SIMD: An Additional Pattern for PLPP (Pattern Language for Parallel Programming)

Berna L. Massingill
Department of Computer Science
Trinity University
San Antonio, TX
bmassing@trinity.edu

Timothy G. Mattson
Microprocessor Technology Laboratory
Intel Corporation
DuPont, WA
timothy.g.mattson@intel.com

Beverly A. Sanders
Department of Computer and Information Science and Engineering
University of Florida
Gainesville, FL
sanders@cise.ufl.edu

ABSTRACT

Recent trends in hardware, such as IBM's Cell Broadband Engine and GPUs that can be used for general-purpose computing, have made widely available systems for which a SIMD (Single Instruction, Multiple Data) style of data-parallel programming is appropriate. This paper presents a pattern to help software developers construct parallel programs for environments that support this style of data parallelism. In this approach, the program is viewed as a single thread of control, with implicitly parallel updates to data. This pattern is a new addition to the Pattern Language for Parallel Programming (PLPP) presented in our previous work [18, 19].

Categories and Subject Descriptors

D.1.3 [**Concurrent Programming**]: Parallel Programming.

General Terms

Algorithms, Design.

Keywords

Parallel programming, design patterns, pattern language, SIMD (Single Instruction, Multiple Data).

1. INTRODUCTION

Over the course of several years we have developed a pattern language for developing parallel application programs. The patterns were developed in a series of PLoP papers [12–16] and published in [19]. This pattern language, which we call PLPP (Pattern Language for Parallel Programming), embodies a methodology in which we develop a parallel application by starting with a good understanding of the problem and then working through a sequence of patterns, ending up with code. One of our long-term goals for this work is to review the pattern language and revise it as needed. A first step in this direction was the development of an additional pattern, *Reengineering for Parallelism* [17, 18] discussing how to apply the pattern language to existing applications (i.e., when the starting point is not simply a good understanding of the problem, but existing sequential code). In this paper, we present another proposed addition to PLPP, to address a class of target programming environments not addressed well by the current patterns.

1.1 Data-Parallel Programming and SIMD, Then and Now

From the early days of parallel computing, making the programmer's job easier has been a key concern. A popular idea in the late 1980s and early 1990s was to think of the computation in terms of a single stream of instructions operating on multiple elements of data, with concurrency emerging from the fact that the single instruction was operating on multiple data elements at the same time. This is the so-called *SIMD* (Single Instruction, Multiple Data) model. Hardware was built according to this model, the most famous example being the Connection Machine [6]. A natural and more general programming model for such hardware is *data parallelism*, in which concurrency is based on updating elements or sections of large data structures (such as arrays) in parallel. Hillis and Steele wrote an early and influential paper [7] on this style of programming. The SIMD model fell out of favor in the hardware world, supplanted by various forms of *MIMD* (Multiple Instruction, Multiple Data).[1] The idea of data-parallel programming, however, survived in a number of forms, including both explicitly data-parallel programming environments such as HPF [8] and strategies for finding and exploiting concurrency by focusing on the data rather than the computation — in PLPP terms, by starting with a *Data Decomposition* rather than a *Task Decomposition*.

Recent trends in hardware have renewed interest in more explicitly data-parallel programming in general and the SIMD model in particular: Several mainstream hardware

[1]The terminology — SIMD versus MIMD — is based on a taxonomy defined in an early paper by Flynn [5].

platforms involve arrays of special-purpose processors, notably programmable GPUs (such as the NVIDIA GeForce) and IBM's Cell Broadband Engine (used in the PlayStation 3). Such hardware can be programmed at a fairly low level (e.g., using NVIDIA's CUDA [21]), but programmers may find it easier to work at a higher level of abstraction and depend on a compiler and/or runtime library to map appropriate abstractions onto the details of particular hardware platforms. One approach to doing this is *stream processing*, featuring collections of data (called streams) to be operated on in parallel by sequences of instructions (called kernels); examples include Brook [4] and the RapidMind Development Platform [23]. Another approach focuses on defining data types representing collections of data to be operated on in parallel and operations on these data types; examples include the PeakStream Platform [22] and Microsoft's Accelerator [1]. An advantage of these higher-level approaches is greater potential for portability.

1.2 Data-Parallel Programming and PLPP

PLPP is organized into four *design spaces* (groups of patterns) that correspond to the four phases of the development methodology it embodies. (Appendix A gives an overview of PLPP and the four design spaces, for readers unfamiliar with it.)

The first of these design spaces, *Finding Concurrency*, is concerned with identifying and analyzing exploitable concurrency. Of the two starting points, one (*Data Decomposition*) is a good fit for the kinds of hardware and programming environments just described.

The next design space, *Algorithm Structure*, describes strategies for exploiting the concurrency identified in the first space. Several of the patterns are again good fits: *Task Parallelism* (despite its name), *Geometric Decomposition*, and *Recursive Data*.

The next design space, *Supporting Structures*, includes patterns that represent different approaches to structuring programs, such as *SPMD* (Single Program, Multiple Data, as exemplified by most MPI programs) and *Loop Parallelism* (as exemplified by most OpenMP programs). None of these patterns, however, really fits the platforms (hardware and programming environments) described above. We believe now that to fully address these platforms we will need to address three broad categories of data-parallel programming:

- Data-parallel programming in which different units of execution (UEs)[2] operate concurrently on relatively large sections of data, possibly performing slightly different operations — for example, to deal with boundary conditions.

- Data-parallel programming in which different UEs operate concurrently on what are typically smaller sections of data, perhaps single elements, with each UE executing the same computational kernel, but not necessarily in lockstep.

- Data-parallel programming in which different UEs operate concurrently on what are typically smaller sections of data, perhaps single elements, with all UEs executing the same sequence of instructions in lockstep.

[2]Generic term used in PLPP for processes or threads.

The first of these categories is addressed by our existing *SPMD* pattern. The other two categories are not as well addressed; [19] does include a brief discussion of "the *SIMD* pattern" that fits the last category, but it is not a full-fledged pattern.

In this paper we present a complete *SIMD* pattern to address the last of the three categories described above. Future work will address the remaining category.

2. THE *SIMD* PATTERN

Problem

How do you exploit concurrency defined as parallel updates to data without specifying in detail how that concurrent execution is implemented?

Context

Programmers tend to think of a program's execution as a sequence of instructions. The program's source code may include many branches with multiple pathways through the code, but only one path at a time is active. When concurrent execution is needed, however, this way of thinking about the program's execution is usually inadequate: In many parallel algorithms, different pathways through a program may be active at the same time, or there may be multiple UEs[3] executing the same code at the same time but more or less independently. This parallel mindset has proved very difficult for programmers to master.

It would be valuable if we could define a high-level abstraction that allows programmers to think in terms of a single thread of control, with the concurrency being hidden inside lower-level primitives. Defining a high-level abstraction for the general case has proved difficult if not impossible. However, there is a subset of problems for which it is easier: those where the concurrency is defined in terms of largely independent updates to the elements of the key data structures in the problem. For such problems, the *data-parallel model*, in which the program appears to have a single thread of control, but operations on these key data structures are implicitly concurrent, seems reasonable.

This is a particularly attractive model for some hardware platforms, particularly those involving groups of processing elements that can readily be made to operate in a SIMD (Single Instruction, Multiple Data) style. Several programming environments for such platforms are based on this model; they typically provide predefined datatypes suitable for these key data structures and ways of operating on them in parallel.

Forces

All the program structure patterns in PLPP share a common set of forces, described in some detail in [19] and summarized here, with discussion of how they apply to the *SIMD* pattern in particular.

- **Clarity of abstraction.** Is the parallel algorithm clearly apparent from the source code?

