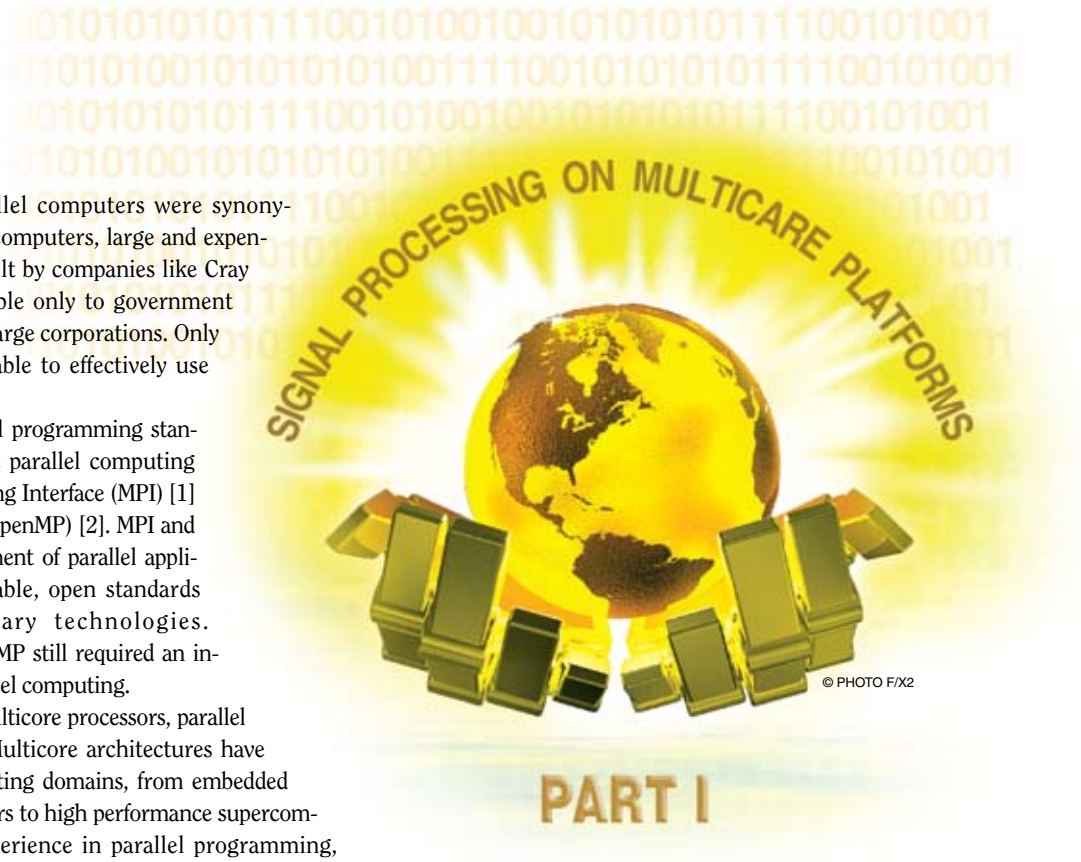
Hahn Kim and Robert Bond

Multicore Software Technologies

A survey



For decades, parallel computers were synonymous with supercomputers, large and expensive machines built by companies like Cray and IBM, affordable only to government laboratories and large corporations. Only expert programmers were able to effectively use these systems.

In the 1990s, two parallel programming standards grew to dominate the parallel computing landscape: the Message Passing Interface (MPI) [1] and Open Multiprocessing (OpenMP) [2]. MPI and OpenMP simplified development of parallel applications by promoting portable, open standards over multiple proprietary technologies. Nevertheless, MPI and OpenMP still required an in-depth understanding of parallel computing.

Today, with the rise of multicore processors, parallel computing is everywhere. Multicore architectures have quickly spread to all computing domains, from embedded systems to personal computers to high performance supercomputers. Knowledge and experience in parallel programming, unfortunately, have not kept pace with the trend towards parallel hardware.

Multicore architectures require parallel computation and explicit management of the memory hierarchy, both of which add programming complexity and are unfamiliar to most programmers. While MPI and OpenMP still have a place in the multicore world, the learning curves are simply too steep for most programmers. New technologies are needed to make multicore processors accessible to a larger community. The signal and image processing (SIP) community stands to benefit immensely from such technologies. For examples of key SIP algorithms, see [3] and [4].

This article provides a survey of new software technologies that hide the complexity of multicore architectures, allowing programmers to focus on algorithms instead of architectures.

TECHNOLOGY CAPABILITIES

This section describes various capabilities found in many software technologies that enable programmers to program a range of multicore architectures. These capabilities are summarized in Table 1.

Digital Object Identifier 10.1109/MSP.2009.934141

Multicore designs range from conventional, homogeneous architectures (e.g., multicore x86) to novel, heterogeneous architectures (e.g., IBM Cell). Homogeneous multicore systems continue to be popular and many software technologies provide programming interfaces that are more approachable than MPI and OpenMP. Many emerging architectures, however, have heterogeneous designs that use a host processor for program control and coprocessors for computation. Graphics processing units (GPUs) supplement a conventional processor, e.g., x86, which acts as the host. IBM's Cell contains a host [PowerPC Processor Element (PPE)] and multiple coprocessors [Synergistic Processor Elements (SPEs)] on the same chip [5]. Field-programmable gate arrays (FPGAs) are becoming supplements or alternatives to multicore processors.

Architectures such as these have explicit memory hierarchies that span the host and coprocessor. Efficient data movement across the memory hierarchy is key to achieving high performance. Many new software technologies have interfaces for managing computation and data movement for coprocessors.

Data parallelism is a form of parallelism where data are distributed across multiple processors, with each processor performing the same computation, thus reducing the total amount of work on each processor. Data parallelism is often synonymous with the single-program multiple-data (SPMD) programming model, in which the same program executes on multiple processors but processes different data [6].

Traditionally, the programmer is responsible for allocating memory on each processor and "bookkeeping," e.g., keeping track of which data elements are on each processor. Many of the technologies discussed in this article contain distributed array constructs. Distributed (or parallel) arrays take care of memory allocation and data movement across processors and include bookkeeping capabilities, relieving the programmer of this burden.

Task parallelism is a form of parallelism where different computational tasks are assigned to different processors, each processing either the same or different data. Task parallelism is often synonymous with the multiple-program multiple-data (MPMD) model, in which different processors execute different programs, processing the same or different data [6]. Task parallelism enables data flow patterns, e.g., pipelines, fork-joins, and round-robining, common to real-time streaming data, e.g., video.

The default mechanism for running parallel programs on multicore processors is to run a separate process on each core. Threads enable a lighter weight approach to processes [7]. Technologies implement threads (and similar constructs) with support from software and/or hardware.

