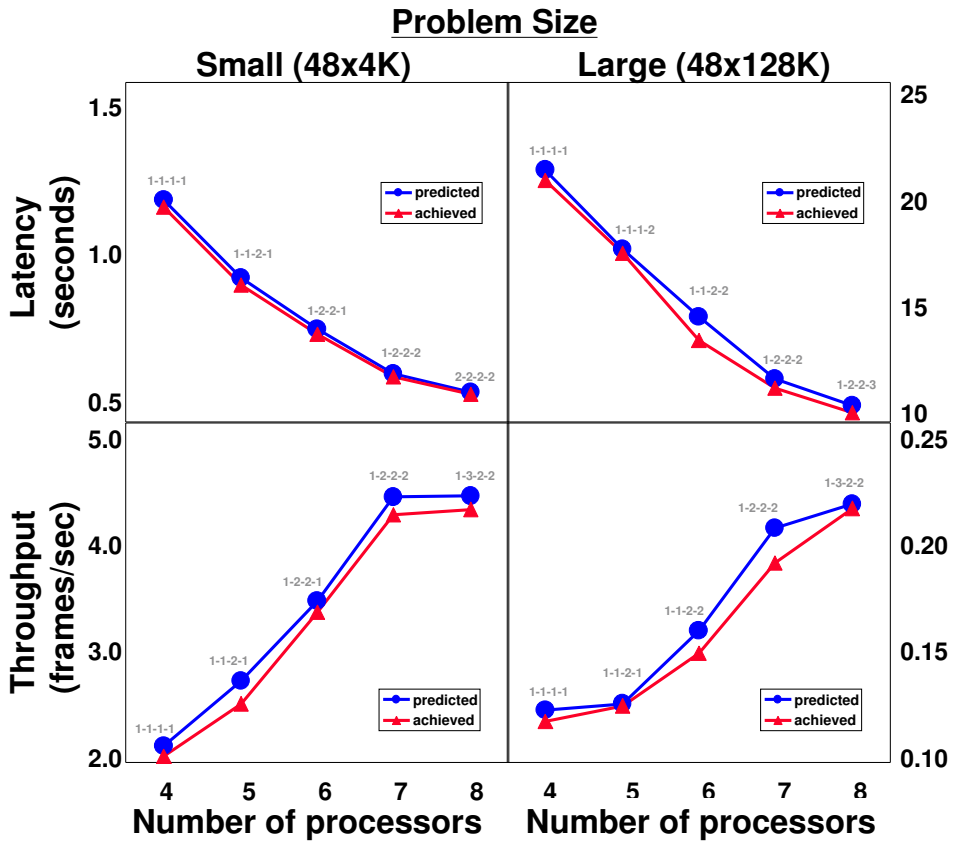

Fig. 21. S^3P **Performance.** The S^3P framework is tested on two different problem size in a four stage application with two different criterion: minimize latency for a given number of processors; maximize the throughput for a given number of processors. In all cases S^3P picks the correct mapping and predicts the performance of the best map to within a few percent. The number of processors to be used in each stage of the application is shown in grey.

V. Summary and Conclusions

Parallel VSIPL++ is designed to address the unique challenges presented by parallel embedded signal processors. Specific examples of these challenges can be found in Ground Moving Target Indicator (GMTI) radars and missile based image processors, both of which exhibit the needs for task, pipeline, and data parallelism that are common in the signal processing applications. These applications have strict timing requirements which demand high performance.

In this paper we have presented an overview of the Parallel VSIPL++ standard and the ways in which it supports platform optimization. C++ expression template technology is exploited to allow the compiler to do hardware specialization. This includes the elimination of unnecessary temporary variables in array expressions, use of pointer arithmetic in for loops, and use of processor specific instructions (e.g. vector multiply and accumulate). VSIPL++ standardizes many computations which are used routinely in signal processing. These computation objects (e.g. Fast Fourier Transforms, Finite Impulse Response filters, QR decompositions) are built with explicit setup and run stages. The setup stage allows for additional tuning based on information that is only available at run time (e.g. specific vector or matrix sizes).

Parallel arrays and functions in Parallel VSIPL++ also support expression templates and setup and run stages. These features allow communication buffers to be allocated in advance and are used to accelerate the determination of which parts of an array need to be computed and which parts can be kept local. Determining the optimal allocation of different computation stages to different processors is an extremely challenging task, and there are a variety of ways of addressing this problem. Parallel VSIPL++ provides an external parallel mapping interface that allows optimal mappings to be generated offline and read into the system at run-time. S^3P is one example of this type of technology.

Finally, parallel VSIPL++ is a community specified standard, which has benefited from extensive input from users, vendors, tool builders, and researchers. Specifically, the standard has been developed in collaboration with high performance embedded computing vendors and is compatible with their proprietary approaches to achieving performance.