  Code that allows the reader to see the details of the algorithm with little mental effort is always important

[3]Units of execution — generic term for processes or threads.

for writing correct code, but is particularly essential for parallel programs.

It is particularly important if the target hardware platform is one that may be difficult to program at a low level (as is the case with some hardware platforms involving special-purpose processors).

- **Scalability.** How many processors can the parallel program effectively utilize?

  Scalability of a program is restricted by three factors: the amount of concurrency available in the algorithm, the fraction of the runtime spent doing inherently serial work, and the parallel overhead of the algorithm.

  The importance of scalability relative to other goals may depend in part on the target hardware platform (i.e., what constraints exist on the maximum number of processing elements). However, it is always important to be sure the amount of exploitable concurrency is large enough, the serial fraction of the code small enough, and the overhead small enough to allow the algorithm to make good use of whatever processing elements are available. Notice that for target hardware platforms where the data-parallel operations are executed on a coprocessor (such as a GPU), overhead can include the cost of moving or copying data between host memory and the coprocessor.

- **Efficiency.** How close does the program come to fully utilizing the resources of the parallel computer?

  Quantitatively, efficiency is typically measured by comparing the time taken by the best sequential algorithm to the time taken by the parallel algorithm multiplied by the number of processing elements.

- **Maintainability.** Is the program easy to debug, verify, and modify?

  Programs that are difficult to understand are usually also difficult to get running in the first place, and to modify throughout their lifetimes. Parallel programs are notoriously difficult to debug, especially if programmers cannot understand them without reasoning about multiple independent threads of control interacting in different ways.

- **Environmental affinity.** Is the program well aligned with the programming environment and hardware of choice?

  Whenever feasible, program designs should be portable to as many platforms as possible, but in any case it should be possible to implement them efficiently on likely target platforms.

- **Sequential equivalence.** Where appropriate, does a program produce equivalent results when run with many UEs[4] as with one? If not equivalent, is the relationship between them clear?

  It is highly desirable that the results of an execution of a parallel program be the same regardless of the number of processing elements used. This is not always possible, because changes in the order of floating-point operations can produce small (or not so small, if the algorithm is not well-conditioned) changes in results. However, it is highly desirable, since it allows us to reason about correctness and do most testing on a single-process version of the program.

[4]Units of execution, the term we use in PLPP as a generic way of referring to either processes or threads.

Solution

The *SIMD* pattern is based in general on the data-parallel model described in the Context section, and in particular on the SIMD style. The pattern is based on two key ideas: (1) organizing the data relevant to a problem into regular iterative data structures such as arrays, which we will call *parallel data structures*, and (2) defining the computation in terms of a sequence of parallel updates to these parallel data structures. Often in applying these two ideas one must also introduce parallel data structures to hold intermediate results.

Notice that the meaning of a data-parallel program can be understood in terms of a sequence of instructions, in much the same way as a sequential program. The concurrency is hidden inside the updates to the elements of the parallel data structures. This is valuable to a programmer trying to understand a new program and also in debugging.

Applying this pattern is often best done using the strategy of incremental parallelism described in the PLPP *Loop Parallelism* and *Reengineering for Parallelism* patterns. Consider a sequential program that we wish to execute on parallel hardware, such as a GPU or a multi-core processor. An effective and practical approach to incrementally transforming the code into a parallel program is as follows.

- **Identify exploitable concurrency.** Find the bottlenecks in the program and the data structures central to the computation. (Profiling may help with this.) In cases where this pattern is effective, these will often appear as computations organized as loops over all elements of an array. You may need to restructure the problem to recast it into this form.

- **Refactor to eliminate obstacles to parallelism.** Eliminate loop-carried dependencies (as described in the PLPP *Loop Parallelism* patterns), and express each computation as a simple loop over all elements of an array, where all the iterations are independent.

- **Revise to use data-parallel operations.** Recast the loop body as a data-parallel operation on the array using the data-parallel operations provided by the target programming environment. (If the environment provides both a `foreach` construct and whole-array operations, consider making this transformation in two stages: First replace the loop with a `foreach` and verify that the result is correct. Then replace the `foreach` with a whole-array operation or operations.)

- **Allocate elements to threads (if needed).** Allocate elements of the array to threads. In some environments, this task is taken care of by the compiler/runtime system combination. In others, it is explicitly managed by the programmer, who needs to be aware of the resources of the specific hardware.

- **Tune for performance.** Build the program and evaluate the effectiveness of the solution so far. Is there enough work done on each element to justify the overhead? In a multi-core environment, the overhead comes from creating and managing the threads. There should be enough work per thread to make this worthwhile. If a coprocessor is being used, the overhead tends to be dominated by the cost of transferring data between the host and coprocessor memory. In addition, coprocessors may not have caches, so the number of memory accesses should be minimized.

The following short examples are typical of the kinds of transformations that can be helpful in applying this pattern. These examples, like the longer code examples later in the pattern, are in Fortran 95. A short summary of its relevant features appears in the section "Example Programming Environment (Fortran 95)" below.

- Replacing loops with `foreach` (`forall`) constructs and then array operations is fairly straightforward.

  For example, suppose you start with arrays `x`, `y`, and `z` with indices ranging from 1 to `N` and a calculation of the form

  ```
  do i = 1, N
    z(i) = a * x(i) + y(i)
  ```

  Since the iterations of the loop are independent, this can be readily transformed to use a `forall` as follows

  ```
  forall (i = 1 : N)
    z(i) = a * x(i) + y(i)
  end forall
  ```

  and further transformed to use array operations as follows

  ```
  z = a * x + y;
  ```

- The PLPP *Algorithm Structure* pattern *Task Parallelism* discusses several techniques for eliminating loop-carried dependencies, most involving replacing a single variable with one copy per UE. These same techniques can be applied here, replacing single variables with arrays.

  For example, suppose you start with a calculation of the form

  ```
  do i = 1, numSteps
    x = x + f(i)
  ```

  This can be transformed by introducing an array `y` to hold intermediate results which are later combined, giving

  ```
  forall (i = 1 : numSteps)
    y(i) = f(i)
  end forall
  x = sum(y)    ! library function,
                ! returns sum of elements of array
  ```

Performance Considerations

In order for this pattern to be effective, the following two conditions must be met.

- The arrays, or the work per element, must be large enough to compensate for overheads inherent in managing the parallel execution.
- A large enough fraction of the computation must fit this pattern to keep the serial fraction low; otherwise overall scalability will be limited, as described by Amdahl's law.

The pattern also must be appropriate for the hardware target. A common use of this pattern is for general-purpose programming of a GPU. In this case, the GPU serves as a coprocessor. This adds a number of extra costs:

- The data must be marshaled into a buffer for movement to the coprocessor.
- The data must be moved from the CPU address space into the coprocessor address space.
- The computations must be managed to satisfy any synchronization constraints between the coprocessor and the CPU.
- The results must be moved from the coprocessor address space back to the CPU address space.

The time required for these steps must be small compared to the runtime on the coprocessors if this pattern is to be effective.

Programming Environments

This pattern works best when the target programming environment makes it easy to express parallel data structures and operations. Such environments usually provide the following high-level constructs.

- Element-wise basic arithmetic operations.
- Common collective operations such as scatter, gather, reduction, and prefix scan.
- A `foreach` command to handle more general element-wise updates.

Examples of such environments include the following.

- Fortran 95 [9, 20]. Many of the array operations in Fortran 90 were designed to allow compilers to exploit vector-processing or other parallel hardware if available; Fortran 95 added a `forall` construct similar to the `foreach` described above.

- High Performance Fortran (HPF) [8]. HPF is a set of extensions to Fortran 90 for data-parallel computing. It includes some of the additions of Fortran 95 (e.g., `forall`), compiler directives for indicating how arrays should be distributed among UEs, and additional library functions.