Multicore software technologies can be implemented as languages or libraries. New languages are often extensions of existing languages, e.g., C. In some cases, existing languages impose restrictions that limit their ability to express new programming constructs, warranting an entirely new language. Libraries support a range of languages, e.g., MATLAB, C, and C++. Table 1 identifies which technologies are languages or libraries and what language the technology is based on, if applicable.

Porting software between different parallel architectures has always been difficult. The wide range of multicore architectures now available only increases the difficulty of this task. A number of software technologies support multiple architectures, minimizing the difficulty of porting applications between different multicore architectures.

LANGUAGES

The vast majority of programming languages are designed for serial computation. There are a number of initiatives to add explicit support for parallelism to popular languages. Unified Parallel C (UPC) and Sequoia are two languages presented here

[TABLE 1] SUMMARY OF CAPABILITIES OF MULTICORE SOFTWARE TECHNOLOGIES.

TECHNOLOGY	ORGANIZATION	COPROCESSOR SUPPORT	DATA PARALLEL	TASK PARALLEL	THREAD SUPPORT	LANGUAGE/ LIBRARY	CURRENT TARGET ARCHITECTURES
BROOK+	AMD	Y	Y	Y	Y	LANGUAGE (C)	AMD GPU
CHAPEL	CRAY		Y	Y	Y	LANGUAGE	MULTICORE CPU
CILK++	CILK ARTS		Y	Y	Y	LANGUAGE (C++)	MULTICORE CPU (SHARED MEMORY X86)
CO-ARRAY FORTRAN	STANDARDS BODY		Y		Y	LANGUAGE (FORTRAN)	MULTICORE CPU
CUDA	NVIDIA	Y	Y	Y	Y	LANGUAGE (C)	NVIDIA GPU
FORTRESS	SUN/OPEN SOURCE		Y	Y	Y	LANGUAGE	MULTICORE CPU
OPENCL	STANDARDS BODY	Y	Y	Y	Y	LANGUAGE (C)	GPU
PCT	MATHWORKS		Y			LANGUAGE (MATLAB)	MULTICORE CPU
pMATLAB	MIT-LL		Y			LIBRARY (MATLAB)	MULTICORE CPU
PVL	MIT-LL		Y	Y		LIBRARY (C++)	MULTICORE CPU
PVTOL	MIT-LL	Y	Y	Y	Y	LIBRARY (C++)	MULTICORE CPU, CELL, GPU, FPGA
SEQUOIA	STANFORD	Y	Y			LANGUAGE (C)	CELL
STAR-P	INTERACTIVE SUPERCOMPUTING		Y			LIBRARY (MATLAB, PYTHON)	MULTICORE CPU
STREAMIT	MIT		Y	Y	Y	LANGUAGE	MULTICORE CPU
TITANIUM	UC BERKELEY		Y		Y	LANGUAGE (JAVA)	MULTICORE CPU
UPC	STANDARDS BODY		Y		Y	LANGUAGE (C)	MULTICORE CPU
VSIPL++	STANDARDS BODY	Y	Y			LIBRARY (C++)	MULTICORE CPU, NVIDIA GPU, CELL, FPGA
X10	IBM		Y	Y	Y	LANGUAGE (JAVA)	MULTICORE CPU

that add parallelism to C. Several other languages, including languages derived from Fortran and Java and entirely new languages, are also briefly described.

UNIFIED PARALLEL C

UPC is a parallel extension of the C Language [8]. The UPC standard is maintained by the UPC Consortium, an open group of industry, government, and academic institutions.

UPC programs are modeled as a collection of threads that share a single global address space that is logically partitioned among the threads, a programming model known as Partitioned Global Address Space (PGAS). A thread's portion of the address space is divided into private and shared spaces. Only the local thread can access variables in the private space; any thread can access variables in the shared space. Variables are allocated in the shared space by adding the `shared` keyword to the declaration. UPC supports data parallelism via shared arrays. Shared arrays are partitioned among threads such that each thread has part of the array allocated in its shared space. Figure 1 contains example code for a vector sum on shared arrays [9].

When a thread accesses data in another thread's shared space, data are communicated transparently. UPC supports a range of homogeneous multicore architectures; threads can execute on different cores and different processors. On shared memory systems, threads communicate through memory. On distributed memory systems, threads communicate through runtime layers like MPI.

Compiling a C program with a UPC compiler results in a program that runs N instances of the program in N threads. This allows programmers to write a serial C program and then parallelize the program with UPC. Also, any library with an ISO C interface is compatible with UPC. For example, the Fastest Fourier Transform in the West (FFTW) and Basic Linear Algebra Subprograms (BLASs) libraries can be used in UPC programs.

SEQUOIA

Sequoia is a research project at Stanford University [10]. Sequoia explicitly targets the management of data in a processor's memory hierarchy, i.e., allocation and movement, to achieve high performance.

Sequoia is a syntactic extension of C but introduces programming constructs that result in a different programming model. Unlike C's model, which consists of a single processor and single memory space, Sequoia exposes coprocessors and memory hierarchies. A memory hierarchy is a tree of memory modules, where modules closer to the leaves of the tree are generally smaller and faster. Hierarchical arrays are allocated across the memory tree. A hierarchical processor is composed of control processors (i.e., hosts) and processing elements (i.e., coprocessors); processing elements operate on leaf memory modules while control processors operate on nonleaf modules.

Sequoia programs consist of computational functions called tasks. A task can only access its function arguments and locally declared variables, like a regular C function. Tasks support data parallelism by executing on multiple processing elements. Tasks are written with no references to the specific architecture.

```
// Allocate shared arrays
#define N 100*THREADS
shared int v1[N], v2[N], v1plusv2[N];

// Do vector sum
int i;
upc_forall(i = 0; i < N; i++; i)
  v1plusv2[i] = v1[i] + v2[i];
```

[FIG1] Example code for UPC.

When a program is compiled for a specific machine, the programmer specifies how the program's tasks are executed on the target architecture; this is done separately from the program code. This allows the same program to be mapped to different architectures without modifying the program code. Consider the vector add example in Figure 2 [11]. The `VectAdd` task is composed of two parts, an inner task and leaf task. On the Cell, one way to map the task is to execute the inner task on the PPE and the leaf task on the SPEs. The inner task partitions arrays of size N into subarrays of size T, which are sent to SPE memory to be added by the leaf task.

OTHER LANGUAGES

Co-Array Fortran [12] and Titanium [13] are PGAS languages. Like UPC, multiple copies of a program execute on different cores but share a global address space. Co-Array Fortran and Titanium add language extensions to Fortran and Java, respectively, to distribute data and allow each copy to access data in other copies' address spaces.