Acknowledgments

The Parallel VSIPL++ library builds on previous work of the VSIPL forum and of MIT Lincoln Laboratory. The reference implementation of the Parallel VSIPL++ library was built by Jeffrey Oldham and Mark Mitchell of CodeSourcery, LLC. The GMTI application was described in a technical report by Albert Reuther of MIT Lincoln Laboratory. The Standard Missile application was described in a presentation by Daniel Rabinkin, Edward Rutledge, and Paul Monticciolo of MIT Lincoln Laboratory.

The authors gratefully acknowledge the support of Charlie Holland, John Grosh, Rich Linderman, Richard Games, Ed Baranoski and the entire HPEC-SI community. Bob Bond has provided invaluable technical guidance throughout this effort. Finally, we would like to thank several anonymous referees whose comments significantly improved this article.

BIOGRAPHIES

James Lebak received B.S. degrees in Mathematics and Electrical Engineering in 1989, and the M.S. degree in Electrical Engineering in 1991, both from Kansas State University, Manhattan, KS. He received the Ph.D. degree in Electrical Engineering from Cornell University, Ithaca, NY, in 1997, and is currently employed by MIT Lincoln Laboratory as a technical staff member in the Embedded Digital Systems group. He is co-chair of the VSIPL forum, and is interested in parallel and numerical algorithms for digital signal processing. E-mail: jlebak@ll.mit.edu

Jeremy Kepner received his B.A. in Astrophysics from Pomona College (Claremont, CA). He obtained his Ph.D. in Astrophysics from Princeton University in 1998, after which he joined MIT Lincoln Lab. 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. E-mail: kepner@ll.mit.edu

Henry Hoffmann received his B.S. with highest honors in Mathematical Sciences from the University of North Carolina at Chapel Hill in 1999. He received his S.M. degree in Electrical Engineering and Computer Science from the Massachusetts Institute of Technology in 2003 as part of MIT Lincoln Laboratory's Lincoln Scholars Program. He is currently a technical staff member at Lincoln Laboratory and a Ph.D. student working with Anant Agarwal on the Raw microprocessor. His research addresses parallel algorithms and architectures in the face of computation, area, and energy efficiency concerns. E-mail: hoffmann@ll.mit.edu

Edward Rutledge received his B.S. in Computer Science from North Carolina State University in May of 1997. He joined MIT Lincoln Laboratory in July of 1997, and is currently an associate staff member at the Lab. At MIT Lincoln Laboratory, Eddie's focus has been on parallel signal processing software. He has participated in the design and development of portable, parallel, embedded, real-time signal processing middleware, and has also participated in the design and development of parallel signal processing systems and applications. E-mail: rutledge@ll.mit.edu

REFERENCES

[1] J. Lebak, R. Janka, R. Judd, M. Richards, and D. Campbell, "VSIPL: An object-based open standard API for vector, signal, and image processing," in *Proceedings of the 2001 IEEE International Conference on Acoustics, Speech, and*

Signal Processing, May 2001.

[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," Georgia Tech Research Corporation, Tech. Rep., 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," in *ISCOPE '97: First Conference on International Scientific Computing in Object-Oriented Parallel Environments*, Dec. 1997.

[4] H. Hoffmann, J. Daly, J. Matlis, P. Richardson, E. Rutledge, and G. Schrader, "Achieving portable task and data parallelism on signal processing architectures," in *Proceedings of the Fourth Annual High-Performance Embedded Computing (HPEC) Workshop*, Sept. 2000.

[5] S. Haney, J. Crotinger, S. Karmesin, and S. Smith, "Easy expression templates using PETE, the portable expression template engine," *Dr. Dobb's Journal*, Oct. 1999.

[6] J. C. Cummings, J. A. Crotinger, S. W. Haney, W. F. Humphrey, S. R. Karmesin, J. V. Reynders, S. A. Smith, and T. J. Williams, "Rapid application development and enhanced code interoperability using the POOMA framework," in *Proceedings of the SIAM workshop on Object-oriented methods and code interoperability in scientific and engineering computing (OO98)*, Oct. 1998.

[7] A. I. Reuther, "Preliminary design review: Narrowband GMTI processing for the basic PCA radar-tracker application," MIT Lincoln Laboratory, Lexington, MA, Project Report PCA-IRT-3, Feb. 2003.

[8] D. Rabinkin, E. Rutledge, and P. Monticciolo, "Missile seeker common computer signal processing architecture for rapid technology upgrade," in *Proceedings of the Sixth Annual High-Performance Embedded Computing Workshop*, Sept. 2002.

[9] ——, "Missile seeker common computer architecture for rapid technology upgrade," in *SPIE Proceedings Volume 5559*, July 2004.

[10] C. F. Van Loan, *Computational Frameworks for the Fast Fourier Transform*. Society for Industrial and Applied Mathematics, 1992.

