The DKU Pattern for Performance Portable Parallel Programming

Sean Halle
UC Santa Cruz,
Santa Cruz, Ca
Email: seanhalle@yahoo.com

Albert Cohen
INRIA,
Saclay, France
Email: albert.cohen@inria.fr

Abstract

The shift to an ever increasing number of cores on a chip is driving the need for parallel programming methods that allow "single source, multiple hardware, high performance on each". The difficulty of designing such methods is exemplified in the embedded industry. Here, the tradition is hand-coding for ultra-high performance on specialized architectures, so the single source must be automatically transformed to give performance comparable to hand-coding. However, it is desired that source have no knowledge of the MPSoC; it must be written in a generic way, providing the information that an automated process will use to make the program efficient on a particular chip. To achieve this, the programmer must state, in essence, how to change the size of a scheduled unit of work so that automated task-size tuning can take place.

We propose a programming pattern, for the case of data-parallelism, to help provide that information, called DKU, which is short for "Divider-Kernel-Undivider". In this construct, the programmer writes three separate pieces of code: code that divides the iteration space plus data-structure into pieces; code that computes the answer for one piece; and code that puts the individual answers together into the larger answer. Because the programmer provides this code, the code can work with any data-structure, leaving the programmer free to choose structures natural to the problem.

DKU has been implemented as part of the Open Media Platform project where it is added to Java and supported by a web infrastructure that specializes the single source to multiple hardware platforms. A client device requests a program and automatically receives the executable specialized to that client's hardware.

1 Introduction

The DKU pattern is intended for performance portability of parallel code when exploiting data-parallelism. The pattern can be added to any language and remain backwards compatible; DKU-enabled applications run un-modified in the original language. The pattern specifies a certain grouping of functionality and uses naming conventions to identify each function. In this way, the functional aspects of parallel code are separated from performance aspects, and each kind of code is clearly identified to the compiler.

Performance portability requires automation that modifies the source code to fit with hardware characteristics. For example, high performance is achieved in different ways on a 4 core shared-memory CPU, versus a distributed memory 7-core heterogeneous MPSoC, versus a 100-core GPU. In each case, the source must be transformed to fit the hardware if its performance is to compete with code written specifically for that hardware [9].

Because of the need to specialize to hardware, the compiler must be split into a front-end used like a normal compiler, plus one or more additional back-ends that specialize. The desires for single, generic, source and simultaneously high performance can only both be achieved by separating source development from specialization, making them two steps, and automating the second. The front-end is used to develop the source, while hardware specialization is automatically performed when hardware knowledge becomes available. Hardware knowledge is gained during download, during install, or during a run, in all of these cases, after and independently of writing the source. Hence the separation of source development, using a front-end compiler, and specialization, using some automated back-end tool.

The back-end specializers have hardware knowledge built-in to them. They use the information exposed by the DKU pattern to specialize to that hardware.

In our proof-of-concept implementation, specialization is performed inside web-based infrastructure. This DKU web infrastructure automatically runs the back-end specializers upon receipt of new source code. The benefit is automated decoupling of source development from hardware specialization. Thus new specializers can be added long after source code has been completed, allowing new parallel hardware to use "dusty deck"s.

The process of specialization is hidden from both source developers and end-users. The web infrastructure receives all source, so the source developers only see one site that they send their program to. The site generates specializations upon receipt. End-users also see the same site. They request programs from that site (telling it the hardware they have). Behind the scenes, the web infrastructure automatically chooses the specialized executable most appropriate for the client hardware. Thus the specialization is hidden, only the provider of the web infrastructure and the providers of the specializer back-ends are aware of the specialization taking place.

The question arises as to how convenient is to use a construct such as the DKU pattern. The currently popular parallel programming solutions that give high performance on general data structures already require the programmer to write some form of divider, kernel, and undivider code, even though this decomposition may not be readily apparent. They also require the application to include varying degrees of support code for exploiting the parallelism. The support code discovers machine characteristics, decides optimal data-piece sizes, coordinates the distribution of data-pieces and then re-combines results. The support code determines the amount of parallelism delivered – but it doesn't do any of the actual work. This code exists in current parallel source even though the boundary between support code and the divider, kernel, and undivider code is not always obvious.

The DKU pattern makes the boundary explicit between the divider, kernel, and undivider code, and automates the support tasks. The support tasks are not functional, they only determine how much parallelism is exploited by the hardware, and thus the performance. The support behavior can safely change without changing the result. This allows taking the support tasks out of the source code and placing them inside the "language", where they are implemented differently on each hardware. Thus, the same source code will behave differently on each hardware platform; it will behave functionally the same on all hardware, but the parallelism behavior will be different.

Hence, using the DKU pattern requires the same effort as other methods for the division, kernel, and undivision code, but eliminates the effort spent on support-of-parallelism code.

In summary, the DKU pattern conveys the information needed by back-ends to specialize the source to high performance versions. The needed information consists of: how to break a data-structure into smaller pieces, how to break the iteration space into smaller pieces, how to perform the calculations within one piece of the iteration space, and how to put together the results from the individual interation spaces. The DKU pattern does this whithout any information in the source that would favor one kind of hardware over another. The information provided enables the back-end specializers to re-size computation-pieces and map them onto hardware resources in whatever way the specializer approaches this optimization problem. Finally, the pattern isolates the application from communication protocol and synchronization mechanisms.