Cilk++ is a parallel extension of C++ developed by Cilk Arts [14] and is based on the Cilk language developed at Massachusetts Institute of Technology (MIT) [15]. Cilk++ adds three keywords, `cilk_spawn`, `cilk_sync`, and `cilk_for`, to mark parallelism and synchronization within a program. The keyword `cilk_spawn` asynchronously launches a child function in parallel with the caller. The caller may spawn multiple functions to increase the degree of parallelism. Child functions may also call `cilk_spawn` themselves; recursive, divide-and-conquer algorithms can be parallelized in this manner. The keyword `cilk_sync` waits until all of the caller's child functions have completed, and then `cilk_for` executes a for loop's iterations in parallel.

StreamIt was developed at MIT [16] and targets applications that process streaming data, e.g., real-time audio or video. StreamIt takes advantage of the fact that streaming applications often repeatedly perform the same computations in a regular, static pattern that can execute as parallel streams of instructions. Several common data flow patterns, e.g., pipelines, fork-joins, and round-robining, are directly expressible in the language.

The Defense Advanced Research Projects Agency's High Productivity Computing Systems (HPCS) program is developing high-performance computing languages to allow applications to

```
void task VectAdd(in float A[N], in float B[N],
                  out float C[N]);

void task<inner>
VectAdd::Inner(in float A[N], in float B[N],
               out float C[N]) {
  tunable T;
  mappar (unsigned int i = 0 : N/T) {
    VectAdd(A[i*T:(i+1)*T;T],
            B[i*T:(i+1)*T;T],
            C[i*T:(i+1)*T;T]);
  }
}

void task<leaf>
VectAdd::Leaf(in float A[N], in float B[N],
              out float C[N]) {
  for (unsigned int i = 0; i < N; i++) {
    C[i] = A[i] + B[i];
  }
}
```

[FIG2] **Example code for Sequoia.**

scale future parallel systems and improve productivity of future programmers. HPCS has produced Chapel [17], Fortress [18], and X10 [19], developed by Cray, Sun, and IBM, respectively.

GENERAL-PURPOSE COMPUTATION ON GPUs

GPUs are optimized for graphics algorithms, resulting in highly parallel architectures. Researchers began to explore the use of GPUs to accelerate nongraphics algorithms, giving rise to the general-purpose computation on GPUs (GPGPU) field. Early on, algorithms had to be refactored to use graphics operations and APIs [e.g., Open Graphics Library (OpenGL)].

```
// GPU device code for vector sum kernel
__global__
void vecAdd(float *A, float *B, float *C) {
  int i = threadIdx.x + blockDim.x + blockIdx.x;
  C[i] = A[i] + B[i];
}

// Host code that calls the vector sum kernel
int main() {
  // Allocate arrays dA, dB, dC
  //(omitted for brevity)
  float *hA = ...;
  float *hB = ...;
  float *hC = ...;
  float *dA, *dB, *dC;
  cudaMalloc((void **) &dA, N * sizeof(float));
  cudaMalloc((void **) &dB, N * sizeof(float));
  cudaMalloc((void **) &dC, N * sizeof(float));

  // Copy host data to device
  cudaMemcpy(dA, hA, N * sizeof(float),
             cudaMemcpyHostToDevice);
  cudaMemcpy(dB, hB, N * sizeof(float),
             cudaMemcpyHostToDevice);

  // Run N/256 blocks of 256 threads each
  vecAdd<<<N/256, 256>>>(dA, dB, dC)
}
```

[FIG3] **Example code for CUDA.**

In recent years, more general programming technologies for GPUs have appeared, eliminating the need for a background in graphics programming. This section discusses three of these technologies: Compute Unified Device Architecture (CUDA), Brook+, and Open Computing Language (OpenCL). By design, GPUs are coprocessors and, hence, these technologies have the same basic programming model: an application running on the host allocates host and GPU memory, copies input data from host memory to GPU memory, then launches parallel kernels on the GPU.

COMPUTE UNIFIED DEVICE ARCHITECTURE

CUDA was developed by NVIDIA and was the first mainstream GPGPU programming technology [20]. CUDA represents a GPU as a hierarchy of concurrent threads. Global GPU memory is partitioned among groups of threads, called thread blocks, while threads in a thread block maintain a shared address space. CUDA supports data parallelism by running parallel kernels on one or more thread blocks. See Figure 3 for an example of a vector sum [21]. CUDA supports task parallelism by executing different kernels in different thread blocks since thread blocks execute independently of each other. CUDA allows a single host processor to launch kernels onto multiple GPUs. Other technologies, like MPI, are required if a system is comprised of multiple host processors, e.g., clusters.

CUDA consists of C language extensions plus a runtime library. The CUDA Developer SDK includes a number of components for SIP, including BLAS and FFT libraries and SIP code examples.

BROOK+

The GPGPU environment from Advanced Micro Devices (AMD), the ATI Stream SDK [22], is centered around the Brook+ language, based on BrookGPU developed at Stanford University [23]. Brook+ represents a GPU as a collection of thread processors that can issue up to five scalar operations per instruction. Brook+ extends C with constructs for expressing data parallelism using single-instruction multiple-data (SIMD) operations that execute across thread processors. Two key constructs in Brook+ are streams and kernels. Streams are a collection of data elements that can be processed in parallel and kernels are functions that operate on streams and execute across multiple thread processors. See Figure 4 for an example of a vector sum [24].

AMD also provides a GPU version of the AMD Core Math Library, which includes BLAS, Linear Algebra PACKage (LAPACK), and FFTs.

OPEN COMPUTING LANGUAGE

OpenCL is a standard for heterogeneous computing [25] maintained by the Khronos Group, which also oversees the popular graphics standard OpenGL. OpenCL was explicitly designed with abstractions that are low level and high performance but still portable. The goal is to create a foundation that allows expert programmers to create higher-level middleware libraries and

tools. OpenCL targets a range of heterogeneous computing platforms, but is initially focused on supporting GPUs.

In OpenCL's coprocessor model, a processing platform is comprised of one host and one or more compute devices, which are comprised of compute units, which are further comprised of processing elements. OpenCL programs are comprised of a host program and a set of kernels that run on devices, i.e., GPUs. The host program is written using two software layers: a platform layer and a runtime layer. The platform layer enables programs to query devices for platform-specific information and create contexts. A context is a scoping mechanism that allows the program to manage execution of kernels on devices. Within a context, the runtime layer allocates memory, compiles and creates kernels, adds kernels to a command queue, synchronizes commands and cleans up resources. Kernels that execute on the device are written using a modified version of C.