- The PeakStream Platform [22]. This platform provides C/C++ library classes and functions to define and operate on parallel arrays.

- Microsoft's Accelerator [1]. This system, aimed at facilitating general-purpose use of GPUs, provides C# library classes to define and operate on parallel arrays.

Example Programming Environment (Fortran 95)

In this section we briefly describe the features of Fortran 95 that make it a data-parallel programming environment. This is the environment we chose for the code examples in this pattern, largely because there are free compilers readily available for common platforms, so the code can be compiled and its correctness verified without requiring special software or hardware.

- Whole-array operations. As an example of operations on arrays, consider the following code fragment to compute the product of a scalar value `a` and an array `x`, add it to an array `y`, and assign the result to array `z`. Indices for each array range from 1 to `N`, where `N` is an integer defined elsewhere.

```
real a
real x(1:N), y(1:N), z(1:N)
z = a * x + y
```

- A `forall` construct similar to the `foreach` mentioned earlier. A Fortran 95 `forall` consists of a range of index values and a "loop body" of statements to be executed for each index in the range. Assignment statements in the loop body work as parallel assignments, with all right-hand sides evaluated before values are assigned to any left-hand sides. We could rewrite the preceding example using this construct as follows:

```
real a
real x(1:N), y(1:N), z(1:N)
forall (i = 1 : N)
  z(i) = a * x(i) + y(i)
end forall
```

  This construct also includes an optional mask specifying that the loop body is to be executed only for indices that match a particular criterion. For example, the following code changes all negative elements of array `x` to zero.

```
real x(1:N)
forall (i = 1 : N, x(i) < 0)
  x(i) = 0.0
end forall
```

Examples

As examples of this pattern in action, we will consider in some detail the following, and then briefly describe other examples and known uses.

- Computing π using numerical integration. (This example is used in several patterns of PLPP.)
- Simplified modeling of heat diffusion in a one-dimensional pipe, as an example of a mesh computation. (This example is also used in several patterns of PLPP.)
- The Black-Scholes method for option pricing, a widely used example of Monte Carlo methods.
- The so-called *scan* problem, a generalized version of computing partial sums of an array.

Numerical Integration

Consider the problem of estimating the value of π using the following equation.

$$\pi = \int_0^1 \frac{4}{1+x^2}\,dx$$

We can use the midpoint rule to numerically approximate the integral. The idea is to approximate the area under a curve using a series of rectangles, as shown in Fig. 1.

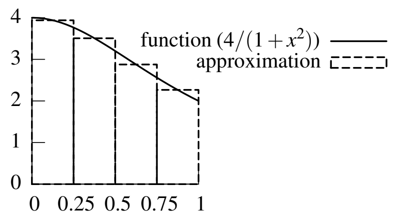

Figure 1: Approximating the area under a curve with a series of rectangles. Notice that the width of each rectangle is the width of the whole area divided by the number of rectangles, and the height of each rectangle is the value of the function at the *x* coordinate in the middle of the rectangle.

A program to carry this calculation out on a single processor is shown in Fig. 2. To keep the code simple, we hard-code the number of rectangles (steps). The variable `sum` is initialized to zero and the step size is computed as the range in `x` (equal to 1.0 in this case) divided by the number of steps. The area of each rectangle is the width (the step size) times the height (the value of the integrand at the center of the interval), as shown in Fig. 1. Since the width is a constant, we pull it out of the summation and multiply the sum of the rectangle heights by the step size, `step`, to get our estimate of the definite integral.

```
integer NUM_STEPS
parameter (NUM_STEPS=1000000)
integer i
real step, sum, x, pi

step = 1.0/(NUM_STEPS)
sum = 0.0
do i = 0, NUM_STEPS-1
    x = (i+0.5)*step
    sum = sum + 4.0/(1.0+x*x)
end do
pi = step*sum

print *, pi
```

Figure 2: Sequential Fortran 95 program to carry out a midpoint rule integration to solve $\int_0^1 \frac{4}{1+x^2}\,dx$.

Since we are beginning with sequential code, we can work through the steps described previously for a strategy based on making changes incrementally, as follows.

Identify exploitable concurrency.

The only source of exploitable concurrency is the iterations of the loop, and they are almost independent and therefore a good fit for the simplest of the PLPP *Algorithm Structure* patterns, *Task Parallelism*.

Refactor to eliminate obstacles to parallelism.

As discussed in *Task Parallelism*, the single loop-carried dependency is the summing of intermediate values into `sum`, which we can eliminate by having each thread of control compute its own partial sum and combining them at the end. We can do this in a way that fits with the data-parallel model by simply replacing temporary variable `x` with an array to hold the intermediate values that will be summed. We can then compute the intermediate results by performing operations on this array, ending with a reduction operation to compute their sum.

Revise to use data-parallel operations.

Following the strategy of making changes incrementally, we could start by making the changes just described and replacing the `do` loop with a `forall`, as shown in Fig. 3.

Further revise to use more-data-parallel operations.

We can then make the program more purely data-parallel by replacing each line of the body of the `forall` with an element-wise operation on the array `x`, as shown in Fig. 4. (We do still have to use a `forall` to give each element of the array a value based on its index. Some programming environments might provide a library function that would do this, however.)

Concluding remarks.

Two of the PLPP program structure patterns, *SPMD* and *Loop Parallelism*, also discuss this example. Both use the same strategy presented here (computing partial sums and

```
integer NUM_STEPS
parameter (NUM_STEPS=1000000)
integer i
real step, pi
real x(0:NUM_STEPS-1)

step = 1.0/(NUM_STEPS)
forall (i = 0:(NUM_STEPS-1))
    x(i) = (i+0.5)*step
    x(i) = 4.0/(1.0+(x(i)*x(i)))
end forall
pi = step*sum(x)

print *, pi
```

Figure 3: Data-parallel Fortran 95 program to carry out a midpoint rule integration to solve $\int_0^1 \frac{4}{1+x^2}\,dx$. This version uses `forall`.

```
integer NUM_STEPS
parameter (NUM_STEPS=1000000)
integer i
real step, pi
real x(0:NUM_STEPS-1)

step = 1.0/(NUM_STEPS)
forall (i = 0:(NUM_STEPS-1))
    x(i) = i
end forall
x = (x + 0.5)*step
x = 4.0 / (1.0 + x*x)
pi = step*sum(x)

print *, pi
```

Figure 4: Data-parallel Fortran 95 program to carry out a midpoint rule integration to solve $\int_0^1 \frac{4}{1+x^2}\,dx$. This version uses array operations.

combining them); only the way in which this strategy is turned into code is different.

Mesh Computation

The problem is to model one-dimensional heat diffusion (i.e., diffusion of heat along an infinitely narrow pipe). Initially the whole pipe is at a stable and fixed temperature. At time 0, we set both ends to different temperatures, which will remain fixed throughout the computation. We then calculate how temperatures change in the rest of the pipe over time. (What we expect is that the temperatures will converge to a smooth gradient from one end of the pipe to the other.) Mathematically, the problem is to solve a one-dimensional differential equation representing heat diffusion:

$$\frac{\partial U}{\partial t} = \frac{\partial^2 U}{\partial x^2}$$

The approach used is to discretize the problem space (representing U by a one-dimensional array and computing values for a sequence of discrete time steps). We will output values for each time step as they are computed, so we need only save values for U for two time steps; we will call these arrays `uk` (U at the timestep k) and `ukp1` (U at timestep $k+1$). At each time step, we then need to compute for each point in array `ukp1` the following:

```
ukp1(i) = uk(i) + &
    (dt/(dx*dx))*(uk(i+1)-2*uk(i)+uk(i-1))
```

Variables `dt` and `dx` represent the intervals between discrete time steps and between discrete points respectively. A program to carry this calculation out on a single processor is shown in Fig. 5. (Code for `print_values` is straightforward and not shown.) Notice that it employs a bit of cleverness to avoid copying values from `ukp1` to `uk` at every time step: It defines pointers to the two arrays and simply swaps pointers, achieving the same result without the additional cost of copying the whole array.