[11] J. Mirod, "Continuous real-time signal processing: comparing the TigerSHARC and the PowerPC," *COTS Journal*, no. 12, pp. 32–37, Dec. 2002.

[12] R. Teachey, A. Donadeo, E. Pancoast, G. Faix, and B. Chin, "COTS software portability standards and VSIPL benchmarks," in *Proceedings of the Fourth Annual High-Performance Embedded Computing (HPEC) Workshop*, Sept. 2000.

[13] D. Averill, "Successful VSIPL software application migration – a case study: NATO Seasparrow illumination radar signal processing," in *Proceedings of the Seventh Annual High-Performance Embedded Computing (HPEC) Workshop*, Sept. 2003.

[14] MPI Forum, "MPI: A message-passing interface standard," University of Tennessee, Tech. Rep., Apr. 1994.

[15] A. Skjellum and P. V. Bangalore, "Interfacing to VSIPL," in *VSIPL Tutorial and Users Group Meeting*, Feb. 2002. [Online]. Available: http://www.vsipl.org

[16] G. H. Golub and C. F. Van Loan, *Matrix Computations*, 3rd ed. Johns Hopkins University Press, 1996.

[17] T. Veldhuizen, "Expression templates," *C++ Report*, vol. 7, no. 5, 1995.

[18] P. J. Morrow, D. Crookes, J. Brown, G. Mcaleese, D. Roantree, and I. Spence, "Efficient implementation of a portable parallel programming model for image processing," *Concurrency: Practice and Experience*, vol. 11, no. 11, pp. 671–685, 1999.

[19] M. Püschel, B. Singer, M. Velosa, and J. Moura, "Fast automatic generation of DSP algorithms," in *Proceedings of ICCS 2001*. Springer-Verlag, 2001, pp. 97–106.

[20] B. Singer and M. Veloso, "Learning to construct fast signal processing implementations," *Journal of Machine Learning Research*, vol. 3, pp. 887–919, 2002.

[21] M. Frigo and S. G. Johnson, "FFTW: An adaptive software architecture for the FFT," in *Proceedings of the 1998 IEEE International Conference on Acoustics, Speech, and Signal Processing*, vol. 3. IEEE Signal Processing Society, May 1998, pp. 1381–1384.

[22] R. C. Whaley, A. Petitet, and J. J. Dongarra, "Automated empirical optimization of software and the ATLAS project," University of Tennessee, Tech. Rep., Sept. 2000, LAPACK working note 147. [Online]. Available: http://www.netlib.org/lapack/lawns/lawn147.ps

[23] F. Seinstra and D. Koelma, "User transparency: A fully sequential programming model for efficient data parallel image processing," *Concurrency and Computation: Practice and Experience*, vol. 16, no. 7, 2004.

[24] H. Printz, H. Kung, T. Mummert, and P. Scherer, "Automatic mapping of large signal processing systems to a parallel machine," in *SPIE Proceedings Volume 1154: Real-time signal processing XII*, J. P. Letellier, Ed., Aug. 1989, pp. 2–16.

[25] M. S. Moore, J. Sztipanovitz, G. Karsai, and J. Nichols, "A model-integrated program synthesis environment for parallel/real-time image processing," in *SPIE Proceedings Volume 3166: Parallel and distributed methods for image processing*, H. Shi and P. C. Coffield, Eds., July 1997.

[26] J. M. Squyres, A. Lumsdaine, and R. L. Stevenson, "A toolkit for parallel image processing," in *SPIE Proceedings Volume 3452: Parallel and Distributed Methods for Image Processing II*, July 1998, pp. 69–80.

[27] B. Stroustrup, *The C++ Programming Language*, 3rd ed. Addison-Wesley,Inc., 1997.

[28] High Performance Fortran Forum, *High Performance Fortran Language Specification*, 1993.

[29] *AltiVec Technology Programming Interface Manual*, Motorola Semiconductor Products, June 1999.

[30] F. Franchetti and M. Püschel, "Short vector code generation for the discrete Fourier transform," in *Proceedings of International Parallel and Distributed Processing Symposium*, Apr. 2003.

[31] E. Rutledge, "Altivec extensions to the portable expression template engine (PETE)," in *Proceedings of the Sixth Annual High-Performance Embedded Computing (HPEC) Workshop*, Sept. 2002.

[32] J. Bergmann, S. Emeny, and P. Bronowicz, "VSIPL++/FPGA design methodology," in *Proceedings of the Seventh Annual High-Performance Embedded Computing (HPEC) Workshop*, Sept. 2003.

[33] H. Hoffmann, J. V. Kepner, and R. A. Bond, "S3P: automatic, optimized mapping of signal processing applications to parallel architectures," in *Proceedings of the Fifth Annual High-Performance Embedded Computing (HPEC) Workshop*, Nov. 2001.