OpenCL host programs invoke kernels over an index space called an NDRange. Each instance of a kernel at a point in the NDRange is called a work-item; work-items are grouped into work-groups. Data parallelism is supported by work-items in a work-group executing a kernel together or multiple work-groups executing in parallel. See Figure 5 for an example of a vector add [26]. OpenCL requires all compute devices to support data parallelism, but some devices may also support task parallelism.

HIGH-LEVEL LANGUAGES

High-level languages (HLLs), such as Mathematica and MATLAB, are attractive to engineers and scientists for their high levels of abstractions and interactive programming environments. HLLs, however, suffer from decreased performance that often results in application runtimes of hours or days. This section discusses several technologies for supporting parallel computing in MATLAB without sacrificing the ease of use of the MATLAB language.

pMATLAB

MIT Lincoln Laboratory (MIT-LL) develops pMatlab [27], which (along with several other libraries that are discussed in the section "SIP Middleware") supports data parallelism using a programming construct called a map. To develop a pMatlab application, the user first writes a functionally correct serial MATLAB program. The program is then parallelized by adding maps. Maps are objects that describe how an array should be distributed, including the number of processors to distribute each dimension across and the distribution type (i.e., block, cyclic, and block-cyclic). Maps are passed into array constructors, e.g., `zeros()`, to create distributed arrays. The rest of the code that processes distributed arrays is unchanged. This is called a separation of concerns approach; the code for defining maps and code for implementing the algorithm are orthogonal. pMatlab distributes and communicates data without the user's knowledge. See Figure 6 for an example of allocating a distributed array [27].

Since pMatlab is written entirely in the MATLAB language, it runs on any architecture supported by MATLAB. This ranges from personal computers to clusters, containing either serial or homogeneous multicore processors.

```
// Kernel code that runs on GPU
kernel void sum (float a<>, float b<>, out float
c<>) {
  c = a + b;
}

// Host code that calls kernel code
int main() {
  float data[4] = {1.0, 2.0, 3.0, 4.0};
  float a<4>, c<4>;   // Allocate streams on GPU
  int i;

  streamRead(a, data);  // Copy from host to GPU
  sum(a, a, c);         // Call kernel on GPU
  streamWrite(c, data); // Copy from GPU to host
}
```

[FIG4] Example code for Brook+.

STAR-P

Star-P is a product from Interactive Supercomputing [28], initially targeted at MATLAB but that has expanded to other HLLs, e.g., Python. Both pMatlab and Star-P share a similar approach to supporting data parallelism. First, the user writes a functionally cor-

```
// GPU device code for vector sum kernel
__kernel void vec_add(__global const float *a,
                      __global const float *b,
                      __global       float *c){
  int gid = get_global_id(0);
  c[gid] = a[gid] + b[gid];
}

// Host code that calls vector sum kernel
int main() {
  // Create OpenCL context on GPU device and
  // create the command queue
  cl_context context =
    clCreateContextFromType(...);
  cl_cmd_queue cmd_queue =
    clCreateCommandQueue(context, ...);

  // Allocate memory (omitted for brevity)
  ...

  // Create and build the program
  cl_program program =
    clCreateProgramWithSource(context, ...);
  clBuildProgram(program, ...);

  // Create the kernel
  cl_kernel =
    clCreateKernel(program, "vec_add", NULL);

  // Set kernel args (omitted for brevity)
  ...

  // Run kernel and read output
  clEnqueueNDRangeKernel(cmd_queue, kernel,
                         ...);
  clEnqueueReadBuffer(context, ...);
}
```

[FIG5] Example code for OpenCL.

```
// Allocate distributed arrays
Nprocs = 4;
ABCmap = map([1 Nprocs], {}, 0:Nprocs-1);
A = zeros(1, 100, ABCmap);
B = rand(1, 100, ABCmap);
C = rand(1, 100, ABCmap);

// Do distributed computation
A(:, :) = B + pi * C;
```

[FIG6] Example code for pMatlab.

rect serial program. The user parallelizes the program by annotating MATLAB array constructors to create distributed arrays, known as a `ddense`. Star-P indicates which dimensions of the array to distribute by tagging the dimension arguments with `*p`. See Figure 7 for an example of allocating a `ddense` [29].

Star-P has a client-server based architecture, where the client is a MATLAB session running on the user's computer and the server is a cluster that runs the Star-P software. When a `ddense` constructor is called, the `ddense` object is allocated on the cluster and all operations on any `ddense` object execute on the cluster with high-performance parallel numerical libraries, such as Scalable LAPACK (ScaLAPACK).

MATLAB PARALLEL COMPUTING TOOLBOX

Parallel Computing Toolbox (PCT) is a parallel toolbox for MATLAB developed by The Mathworks, the developers of MATLAB [30]. PCT also supports data parallelism using distributed arrays, known as codistributed arrays. Again, the user first develops a functional serial program. The user parallelizes the program by marking the beginning and end of parallel sections of code with the keywords `spmd` and `end`, respectively. Arrays created inside `spmd` blocks are distributed across cores. A number of MATLAB functions have parallel implementations that are called when the function is invoked on a codistributed

```
// Allocate distributed arrays
a = rand(100*p);
b = rand(100*p);

// Do distributed computation
a * b;
```

[FIG7] Example code for Star-P.

```
spmd
  // Allocate distributed array
  A = rand(10000, 10000, codistributor());

  // Do distributed computation
  b = sum(A, 2);
end
```

[FIG8] Example code for PCT.

array. Many of these functions are based on ScaLAPACK. See Figure 8 for an example of allocating a codistributed array [31].

PCT also uses a client-server architecture, where the client is the user's MATLAB session. The client launches parallel code on workers which can execute either on multiple cores on the user's computer or on a cluster.

SIP MIDDLEWARE

A main challenge of building real-time embedded SIP systems is minimizing changes to existing software with the insertion of new hardware. Many systems are built on commercial hardware using vendor libraries for computation and communication. This leads to software that is hard to write, nonportable and nonscalable. Standards such as MPI and the Vector Signal and Image Processing Library (VSIPL) have improved portability, but not ease of development or scalability.

Middleware is a layer of software that provides a common interface to different hardware and hardware-specific software, e.g., vendor libraries. Middleware is key to isolating software from hardware changes. This section describes several real-time SIP middleware libraries. These libraries are unique in that they focus on programmer productivity, in addition to high performance and portability, by using maps.

PARALLEL VECTOR LIBRARY

The Parallel Vector Library (PVL) was developed at MIT-LL and has been used in a number of real-time embedded SIP systems [32]. Its goals are to simplify writing parallel SIP software and provide portability across hardware platforms, while maintaining high performance. It is a C++ library and contains a number of classes that hide the details of data and task parallelism.