Since we are beginning with sequential code, we can work through the steps described previously for a strategy based on making changes incrementally, as follows.

Identify exploitable concurrency.

The overall calculation in this problem is a good fit for the PLPP *Algorithm Structure* pattern *Geometric Decomposition*, because the heart of the computation is repeated updates of a large data structure, namely `ukp1`.

Revise to use data-parallel operations.

Expressing the same calculation using data parallelism is actually simpler, however, given the semantics of the `forall` construct (as described in the section "Example Programming Environment (Fortran 95)" earlier); We can maintain a single copy of the array representing U and update all elements in parallel, thus:

```
integer NX, NSTEPS
parameter (NX=100)
parameter (NSTEPS=10000)
real LEFTVAL, RIGHTVAL      ! values for left,
                            ! right endpoints
parameter (LEFTVAL=1.0)
parameter (RIGHTVAL=10.0)
real dx, dt
integer i, k

real, pointer :: uk(:), ukp1(:), temp_ptr(:)

dx = 1.0/NX
dt = 0.5*dx*dx

allocate(uk(NX))          ! allocate space for uk
allocate(ukp1(NX))        ! allocate space for ukp1

do i = 2, NX-1
    uk(i) = 0.0
end do
uk(1) = LEFTVAL ; ukp1(1) = LEFTVAL
uk(NX) = RIGHTVAL ; ukp1(NX) = RIGHTVAL

do k = 1, NSTEPS

    do i = 2, NX-1
        ukp1(i) = uk(i) + &
            (dt/(dx*dx))*(uk(i+1)-2*uk(i)+uk(i-1))
    end do
    ! "copy" ukp1 to uk by swapping pointers
    temp_ptr => ukp1; ukp1 => uk; uk => temp_ptr

    call print_values(nx, uk)

end do
```

Figure 5: Sequential Fortran 95 program for one-dimensional heat equation.

```
forall (i = 2 : NX-1)
    uk(i) = uk(i) + &
      (dt/(dx*dx))*(uk(i+1)-2*uk(i)+uk(i-1))
end forall
```

Fig. 6 shows the complete program.

```
integer NX, NSTEPS
parameter (NX=100)
parameter (NSTEPS=10000)
real LEFTVAL, RIGHTVAL
parameter (LEFTVAL=1.0)
parameter (RIGHTVAL=10.0)

real uk(1:NX)
real dx, dt
integer i, k

dx = 1.0/NX
dt = 0.5*dx*dx

forall (i = 2 : NX-1)
    uk(i) = 0.0
end forall
uk(1) = LEFTVAL
uk(NX) = RIGHTVAL

do k = 1, NSTEPS
    forall (i = 2 : NX-1)
        uk(i) = uk(i) + &
            (dt/(dx*dx))*(uk(i+1)-2*uk(i)+uk(i-1))
    end forall
    call print_values(nx, uk)
end do
```

Figure 6: Data-parallel Fortran 95 program for one-dimensional heat equation. This version uses `forall`.

Further revise to use more-data-parallel operations.

Alternatively, we could express the required calculation in terms of operations on arrays, though it is slightly less straightforward to do so: Calculation of U for each point requires old values for the point itself and its left and right neighbors. Left and right neighbors are represented in the sequential code by array elements with indices `i-1` and `i+1`. But element `i-1` is just element `i` of the array formed by shifting all elements of `uk` to the right by one, and similarly for element `i+1` and an array formed by shifting `uk` left. Fortran 95 has such a function, namely `eoshift`: `eoshift(a,n)` returns an array obtained by shifting `a` by `n` positions (left for positive `n`, right for negative `n`) and padding with zeros. We can use it to do the required calculation thus:

```
uk = uk + (dt/(dx*dx)) * &
    (eoshift(uk,-1) - 2*uk + eoshift(uk,1))
```

provided we then correct the wrongly-computed values for the first and last element (which are supposed to remain constant). Fig. 7 shows the complete program.

```
integer NX, NSTEPS
parameter (NX=100)
parameter (NSTEPS=10000)
real LEFTVAL, RIGHTVAL
parameter (LEFTVAL=1.0)
parameter (RIGHTVAL=10.0)

real uk(1:NX)
real dx, dt
integer k

dx = 1.0/NX
dt = 0.5*dx*dx

uk = 0.0
uk(1) = LEFTVAL
uk(NX) = RIGHTVAL

do k = 1, NSTEPS
    ! compute interior points using array operations
    uk = uk + (dt/(dx*dx)) * &
        (eoshift(uk,-1) - 2*uk + eoshift(uk,1))
    ! correct boundary points
    uk(1) = LEFTVAL
    uk(NX) = RIGHTVAL
    call print_values(nx, uk)
end do
```

Figure 7: Data-parallel Fortran 95 program for one-dimensional heat equation. This version uses array operations.

Concluding remarks.

This example is discussed in some detail in text of *Geometric Decomposition* (in [19]), and implementations using MPI and OpenMP are shown.

Black-Scholes Option Pricing

As an example of a calculation from a different domain area, namely finance, consider the Black-Scholes model for option pricing [2]. As commonly defined, an option is an agreement between a buyer and a seller, in which the buyer is granted a right (but not an obligation), secured by the seller, to carry out some operation (exercise the option) at some time in the future. The predetermined price is called the strike price, and the future date is called the expiration date. A call option grants the buyer the right to buy the underlying asset at the strike price; a put option grants the buyer the right to sell the underlying asset at the strike price. There are several types of options, mostly depending on when the option can be exercised; European options can be exercised only on the expiration date, while American-style options are more flexible. For a call option, the profit made is the difference between the price of the asset at the exercise date and the strike price, minus the option price. For a put option, the profit made is the difference between the strike price and the price of the asset at the exercise date, minus the option price. How much one would be willing to pay for the option therefore depends on many factors, including:

- Asset price at expiration date.
- Strike price.
- Time to expiration date. (Longer times imply more uncertainty.)

- Riskless rate of return, i.e., annual interest rate of bonds or other investments considered to be risk-free. P dollars invested at this rate will be worth $P \cdot e^{rT}$ dollars T years from now; equivalently, one gets P dollars T years from now by investing $P \cdot e^{-rT}$ dollars now.

The model gives a partial different equation for the evolution of an option price under certain assumptions. For European options, there is a closed-form solution to this PDE:

$$\begin{aligned}
V_{call} &= S \cdot CND(d_1) - X \cdot e^{-rT} \cdot CND(d_2) \\
V_{put} &= X \cdot e^{-rT} \cdot CND(-d_1) - S \cdot CND(-d_2) \\
d_1 &= \frac{\log(\frac{S}{X}) + (r + \frac{v^2}{2})T}{v\sqrt{T}} \\
d_2 &= \frac{\log(\frac{S}{X}) + (r - \frac{v^2}{2})T}{v\sqrt{T}} \\
CND(-d) &= 1 - CND(d)
\end{aligned}$$

where

- V_{call} is the price for an option call
- V_{put} is the price for an option put
- $CND(d)$ is the Cumulative Normal Distribution function
- S is the current option price
- X is the strike price
- T is the time to expiration
- r is the continuously compounded risk-free interest rate
- v is the implied volatility for the underlying asset

Fig. 8 shows code, based on similar code provided as one of the CUDA SDK examples [21], that uses these equations to calculate V_{call} and V_{put} for a set of randomly-generated values of S, X, and T. To keep the code simpler, r, v, and the number of samples are hard-coded.