PVL uses task maps and data maps to support task parallelism and data parallelism, respectively. Simply put, a map contains all the information necessary to describe how to distribute an object onto one or more processors.

Task parallelism is supported using tasks and conduits. A task encapsulates SPMD code that executes on one or more processors. A task map assigns a task to a set of processors at runtime. Task maps allow developers to write task code without needing to know which processors the task will run on. Conduits send data between tasks. Conduits hide the details of the underlying communication technology, e.g., MPI, shared memory, etc.

Tasks and conduits enable data flow patterns that are common in real-time SIP applications, e.g., pipelines, fork-joins, and round-robining. Tasks run on different sets of processors, executing different stages of a signal processing chain. Two consecutive tasks may distribute data across processors differently, e.g., data in the first task may be row-distributed and data in the second task may be column-distributed. The conduit connecting the tasks automatically redistributes the data from a row to column distribution.

Data parallelism is supported inside tasks with vector and matrix classes, which provide a high-level distributed array interface. Like pMatlab, the developer first writes functionally

correct serial code, then adds data maps to vector and matrix constructors to allocate distributed arrays. PVL supports a large number of SIP and linear algebra functions that operate on distributed objects. See Figure 9 for an example of allocating a distributed matrix.

PVL runs on homogeneous multicore processors, e.g., x86 and PowerPC, and is built on top of computation and communication libraries optimized for each supported hardware architecture, e.g., MPI and VSIPL. PVL users have also developed custom kernels wrapped in the PVL API.

```
// Allocate distributed matrix
vector<int> gridEls(2);
for (int i = 0; i < 2; i++){gridEls[i] = i;}
Map2d map(Grid2d(1, 2, gridEls),
          BlockDist(), BlockDist());
Matrix<float> mat("matrix", map, 4, 8);

mat = elMul(mat,2);  // Do parallel
                                computation
```

[FIG9] **Example code for PVL.**

VSIPL++

The High Performance Embedded Computing Software Initiative (HPEC-SI) is a consortium of industry, academia and government organizations that maintains the VSIPL standards. The first standard, VSIPL, is a C library for high-performance serial computation. VSIPL++ is the successor to VSIPL, a C++ library for both serial and parallel real-time SIP [33].

VSIPL++'s goal is to provide a standard API that supports productivity, performance and portability. To achieve these goals, VSIPL++ adopted concepts from PVL, notably data maps, and expanded on them, such as adding a tensor class to support three-dimensional arrays. See Figure 10 for an example of allocating a distributed matrix.

VSIPL++ uses standard C++ and, hence, can be built on any architecture with a C++ compiler. Like PVL, VSIPL++ can use existing, optimized computation and communication libraries. VSIPL++ runs on homogeneous multicore architectures, but there are ongoing efforts to extend VSIPL++ to other architectures. CodeSourcery has an implementation for the Cell and GPUs and Northeastern University is developing the VSIPL++ for Reconfigurable Computing (VForce) library, which allows VSIPL++ applications to use FPGAs as coprocessors [34].

PARALLEL VECTOR TILE OPTIMIZING LIBRARY

MIT-LL is currently developing the Parallel Vector Tile Optimizing Library (PVTOL) [35]. Its primary goal is to expand the parallel programming constructs in PVL and VSIPL++ to support both homogeneous and heterogeneous multicore architectures. PVTOL is built on top of VSIPL++ and uses the vector, matrix, and tensor data structures and data maps to support data parallelism.

PVTOL also inherits the task and conduit concept from PVL to enable task parallelism in VSIPL++. In PVL, however, a task is a collection of concurrent processes. On a cluster of single-core processors, each processor runs one of a task's processes. On a multicore processor, PVL runs an entire process on each core. Using threads instead of processes on a multicore processor is preferable, due to their lighter weight nature. Yet, multiple processes are still necessary for running on a cluster of multicore processors.

Historically, multiple programming models are used to write multithreaded, multiprocess applications. A common example of this is using a message-passing library, e.g., MPI, to distribute an application among multiple processors and a multithreaded library, e.g., OpenMP, to multithread each process. This approach is unwieldy, at best.

PVTOL supports task parallelism on multicore clusters by implementing multithreaded tasks [36]. PVTOL provides a single interface for scaling applications from a serial processor to a multicore processor to a cluster of multicore processors, by changing task maps.

As mentioned above, VSIPL++ targets homogeneous multicore architectures and work has been done to extend VSIPL++ to the Cell, GPUs, and FPGAs. These extensions of VSIPL++, however, invoke computation on coprocessors in a blocking manner, i.e., the host invokes a kernel on a coprocessor and waits for the coprocessor to complete. The ability to asynchronously launch computation onto coprocessors would allow both the host and coprocessor to execute simultaneously.

PVTOL tasks and conduits are being extended to various coprocessor architectures, including Cell, FPGAs, and GPUs. This will allow developers to asynchronously invoke computation and perform communication on host processors and coprocessors with the same interface. Tasks and conduits will be implemented for new coprocessor architectures as they emerge; having a consistent interface allows coprocessors in embedded systems to be replaced without changes to the application code. Figure 11 shows prototype code for running tasks on host and coprocessors.

As PVTOL develops and is used in MIT-LL systems, proposals will be made to HPEC-SI to include PVTOL concepts into future versions of the VSIPL++ standard.

TECHNOLOGY COMPARISON

```
// Allocate distributed matrix
typedef Map<Block_dist, Block_dist> map_type;
typedef Dense<2, scalar_f, row2_type, map_type>
  block_type;
typedef Matrix<scalar_f, block_type> view_type;
map_type map(1, 2);
view_type matrix(4, 8, map);

matrix = VSIP_IMPL_PI;
matrix *= 2;  // Do distributed computation
```

[FIG10] **Example code for VSIPL++.**

```
// Construct host and co-processor task maps
int hostprocs[] = {0}, coprocs[] = {0};
TaskMap<HOST>   hTaskMap(hostprocs);
TaskMap<COPROC> cTaskMap(coprocs);

// Construct host and co-processor tasks
Task<HostTask, HOST> hTask("Host", hTaskMap);
Task<CoprocTask, COPROC> cTask("Co-proc",
                               cTaskMap);

// Initialize and run the tasks
hTask.init(); cTask.init();
hTask.run(); cTask.run();

// Wait until the tasks complete
hTask.waitTillDone(); cTask.waitTillDone();
```

[FIG11] Example PVTOL code to launch tasks onto a host processor and coprocessor.

This section discusses tradeoffs between the technologies discussed in this article by comparing performance and productivity between different classes of technologies. Table 2 summarizes pros and cons of each technology.

The performance and productivity of multicore software technologies can be characterized in two ways. First is the level of abstraction of the programming language that the technology is based on. Second is the mechanism for parallelizing the application across multiple cores [37].

Programming languages can be classified by their level of abstraction. Assembly languages provide the highest performance, but are extremely difficult to use. Procedural languages, e.g., C, can produce excellent performance but with significantly less effort than assembly. Object-oriented languages, e.g., C++, can produce performance comparable to procedural languages, but often results in code that is more concise and modular. High-level languages, e.g., MATLAB, have significantly lower performance than other types of languages but require the least effort to use.

Parallel programming technologies can be classified by their communication mechanism. DMAs allow processors to directly access each other's memory and produce high performance but are hardware specific, are not applicable to all multiprocessor systems (e.g., clusters), and are difficult to use. Threads enable shared memory communication but limits systems to shared memory architectures and incur the complexity of synchronization.

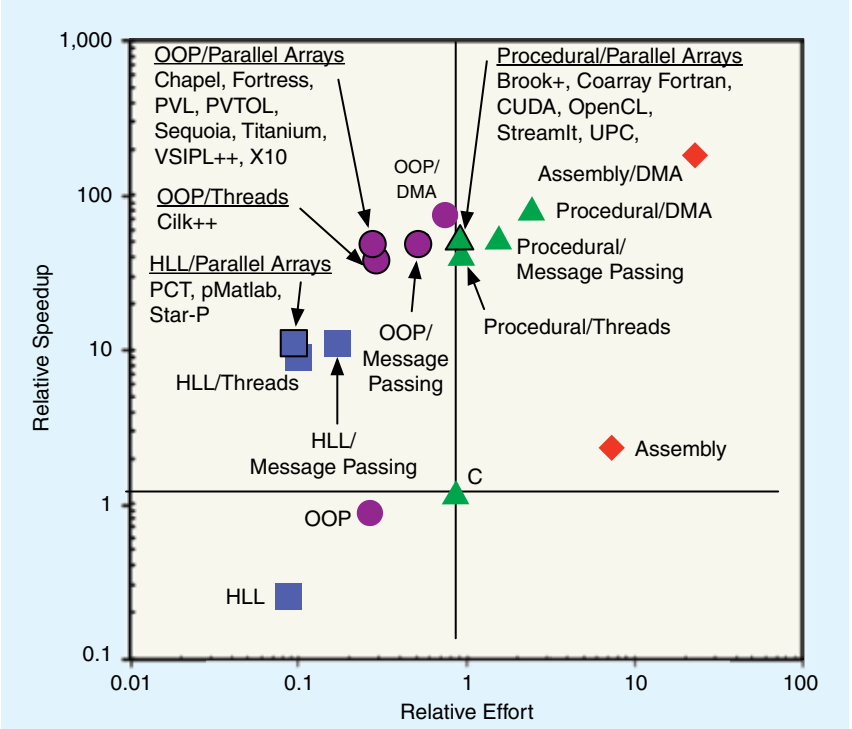
[TABLE 2] TRADEOFF COMPARISON OF MULTICORE SOFTWARE TECHNOLOGIES.

TECHNOLOGY	PROS	CONS
BROOK+	■ SUPPORT FOR BLAS, LAPACK, FFT	■ PROPRIETARY, SUPPORTS AMD GPUS ONLY ■ REQUIRES OTHER TECHNOLOGIES FOR MULTI-NODE COMMUNICATION
CHAPEL	■ DESIGNED FOR HIGH PRODUCTIVITY AND PERFORMANCE ■ OPEN SOURCE	■ REQUIRES LEARNING NEW LANGUAGE SYNTAX, TOOLS, ETC. ■ LANGUAGE UNDER DEVELOPMENT
*CILK++	■ BASED ON A FAMILIAR LANGUAGE (C++)	■ PROPRIETARY, COMMERCIAL (BUT FREE FOR OPEN SOURCE PROJECTS) ■ SUPPORTS SHARED MEMORY, X86 PROCESSORS ONLY
COARRAY FORTRAN	■ INDUSTRY STANDARD ■ BASED ON A FAMILIAR LANGUAGE (FORTRAN)	■ FOCUSES MOSTLY ON DATA PARALLELISM
CUDA	■ CURRENTLY, MOST POPULAR GPGPU TECHNOLOGY ■ SUPPORT FOR BLAS, FFT	■ PROPRIETARY, SUPPORTS NVIDIA GPUS ONLY ■ REQUIRES OTHER TECHNOLOGIES FOR MULTI-NODE COMMUNICATION
FORTRESS	■ DESIGNED FOR HIGH PRODUCTIVITY AND PERFORMANCE ■ OPEN SOURCE	■ REQUIRES LEARNING NEW LANGUAGE SYNTAX, TOOLS, ETC. ■ LANGUAGE UNDER DEVELOPMENT
OPENCL	■ INDUSTRY STANDARD ■ EXTENSIBLE TO MANY COPROCESSOR ARCHITECTURES	■ LOW-LEVEL TECHNOLOGY TARGETED AT EXPERT PROGRAMMERS
PCT	■ HIGH PROGRAMMER PRODUCTIVITY ■ PROVIDES SUPPORT FOR PARALLELISM BUILT INTO MATLAB	■ NOT SUITED FOR MANY PERFORMANCE-CRITICAL APPLICATIONS
PMATLAB	■ HIGH PROGRAMMER PRODUCTIVITY ■ OPEN SOURCE	■ NOT SUITED FOR MANY PERFORMANCE-CRITICAL APPLICATIONS
PVL	■ DESIGNED FOR REAL-TIME SIP APPLICATIONS ■ HIGH LEVEL DATA STRUCTURES	■ RESEARCH PROTOTYPE, NOT AVAILABLE FOR GENERAL USE (BUT CONCEPTS ADOPTED BY VSIPL++)
PVTOL	■ DESIGNED FOR REAL-TIME SIP APPLICATIONS ■ HIGH LEVEL DATA STRUCTURES ■ EXTENSIBLE TO MANY COPROCESSOR ARCHITECTURES	■ RESEARCH PROTOTYPE, NOT AVAILABLE FOR GENERAL USE (BUT CONCEPTS WILL BE PROPOSED FOR ADDITION TO VSIPL++)
SEQUOIA	■ EXTENSIBLE TO MANY COPROCESSOR ARCHITECTURES	■ RESEARCH PROTOTYPE, CURRENTLY IN ALPHA
STAR-P	■ HIGH PROGRAMMER PRODUCTIVITY	■ PROPRIETARY, COMMERCIAL ■ NOT SUITED FOR MANY PERFORMANCE-CRITICAL APPLICATIONS
STREAMIT	■ DESIGNED FOR REAL-TIME SIP APPLICATIONS	■ RESEARCH PROTOTYPE ■ REQUIRES LEARNING NEW LANGUAGE SYNTAX, TOOLS, ETC.
TITANIUM	■ BASED ON A FAMILIAR LANGUAGE (JAVA)	■ FOCUSES MOSTLY ON DATA PARALLELISM
UPC	■ INDUSTRY STANDARD ■ BASED ON A FAMILIAR LANGUAGE (C)	■ FOCUSES MOSTLY ON DATA PARALLELISM
VSIPL++	■ INDUSTRY STANDARD ■ DESIGNED FOR REAL-TIME SIP APPLICATIONS	■ FOCUSES MOSTLY ON DATA PARALLELISM
X10	■ DESIGNED FOR HIGH PRODUCTIVITY AND PERFORMANCE ■ BASED ON A FAMILIAR LANGUAGE (JAVA)	■ LANGUAGE UNDER DEVELOPMENT

*Cilk Arts was recently acquired by Intel.