Since we are beginning with sequential code, we can work through the steps described previously for a strategy based on making changes incrementally, as follows.

Identify exploitable concurrency.

There are four obvious sources of exploitable concurrency, corresponding to the four loops in the main program. For each loop, if its iterations are independent, or nearly so, then they can be executed concurrently, as described in simplest of the PLPP *Algorithm Structure* patterns, *Task Parallelism*.

Refactor to eliminate obstacles to parallelism..

To find out whether the iterations are independent, we need to look at subprograms `rand_in_range` (for the first three loops) and `black_scholes_body` (for the fourth loop). We will assume that local variables for subprograms are allocated in a way that does not introduce dependencies between loop iterations (a reasonable assumption for Fortran 95 and probably for most target environments).

Looking first at `rand_in_range`, we observe that it in turn makes calls to `random_number`. This is a library function,

```
program blackscholes
integer OPT_N
parameter (OPT_N = 1000000)
real RISKFREE, VOLATILITY
parameter (RISKFREE = 0.02)
parameter (VOLATILITY = 0.30)
real option_price(1:OPT_N)         ! input
real option_strike(1:OPT_N)        ! input
real option_years(1:OPT_N)         ! input
real call_result(1:OPT_N)          ! output
real put_result(1:OPT_N)           ! output

! initialize with randomly-generated data
do i = 1, OPT_N
    option_price(i) = rand_in_range(5.0, 30.0)
end do
do i = 1, OPT_N
    option_strike(i) = rand_in_range(1.0, 100.0)
end do
do i = 1, OPT_N
    option_years(i)  = rand_in_range(0.25, 10.0)
end do
! do calculations
do i = 1, OPT_N
    call black_scholes_body(call_result(i), &
        put_result(i), &
        option_price(i), option_strike(i), &
        option_years(i), &
        RISKFREE, VOLATILITY)
end do
! code to print results not shown
end program

! generate uniformly distributed random real in
!   [low, high] range
real function rand_in_range(low, high)
real low, high, temp
call random_number(temp)
rand_in_range = (1.0 - temp)*low + temp*high
end function

! calculate Black-Scholes formula for call and put
subroutine black_scholes_body(call_result, &
    put_result, s, x, t, r, v)
real call_result, put_result
real s  ! option price
real x  ! option strike
real t  ! option years
real r  ! riskless rate
real v  ! volatility rate

real sqrt_t, exp_rt, d1, d2, cnd_d1, cnd_d2
real CND ! function to compute cumulative normal
         !   distribution

sqrt_t = sqrt(t)
d1 = (log(s / x) + (r + 0.5*v*v)*t) / (v*sqrt_t)
d2 = d1 - v*sqrt_t
cnd_d1 = CND(d1)
cnd_d2 = CND(d2)
! calculate call and cut simultaneously
exp_rt = exp(-r*t)
call_result = s*cnd_d1 - x*exp_rt*cnd_d2
put_result  = &
     x*exp_rt*(1.0 - cnd_d2) - s*(1.0 - cnd_d1)
end subroutine
```

Figure 8: Sequential Fortran 95 program for Black-Scholes option pricing.

```
! polynomial approximation of cumulative normal
!   distribution function
real function CND(d)

real A1, A2, A3, A4, A5, RSQRT2PI
parameter (A1 = 0.31938153)
parameter (A2 = -0.356563782)
parameter (A3 = 1.781477937)
parameter (A4 = -1.821255978)
parameter (A5 = 1.330274429)
parameter (RSQRT2PI = 0.3989422804)

real d, k

k = 1.0 / (1.0 + 0.2316419*abs(d))
CND = RSQRT2PI*exp(- 0.5*d*d) * &
    (k*(A1 + k*(A2 + k*(A3 + k*(A4 + k*A5)))))
if (d > 0) CND = 1.0 - CND
end function
```

Figure 9: Sequential Fortran 95 function to compute cumulative normal distribution, adapted from similar code provided as one of the CUDA SDK examples [21]. This code is included only so it can be examined as a potential source of loop-carried dependencies; the algorithm is not relevant to the discussion and is therefore not discussed.

and unless we can be sure it gives correct results even if multiple copies are invoked concurrently, we must regard its use as a loop-carried dependency, and this might mean that this loop would need to stay sequential. Fortunately, however, in addition to generating a single value, the library function is capable of generating multiple values into an array, and with a little refactoring of the sequential code we can make use of this feature to eliminate the first three loops, for example replacing

```
do i = 1, OPT_N
    option_price(i) = rand_in_range(5.0, 30.0)
end do
```

with

```
call rand_in_range(OPT_N, option_price, 5.0, 30.0)
```

(This does not by any means guarantee that the library function will do anything concurrently, but it does allow us to express the computation in a way that is readily seen to be correct and easily mapped onto a library function for generating random values in parallel, if one exists in the target programming environment.)

Looking next at `black_scholes_body`, we notice that it in turn makes calls to `CND`, but this function by inspection is safe for concurrent execution, since it uses only read-only constants and local variables. (Code for `CND` is given in Fig. 9.) So we should be able to replace this loop with a `forall`. Because of restrictions on what can be in the body of a `forall` construct, however, we cannot do this directly, but must refactor so that we pass whole arrays into `black_scholes_body` and perform the loop (to be replaced by a `forall`) inside the subroutine. The revised `black_scholes_body` will have the form

```
subroutine black_scholes_body(n, call_result,
    put_result, s, x, t, r, v)
integer n ! number of options
real call_result(1:n), put_result(1:n), &
    s(1:n), x(1:n), t(1:n)
real r, v
real sqrt_t, exp_rt, d1, d2, cnd_d1, cnd_d2, CND
integer i
do i = 1, n
    sqrt_t = sqrt(t(i))
    ! remaining lines similarly transformed
end do
end subroutine
```

When we do this, however, we introduce loop-carried dependencies for the intermediate results (e.g., `sqrt_t`). These are readily addressed with the technique described earlier: Replace each variable used for an intermediate result with an array that can be operated on as a parallel data structure. So, for example, we could replace the declaration of `sqrt_t` with an array declaration

```
real sqrt_t(1:n)
```

and rewrite the first line of the loop thus

```
sqrt_t(i) = sqrt(t(i))
```

Revise to use data-parallel operations..

At this point we can replace the loop in `black_scholes_body` with a `forall`. Following the strategy of making changes incrementally, at this point it would probably be wise to build the program and confirm that it produces correct output.

Further revise to use more-data-parallel operations..

Finally, we can transform the `forall` construct into a sequence of whole-array operations; for example, the first line can be replaced with

```
sqrt_t = sqrt(t)
```