Message passing supports a wide range of systems, but parallelizing applications can still require significant effort. The primary difficulties of parallel programming are partitioning arrays across processors and redistributing arrays between processors. Historically, this was the programmer's responsibility and highly error-prone. Distributed, or parallel, arrays hide this complexity in a concise interface, improving productivity. Consequently, nearly every new multicore software technology includes distributed arrays or constructs that can be used to implement distributed arrays.

Quantitative estimates for performance efficiency and coding effort for each class of programming languages and parallel programming technologies are calculated in [37]. By combining efficiency and effort estimates, relative speedup and effort for an application implemented with each technology on a hypothetical 100-core system are compared to a serial C implementation, shown in Figure 12 [37].



[FIG12] Estimated speedup versus effort relative to serial C on a hypothetical 100-core system.

Figure 12 is divided into quadrants. The upper and lower halves indicate higher and lower performance than serial C, respectively, and the left and right halves indicate less and greater effort than serial C, respectively. Historically, parallel programming technologies resided in the upper right quadrant, i.e., higher performance at the cost of higher effort. For example, procedural languages using message passing, e.g., MPI, require significantly more effort than serial code. Procedural languages with parallel arrays reduce the effort needed to achieve high performance with slightly more effort than serial programming. As mentioned above, object-oriented languages can produce code that is more concise than procedural code with comparable performance. Hence, object-oriented languages with parallel arrays can produce parallel applications with comparable performance to procedural languages with parallel arrays, but with less effort. Developers can quickly implement algorithms using HLLs, but verification can take a long time due to their poor performance. Parallel arrays can significantly improve the performance of applications with little effort.

Figure 12 indicates which categories each technology discussed in this article belongs to. Note while these are estimates, they give a sense of the relative performance and productivity between different technologies.

SUMMARY

In this article, we described several categories of software technologies for signal and image processing on multicore architectures: parallel programming languages, general-purpose computation on GPUs, high-level languages, and middleware libraries. Technologies for each category were described, including the various capabilities for multicore programming each technology supports. Finally, technologies were compared at a high level for performance efficiency and programmer productivity.

ACKNOWLEDGMENTS

The authors would like to thank Nadya Bliss, Jeremy Kepner, and the anonymous reviewers for their valuable comments.

AUTHORS

Hahn Kim (hgk@ll.mit.edu) is an associate staff member in the Embedded Digital Systems Group at Massachusetts Institute of Technology Lincoln Laboratory. He received a B.S.E degree in computer engineering in 2001 and an M.S.E. degree in computer science and engineering in 2003 from the University of Michigan. He joined Lincoln Laboratory in 2003. His research interests include developing high-performance parallel software technologies for real-time embedded systems. His work includes working on development of the pMatlab library and the Parallel Vector Tile Optimizing Library, supporting pMatlab developers using the Lincoln's LL Grid cluster, and developing real-time software for ISR systems.

Robert Bond (rbond@ll.mit.edu) leads the Embedded Digital Systems group at Massachusetts Institute of Technology Lincoln Laboratory. He earned a B.S. degree (honors) in physics from Queen's University, Ontario, Canada in 1978. He joined Lincoln Laboratory in 1987. His research interests include developing high-performance embedded processors, advanced signal processing technology, and novel embedded middleware architec-

tures. He has managed the development of a massively parallel airborne processor for space-time adaptive processing and radar signal processing and lead the Parallel Vector Library development team. He received the 2003 Lincoln Laboratory Technical Excellence Award.

REFERENCES

[1] W. Gropp, E. Lusk, and R. Thakur, *Using MPI-2*. Cambridge, MA: MIT Press, Nov. 1999.

[2] B. Chapman, G. Jost, and R. van der Pas, *Using OpenMP*. Cambridge, MA: MIT Press, Oct. 2007.

[3] P. Luszczek, J. J. Dongarra, D. Koester, R. Rabenseifner, B. Lucas, J. Kepner, J. McCalpin, D. Bailey, and D. Takahashi, "Introduction to the HPC challenge benchmark suite," Lawrence Berkeley Nat. Lab., Berkeley, CA, Paper LBNL-57493, 2005.

[4] R. Haney, J. Kepner, and T. Meuse, "The HPEC challenge benchmark suite," in *Proc. High Performance Embedded Computing Workshop 2006*, Lexington, MA, Sept. 2006.

[5] J. A. Kahle, M. N. Day, H. P. Hofstee, C. R. Johns, T. R. Maeurer, and D. Shippy, "Introduction to the cell multiprocessor," *IBM J. Res. Develop.*, vol. 49, no. 4/5, pp. 589–604, July 2005.

[6] M. J. Flynn, "Some computer organizations and their effectiveness," *IEEE Trans. Comput.*, vol. C-21, pp. 948–960, Sept. 1972.

[7] A. Silberschatz, P. B. Galvin, and G. Gagne, *Operating System Concepts*, 8th ed. Hoboken, NJ: Wiley, 2009, pp. 153–182.

[8] T. El-Ghazawi, W. Carlson, T. Sterling, and K. Yelick, *UPC: Distributed Shared Memory Programming*. Hoboken, NJ: Wiley-Interscience, 2005.

[9] T. El-Ghazawi, P. Merkey, and S. Seidel, "SC05 tutorial—High performance parallel programming with unified parallel C (UPC)," presented at *Proc. Supercomputing 2005*, Seattle, WA, Nov. 2005.

[10] K. Fatahalian, T. J. Knight, M. Houston, M. Erez, D. R. Horn, L. Leem, J. Y. Park, M. Ren, A. Aiken, W. J. Dally, and P. Hanrahan, "Sequoia: Programming the memory hierarchy," in *Proc. Supercomputing 2006*, Tampa Bay, FL, Nov. 2006.

[11] Sequoia Web site [Online]. Available: http://sequoia.stanford.edu

[12] R. W. Numrich and J. Reid, "Co-arrays in the next Fortran standard," *ACM SIGPLAN Fortran Forum*, vol. 24, no. 2, pp. 4–17, Aug. 2005.

[13] K. A. Yelick, L. Semenzato, G. Pike, C. Miyamoto, B. Liblit, A. Krishnamurthy, P. N. Hilfinger, S. L. Graham, D. Gay, P. Colella, and A. Aiken, "Titanium: A high-performance Java dialect," *Concurrency Pract. Exp.*, vol. 10, no. 11/13, pp. 825–836, Sept./Nov. 1998.

[14] *Cilk++ Programmer's Guide*, Cilk Arts, Lexington, MA, Mar. 16, 2009.

[15] R. D. Blumofe, C. F. Jeorg, B. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou, "Cilk: An efficient multithreaded runtime system," in *Proc. 5th ACM SIGPLAN Symp. Principles and Practices of Parallel Programming*, Santa Barbara, CA, July 1995, pp. 207–216.

[16] W. Thies, M. Karczmarek, and S. Amarsinghe, "StreamIt: A language for streaming applications," in *Proc. 11th Int. Conf. Compiler Construction*, Grenoble, France, Apr. 2002, pp. 179–196.

[17] B. L. Chamberlain, D. Callahan, and H. P. Zima, "Parallel programmability and the Chapel language," *Int. J. High Perform. Comput. Applicat.*, vol. 21, no. 3, pp. 291–312, 2007.

[18] E. Allen, D. Chase, C. Flood, V. Luchangco, J. Maessen, S. Ryu, and G. L. Steele, Jr., "Project Fortress: A multicore language for multicore processors," *Linux Mag.*, pp. 38–43, Sept. 2007.

[19] P. Charles, C. Donowa, K. Ebcioglu, C. Grothoff, A. Kielstra, C. von Praun, V. Saraswat, and V. Sarkar, "X10: An object-oriented approach to non-uniform cluster computing," in *Proc. 20th Conf. Object Oriented Programming Systems Languages and Applications*, San Diego, CA, 2005, pp. 519–538.

[20] *NVIDIA CUDA Programming Guide Version 2.2.1*, NVIDIA, Santa Clara, CA, May 26, 2009.

[21] M. Garland, "CUDA tutorial—Parallel computing in CUDA," presented at *Proc. Architectural Support for Programming Languages and Operating Systems 2008*, Seattle, WA, Mar. 2008.

[22] "ATI stream computing—Technical overview," AMD Web site [Online]. Available: http://developer.amd.com/gpu_assets/Stream_Computing_Overview.pdf

[23] I. Buck, T. Foley, D. Horn, J. Sugerman, K. Fatahalian, M. Houston, and P. Hanrahan, "Brook for GPUs: Stream computing on graphics hardware," *ACM Trans. Graph.*, vol. 23, no. 3, pp. 777–786, Aug. 2004.

[24] B. Zheng, D. Gladding, and M. Villmow, "Tutorial—Building a high level language compiler for GPGPU," presented at *Proc. Programming Language Design and Implementation 2008*, Tucson, AZ, June 2008.

[25] A. Munshi, "The OpenCL specification version 1.0," Khronos Group, Beaverton, OR, May 16, 2009.

[26] N. Trevett, "OpenCL: The open standard for heterogeneous parallel programming," in *Proc. SIGGRAPH Asia 2008*, Singapore, Dec. 2008.

[27] N. T. Bliss and J. Kepner, "pMatlab parallel MATLAB library," *Int. J. High Perform. Comput. Applicat.*, vol. 21, no. 3, pp. 336–359, 2007.

[28] R. Choy and A. Edelman, "Parallel MATLAB: Doing it right." *Proc. IEEE*, vol. 93, no. 2, pp. 331–341 [Online]. Available: http://ieeexplore.ieee.org/xpls/abs_all.jsp?isnumber=30187&arnumber=1386655&count=21&index=8

[29] Interactive supercomputing Web site [Online]. Available: http://www.interactivesupercomputing.com

[30] G. Sharma and J. Martin, "MATLAB: A language for parallel computing," *Int. J. Parallel Program.*, vol. 37, no. 1, pp. 3–36, Oct. 2008.

[31] MATLAB parallel computing toolbox Web site [Online]. Available: http://www.mathworks.com/products/parallel-computing

[32] J. Kepner and J. Lebak, "Software technologies for high-performance parallel signal processing," *Lincoln Lab. J.*, vol. 14, no. 2, pp. 181–198, 2003.

[33] J. Lebak, J. Kepner, H. Hoffman, and E. Rutledge, "Parallel VSIPL++: An open standard software library for high-performance parallel signal processing," *Proc. IEEE*, vol. 93, no. 2, pp. 313–330, Feb. 2005.

[34] N. Moore, A. Conti, M. Leeser, and L. S. King, "Vforce: An extensible framework for reconfigurable supercomputing," *IEEE Comput. Mag.*, vol. 40, no. 3, pp. 39–49, Mar. 2007.

[35] H. Kim, E. Rutledge, S. Sacco, S. Mohindra, M. Marzilli, J. Kepner, R. Haney, J. Daly, and N. Bliss, "PVTOL: Providing productivity, performance and portability to DoD signal processing applications on multicore processors," in *Proc. High Performance Computing Modernization Program Users Group Conf. 2008*, Seattle, WA, July 2008, pp. 327–333.

[36] S. Mohindra, J. Daly, R. Haney, and G. Schrader, "Task and conduit framework for multi-core systems," in *Proc. High Performance Computing Modernization Program Users Group Conf. 2008*, Seattle, WA, July 2008, pp. 506–513.

[37] J. Kepner, T. Meuse, and G. Schrader, "Performance metrics and software architecture," in *High Performance Embedded Computing Handbook: A Systems Perspective*, D. R. Martinez, R. A. Bond, and M. M. Vai, Eds. Boca Raton, FL: CRC, 2008, pp. 303–334.

[SP]

CALLOUTS

NEW TECHNOLOGIES ARE NEEDED TO MAKE MULTICORE PROCESSORS ACCESSIBLE TO A LARGER COMMUNITY.

MULTICORE SOFTWARE TECHNOLOGIES CAN BE IMPLEMENTED AS LANGUAGES OR LIBRARIES.

THE VAST MAJORITY OF PROGRAMMING LANGUAGES ARE DESIGNED FOR SERIAL COMPUTATION.

IN RECENT YEARS, MORE GENERAL PROGRAMMING TECHNOLOGIES FOR GPUS HAVE APPEARED, ELIMINATING THE NEED FOR A BACKGROUND IN GRAPHICS PROGRAMMING.