The complete program is shown in Fig. 10.

```
program blackscholes
integer OPT_N
parameter (OPT_N = 1000000)
real RISKFREE, VOLATILITY
parameter (RISKFREE = 0.02)
parameter (VOLATILITY = 0.30)
real option_price(1:OPT_N)        ! input
real option_strike(1:OPT_N)       ! input
real option_years(1:OPT_N)        ! input
real call_result(1:OPT_N)              ! output
real put_result(1:OPT_N)               ! output

! generate input data
call rand_in_range(OPT_N, option_price, 5.0, 30.0)
call rand_in_range(OPT_N, option_strike, 1.0, 100.0)
call rand_in_range(OPT_N, option_years, 0.25, 10.0)
! do calculations
call black_scholes_body(OPT_N, call_result, &
    put_result, &
    option_price, option_strike, option_years, &
    RISKFREE, VOLATILITY)
! code to print results not shown
end program

! generate uniformly distributed random reals in
!    [low, high] range
subroutine rand_in_range(n, array, low, high)
integer n
real array(1:n), low, high
call random_number(array)
array = (1.0 - array)*low + array*high
end subroutine

! calculate Black-Scholes formula for call and put
subroutine black_scholes_body(n, call_result,
    put_result, s, x, t, r, v)
integer n ! number of options
real call_result(1:n), put_result(1:n)
real s(1:n)  ! option price
real x(1:n)  ! option strike
real t(1:n)  ! option years
real r  ! riskless rate
real v  ! volatility rate

real sqrt_t(1:n), exp_rt(1:n), d1(1:n), d2(1:n), &
    cnd_d1(1:n), cnd_d2(1:n)

! function to compute cumulative normal distribution
!    ("elemental" means it can be applied element-wise
!    to array)
interface
    elemental real function CND(d)
    real, intent(in) :: d
    end function
end interface

sqrt_t = sqrt(t)
d1 = (log(s / x) + (r + 0.5*v*v)*t) / (v*sqrt_t)
d2 = d1 - v*sqrt_t
cnd_d1 = CND(d1)
cnd_d2 = CND(d2)
! calculate call and cut simultaneously
exp_rt = exp(- r*t)
call_result = s*cnd_d1 - x*exp_rt*cnd_d2
put_result  = &
    x*exp_rt*(1.0 - cnd_d2) - s*(1.0 - cnd_d1)
end subroutine
```

Figure 10: Data-parallel Fortran 95 program for Black-Scholes option pricing.

Scan

A simple but frequently useful operation is computing all partial sums of an array of numbers, sometimes called *sum-prefix* or *all-prefix-sums*. This operation can be generalized to a so-called *scan* operation (to use the terminology of APL [10]). This operation requires a sequence $a_0, a_1, \ldots, a_{n-1}$ and a binary operator $\circ$ with an identity element. Output is a new sequence with element i defined as

$$a_0 \circ a_1 \circ \cdots \circ a_i$$

This is a so-called *inclusive scan*; if the sum does not include the element itself, it is called an *exclusive scan.* (It is clearly straightforward to convert between the two types of scans: inclusive to exclusive by shifting right one position, padding with the identity element, and exclusive to inclusive by shifting left one position and inserting as the last element the sum of the last element of the inclusive scan and the first element of the original sequence.) Fig. 11 shows code for a naive sequential algorithm for an inclusive scan using addition.

```
integer a(0:N-1)
integer k
! code to initialize a omitted
do k = 1, N-1
    a(k) = a(k) + a(k-1)
end do
```

Figure 11: Sequential Fortran 95 code for computing partial sums of an array (inclusive scan). `N` is the size of the array.

Although we are starting with sequential code, a strategy of incremental parallelism will not help here. Instead, what is needed is a rethinking of the problem. Hillis and Steele [7] point out that while this appears to be an inherently sequential operation with execution proportional to the length of the array, clever use of data parallelism can reduce the execution time from $O(n)$ to $O(\log n)$, given enough processors to assign a processor to each element. A more work-efficient version of the algorithm was later proposed by Blelloch [3]. The examples in [7] were part of the inspiration for one of the more interesting of the PLPP *Algorithm Structure* patterns, *Recursive Data*, which captures the overall strategy used both for Hillis and Steele's parallel algorithm and for Blelloch's modified version. In the rest of this section we discuss these two algorithms.

Hillis/Steele algorithm.

The overall idea is to make $\log_2 n$ passes through the array, where n is its size, first adding to each element k the $(k-1)$-th element, then the $(k-2)$-th element, then the $(k-4)$-th element, and so forth, as illustrated in Fig. 12. Fig. 13 shows code for this algorithm. Notice that the body of the `forall` construct is executed only for indices that match the mask; in iteration j of the outer loop, the body of the `forall` will be executed only for those k for which $k \geq 2^{j-1}$. Recall also that statements in the body of a `forall` operate as a parallel assignment; in this case, replacing the `forall` with

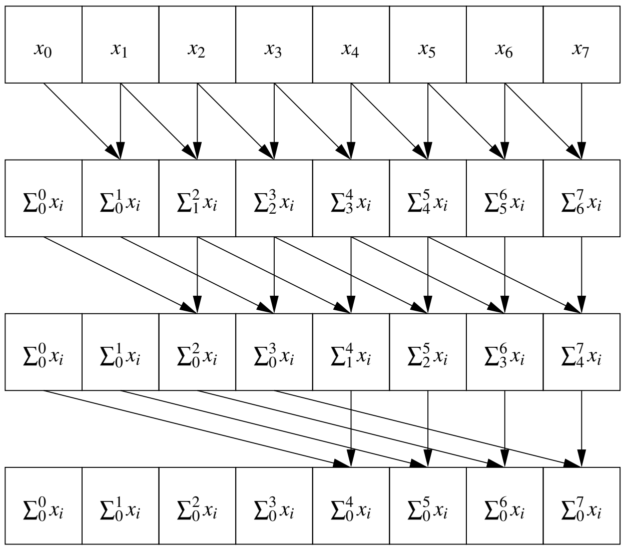

Figure 12: Computing partial sums of an array (inclusive scan) using Hillis and Steele's algorithm.

a sequential loop would give a different and incorrect result.

```
integer a(0:N-1)
integer j, k
! code to initialize a omitted
do j = 1, log2(N)
    ! assign values for all k in 0 .. N-1 such that
    !   k >= 2**(j-1)
    forall (k = 0 : N-1, k >= 2**(j-1))
        a(k) = a(k - 2**(j-1)) + a(k)
    end forall
end do
```

Figure 13: Data-parallel Fortran 95 code for computing partial sums of an array (inclusive scan). `N` **is the size of the array.**

However, although this algorithm is clever, and faster than the simple sequential version if there are enough processors, it achieves this result at a cost of more total work — $O(n \log n)$ operations total, as opposed to the $O(n)$ operations required for the sequential algorithm. If there are significantly more array elements than processors, the data-parallel algorithm of Fig. 13 might even be slower than its sequential counterpart. Addressing this problem requires additional rethinking.

Blelloch algorithm.
The idea of this more work-efficient algorithm is as follows: Conceptually build a binary tree based on the array, with leaf nodes corresponding to elements, and non-leaf nodes that combine successive nodes at the next lower level. (So for an array with 8 elements, the root node of the tree represents elements 0–7, the nodes one level down represent elements 0–3 and 4–7, and so forth.) Divide the computation into two phases, an *up-sweep* that moves up the tree computing partial sums, as illustrated in Fig. 14, and a *down-sweep* that uses these results to compute the desired prefix sums for all array elements, as illustrated in Fig. 15. Notice that this algorithm computes an exclusive scan rather than an inclusive one. Fig. 16 shows code for this algorithm.

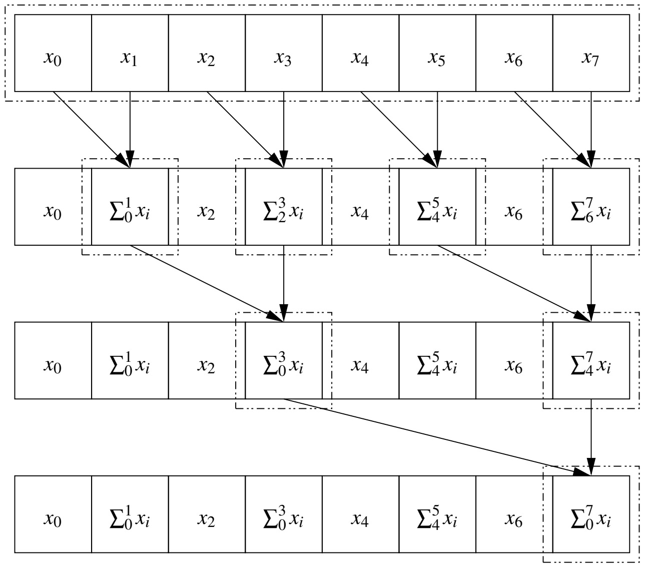

Figure 14: Computing partial sums of an array (exclusive scan) using Blelloch's algorithm, part 1 (up-sweep phase). Dashed boxes show elements that make up the tree described in the text.

Notice that again we use a mask in the `forall`, this time to specify that in iteration j of the outer loop, the body of the `forall` will be executed only for those k for which $(k \mod 2^{j+1}) = 0$ (i.e., every 2^{j+1}-th element). Recall also that statements in the body of a `forall` function as a parallel assignment; in this case, replacing the `forall` with a sequential loop would give a different and incorrect result.

Concluding remarks.
It is worth remarking that neither of these algorithms would probably be used if the only goal were to compute partial sums of an array: The total amount of work even for a fairly large array is too small to justify the added complexity and overhead, and this kind of calculation is common enough that the target programming environment might provide a library function for it. However, both algorithms are interesting in their own right, as illustrations that data parallelism is more broadly applicable than one might at first think. Further, either algorithm might be practical in the right setting — for example, if it were part of a larger calculation with enough total work to justify the overhead of managing data parallelism.

Known Uses

- Programs using one of the programming environments described earlier. Examples using Accelerator can be found in [24]; examples using the PeakStream Platform can be found at the company's Web site [22].

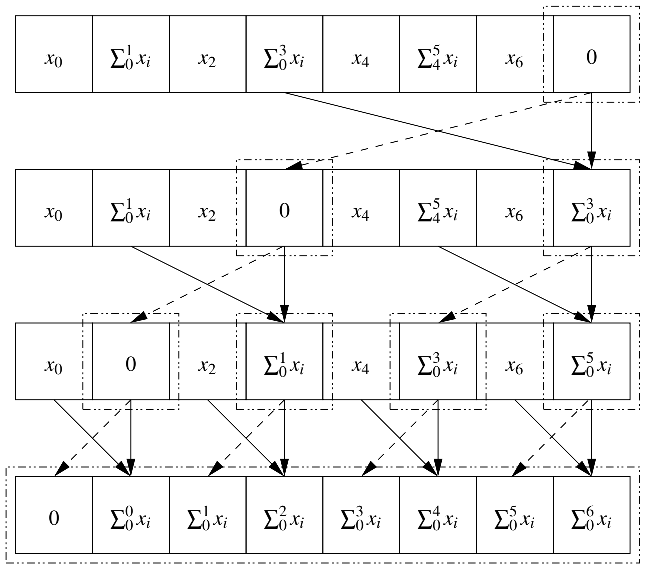

Figure 15: Computing partial sums of an array (exclusive scan) using Blelloch's algorithm, part 2 (down-sweep phase). Dashed boxes show elements that make up the tree described in the text. Not shown is the first step of the phase, which simply stores the identity element (0) in the last element of the array.

```
integer a(0:N-1)
integer t(0:N-1)   ! array of temporary variables
integer j, k
! code to initialize a omitted

! up-sweep phase
do j = 0, log2(N)-1
    ! execute body of forall only for every
    !   2**(j+1)-th element
    forall (k = 0 : N-1, mod(k, 2**(j+1)) == 0)
      a(k + 2**(j+1) - 1) = &
          a(k + 2**j - 1) + a(k + 2**(j+1) - 1)
    end forall
end do

! down-sweep phase
a(N-1) = 0
do j = log2(N)-1, 0, -1
    ! execute body of forall only for every
    !   2**(j+1)-th element
    forall (k = 0 : N-1, mod(k, 2**(j+1)) == 0)
        t(k) = a(k + 2**j - 1)
        a(k + 2**j - 1) = a(k + 2**(j+1) - 1)
        a(k + 2**(j+1) - 1) = &
            t(k) + a(k + 2**(j+1) - 1)
    end forall
end do
```

Figure 16: Work-efficient data-parallel Fortran 95 code for computing partial sums of an array (exclusive scan). `N` is the size of the array.

- Examples used in the PLPP *Recursive Data* pattern:
  - Finding, for each node in a forest of trees, the root of the tree to which it belongs, in time proportional to the logarithm of the height of the forest, as discussed in [11].
  - Finding partial sums in a linked list, as discussed in [7].
- Other data-parallel algorithms from [7].

Related Patterns

The *SIMD* pattern joins and complements the other *Supporting Structures* patterns describing program structure (*SPMD*, *Loop Parallelism*, and so forth). As described in [19], which of these patterns to use for a particular application usually depends on a combination of target programming environment and the overall strategy for exploiting concurrency (i.e., which *Algorithm Structure* patterns are being used). Just as *SPMD* is a good fit for MPI and *Loop Parallelism* is a good fit for OpenMP, *SIMD* is a good fit if the target programming environment provides the features described in the Solution section of this pattern. An additional pattern at this level will probably be needed in order to properly address stream-processing programming environments; these environments target similar hardware, but express calculations in terms of a kernel (sequence of instructions) operating in parallel on elements of a data structure (stream) rather than in terms of a single sequence of operations, each of which operates concurrently on elements of a data structure.

With regard to overall strategy, the *Algorithm Structure* patterns that map best to *SIMD* are *Task Parallelism*, *Geometric Decomposition*, and *Recursive Data*, since all have a kind of regularity that lends itself to being expressed in terms of operations on parallel data structures. The other *Algorithm Structure* patterns (*Divide and Conquer*, *Pipeline*, and *Event-Based Coordination*) do not map as well to *SIMD*.

3. ACKNOWLEDGMENTS

We gratefully acknowledge the help of our shepherd for this paper, Danny Dig.

4. REFERENCES

[1] Microsoft Accelerator. `http://research.microsoft.com/research/downloads/Details/25e1bea3-142e-4694-bde5-f0d44f9d8709/Details.aspx?CategoryID`.

[2] F. Black and M. Scholes. The pricing of options and corporate liabilities. *Journal of Political Economy*, 81(3):637–654, 1973.

[3] G. E. Blelloch. Prefix sums and their applications. In J. H. Reif, editor, *Synthesis of Parallel Algorithms*. Morgan Kaufmann, 1990.

[4] Brook. `http://graphics.stanford.edu/projects/brookgpu/`.

[5] M. J. Flynn. Some computer organizations and their effectiveness. *IEEE Transactions on Computers*, C-21(9), 1972.

[6] W. D. Hillis. *The Connection Machine*. MIT Press, 1985.

[7] W. D. Hillis and G. L. Steele, Jr. Data parallel algorithms. *Communications of the ACM*, 29(12):1170–1183, 1986.

[8] High Performance Fortran Forum: High Performance Fortran Language specification, version 2.0. `http://dacnet.rice.edu/Depts/CRPC/HPFF`, 1997.

[9] IBM Corporation. XL Fortran for AIX 8.1 language reference. `http://www.ncsa.uiuc.edu/UserInfo/Resources/Hardware/IBMp690/IBM/usr/share/man/info/en_US/xlf/html/lr.HTM`.

[10] K. E. Iverson. *A Programming Language*. Wiley, 1962.

[11] J. JáJá. *An Introduction to Parallel Algorithms*. Addison-Wesley, 1992.

[12] B. L. Massingill, T. G. Mattson, and B. A. Sanders. Patterns for parallel application programs. In *Proceedings of the Sixth Pattern Languages of Programs Workshop (PLoP 1999)*, August 1999.

[13] B. L. Massingill, T. G. Mattson, and B. A. Sanders. Patterns for finding concurrency for parallel application programs. In *Proceedings of the Seventh Pattern Languages of Programs Workshop (PLoP 2000)*, August 2000.

[14] B. L. Massingill, T. G. Mattson, and B. A. Sanders. More patterns for parallel application programs. In *Proceedings of the Eighth Pattern Languages of Programs Workshop (PLoP 2001)*, September 2001.

[15] B. L. Massingill, T. G. Mattson, and B. A. Sanders. Some algorithm structure and support patterns for parallel application programs. In *Proceedings of the Ninth Pattern Languages of Programs Workshop (PLoP 2002)*, September 2002.

[16] B. L. Massingill, T. G. Mattson, and B. A. Sanders. Additional patterns for parallel application programs. In *Proceedings of the Tenth Pattern Languages of Programs Workshop (PLoP 2003)*, September 2003.

[17] B. L. Massingill, T. G. Mattson, and B. A. Sanders. Reengineering for parallelism: An entry point into PLPP (pattern language for parallel programming) for legacy applications. In *Proceedings of the Twelfth Pattern Languages of Programs Workshop (PLoP 2005)*, September 2005.

[18] B. L. Massingill, T. G. Mattson, and B. A. Sanders. Reengineering for parallelism: An entry point into PLPP (pattern language for parallel programming) for legacy applications. *Concurrency and Computation: Practice and Experience*, 19(4):503–529, 2007.

[19] T. G. Mattson, B. A. Sanders, and B. L. Massingill. *Patterns for Parallel Programming*. Addison-Wesley, 2004. See also our Web site at `http://www.cise.ufl.edu/research/ParallelPatterns`.

[20] M. Metcalf and J. K. Reid. *Fortran 90/95 Explained*. Oxford University Press, 1996.

[21] NVIDIA CUDA (Compute Unified Device Architecture). `http://developer.nvidia.com/object/cuda.html`.

[22] The PeakStream platform. `http://www.peakstreaminc.com/`.

[23] The RapidMind platform. `http://www.rapidmind.net/`.

[24] D. Tarditi, S. Puri, and J. Oglesby. Accelerator: Using data parallelism to program GPUs for general-purpose uses. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM Press, 2006.

APPENDIX

A. REVIEW OF PLPP

This section provides a brief overview of our pattern language (PLPP) as described in [19], for the convenience of readers unfamiliar with the language.

The pattern language is organized into four design spaces, corresponding to the four phases of the underlying methodology and described in the subsequent sections. Programmers normally start at the top (in *Finding Concurrency*) and work down through the other design spaces in order until a detailed design for a parallel program is obtained.

A.1 The *Finding Concurrency* Design Space

This design space is concerned with structuring the problem to expose exploitable concurrency. The designer working at this level focuses on high-level algorithmic issues and reasons about the problem to expose potential concurrency. Patterns in this space include the following.

- Decomposition patterns, used to decompose the problem into pieces that can execute concurrently:
  - *Task Decomposition*: How can a problem be decomposed into tasks that can execute concurrently?
  - *Data Decomposition*: How can a problem's data be decomposed into units that can be operated on relatively independently?
- Dependency-analysis patterns, used to help group the tasks and analyze the dependencies among them:
  - *Group Tasks*: How can the tasks that make up a problem be grouped to simplify the job of managing dependencies?
  - *Order Tasks*: Given a way of decomposing a problem into tasks and a way of collecting these tasks into logically related groups, how must these groups of tasks be ordered to satisfy constraints among tasks?
  - *Data Sharing*: Given a data and task decomposition for a problem, how is data shared among the tasks?
- *Design Evaluation*: Is the decomposition and dependency analysis so far good enough to move on to the *Algorithm Structure* design space, or should the design be revisited?

A.2 The *Algorithm Structure* Design Space

This design space is concerned with structuring the algorithm to take advantage of potential concurrency. That is, the designer working at this level reasons about how to use the concurrency exposed in working with the *Finding Concurrency* patterns. The *Algorithm Structure* patterns describe overall strategies for exploiting concurrency. Patterns in this space include the following.

- Patterns for applications where the focus is on organization by task:
  - *Task Parallelism*: How can an algorithm be organized around a collection of tasks that can execute concurrently?

- *Divide and Conquer*: Suppose the problem is formulated using the sequential divide and conquer strategy. How can the potential concurrency be exploited?

- Patterns for applications where the focus is on organization by data decomposition:
  - *Geometric Decomposition*: How can an algorithm be organized around a data structure that has been decomposed into concurrently updatable "chunks"?
  - *Recursive Data*: Suppose the problem involves an operation on a recursive data structure (such as a list, tree, or graph) that appears to require sequential processing. How can operations on these data structures be performed in parallel?
- Patterns for applications where the focus is on organization by flow of data:
  - *Pipeline*: Suppose that the overall computation involves performing a calculation on many sets of data, where the calculation can be viewed in terms of data flowing through a sequence of stages. How can the potential concurrency be exploited?
  - *Event-Based Coordination*: Suppose the application can be decomposed into groups of semi-independent tasks interacting in an irregular fashion. The interaction is determined by the flow of data between them, which implies ordering constraints between the tasks. How can these tasks and their interaction be implemented so they can execute concurrently?

A.3 The *Supporting Structures* Design Space

This design space represents an intermediate stage between the *Algorithm Structure* and *Implementation Mechanisms* design spaces: It is concerned with how the parallel algorithm is expressed in source code, with the focus on high-level program organization rather than low-level and very specific parallel programming constructs. Patterns in this space include the following.

- Patterns representing approaches to structuring programs:
  - *SPMD*: The interactions between the various units of execution (UEs) cause most of the problems when writing correct and efficient parallel programs. How can programmers structure their parallel programs to make these interactions more manageable and easier to integrate with the core computations?
  - *Master/Worker*: How should a program be organized when the design is dominated by the need to dynamically balance the work on a set of tasks among the units of execution?
  - *Loop Parallelism*: Given a serial program whose runtime is dominated by a set of computationally intensive loops, how can it be translated into a parallel program?
  - *Fork/Join*: In some programs, the number of concurrent tasks varies as the program executes, and the way these tasks are related prevents the use of simple control structures such as parallel loops. How can a parallel program be constructed around such complicated sets of dynamic tasks?
- Patterns representing commonly-used data structures:
  - *Shared Data*: How does one explicitly manage shared data inside a set of concurrent tasks?
  - *Shared Queue*: How can concurrently-executing units of execution (UEs) safely share a queue data structure?
  - *Distributed Array*: Arrays often need to be partitioned between multiple units of execution. How can we do this so as to obtain a program that is both readable and efficient?

This design space also includes brief discussions of some additional supporting structures found in the literature, including SIMD (Single Instruction Multiple Data), MPMD (Multiple Program, Multiple Data), client server, declarative parallel programming languages, and problem solving environments.

A.4 The *Implementation Mechanisms* Design Space

This design space is concerned with how the patterns of the higher-level spaces are mapped into particular programming environments. We use it to provide descriptions of common mechanisms for process/thread management and interaction. The items in this design space are not presented as patterns since in many cases they map directly onto elements within particular parallel programming environments. We include them in our pattern language anyway, however, to provide a complete path from problem description to code. We discuss the following three categories.

- UE[5] management: Concurrent execution by its nature requires multiple entities that run at the same time. This means that programmers must manage the creation and destruction of processes and threads in a parallel program.
- Synchronization: Synchronization is used to enforce a constraint on the order of events occurring in different UEs. The synchronization constructs described here include memory fences, barriers, and mutual exclusion.
- Communication: Concurrently executing threads or processes sometimes need to exchange information. When memory is not shared between them, this exchange occurs through explicit communication events. The major types of communication events are message passing and collective communication, though we briefly describe several other common communication mechanisms as well.

[5]Units of execution — generic term for processes or threads.