1.1 background: the computation model

This paper assumes a particular computation model:

- Multiple processors
- Each processor atom-ically applies a code-snippet to data (an atomic-computation)
- Each processor communicates data with the other processors
- A Scheduling operation takes place for each processor for each atomic work-unit
- Scheduling operation uses computation resources

- Communication consists of latency and bandwidth: Latency has a hardware-dependent function that states the distribution of latencies as a function of number of communications initiated per communication delivered. Bandwidth delivered to a particular communication is a hardware dependent function of communication pattern. The pattern of communications in progress during an interval sets the bandwidth delivered to a particular communication.

Given this model, performance depends on the atomic-computation size[9]. Too large an atomic-computation size causes processors to sit idle waiting for large atomic-computations to free up work. Too small an atomic-computation size consumes too much scheduling computation, causes too much routing congestion in most networks, and loses too much time to latency of messages while the delivered bandwidth remains small. Also, if the computational complexity is super-linear in data-size, such as O(N logN) or stronger, then the total communication bit-volume consumed during the solution of a problem increases as the message-size decreases. Below the point of perfect overlap, communication becomes the bottleneck, slowing computation.

Atomic-computation size must be tuned to come close to the sweet spot. It can be tuned in two ways[9]:

- Change total trace length performed on a given size of data
- Change the size of data a given code-snippet is applied to

Changing trace length means changing one of: the size of the code-snippet; the complexity of the code-snippet; or the number of iterations performed by the code-snippet. Meanwhile, changing size of data in an atomic-computation requires knowledge of the data-structure, requires a kernel that can compute on changed data-sizes, and requires knowledge of how to combine the results.

In data-parallelism, setting the amount of data paired with a kernel is equivalent to setting the number of iterations the kernel has to do to consume the data. This is expressed in the DKU pattern. However, DKU does not cover changing the trace-length by modifying the code-snippet. Rather, DKU addresses changing the size of data in each atomic-computation, and automates the choice-of-size, scheduling-onto-processors, and communications.

1.2 Existing solutions to parallel programming

Currently popular parallel programming solutions that allow working with general data structures include Skeletons[5], Streaming Languages[11], Object Oriented Language enhancements like X10[3] and Titanium[2], and semi-automated language enhancements like MPI[12] and Cilk[1]. To get parallelism from *general* data structures in the OO language enhancements and the semi-automatic language enhancements, although it is not obvious, the programmer must write some form of divider, kernel, and undivider code. The application must also include support code for exploiting the parallelism: code that discovers machine characteristics, decides optimal data-piece sizes, coordinates the distribution of data-pieces and then re-combines results. The support code determines the amount of parallelism delivered – but it doesn't do any of the actual work. A subtle but important point is that the boundary between support code and the divider, kernel, and undivider code in these languages is not always obvious, but it is there.

The DKU pattern differs from these in two ways: it makes the boundary explicit between the divider, kernel, and undivider code, and automates the support tasks. The support tasks are not functional, they only determine how much parallelism is exploited by the hardware, and thus the performance. Taking the support tasks out of the source code and placing them underneath the pattern allows them to be implemented differently on each hardware. Thus, the same source code will behave differently on each hardware platform. It will behave functionally the same on all hardware, but the parallelism behavior will be different.

The DKU pattern has much in common with Skeletons used for parallel programming. Skeletons are a popular paradigm for parallel programming. What qualifies as a Skeleton is a bit vague, but in general, a Skeleton is considered a pattern embedded within a program written in a traditional programming language. The use of the pattern invokes some form of automation that simplifies at least some of the parallel programming details. DKU's main advantages over other Skeletons are practical ones.

Most Skeletons for parallel programming can be arranged into two sub-categories. One category has skeletons that are quite light-weight, imposing few restrictions on the programmer. However, these also tend to provide relatively little help, providing most often some simple form of fork/join parallelism. The other category has skeletons that provide more automation, such as so-called meta-programming style skeletons [7][4]. However, these tend to restrict the form of the code, and they have a relatively steep learning curve compared to the less restrictive forms.

A language extension called CAPSULE [10], is similar to DKU in that it also explicitly identifies a scheduler in the source code. As in DKU, the implementations of CAPSULE make this scheduler's behavior hardware-dependent. However, their approach focuses on control parallelism (pipeline parallelism); the application tells the scheduler when there is an opportunity for iteration-space division, often in recursive form. The hardware-dependent scheduler decides if it will take that opportunity. Hence the behavior of the CAPSULE language is different on different hardware implementations. This contrasts to DKU, which has no active semantics, only making semantic information available to a specializer. CAPSULE also has extensions to allow division of data for a distributed chip. The semantics of CAPSULE are well suited to implementation with a run-time system (which may in turn be implemented as library calls). However, it is less clear how to extract semantics from CAPSULE code that can be used for back-end specializations.

Sequoia[6] is a popular data parallelism language. It states data affinity to portions of iteration space. This, in essence, intermixes portions of the scheduling process with encoding of dependencies, because stating dependency of data upon a "sub-task" is equivalent to dividing work. Thus the language also has, like DKU, an explicit concept of a divider. The language also exposes the scheduling process explicitly in the source by making sub-task an explicit entity in the source. However, this choice of semantics also forces partially implementing the scheduler in the source, which limits portability.

To manage portability, Sequoia includes explicit management of data assignment to memory hierarchy levels, in the application code. For each hardware platform, the application programmer must supply a mapping that takes the "abstract" hierarchy defined in the application, and assigns pieces of it onto specific hardware. The language then provides some automation for choosing which of the programmer-supplied "mapping"s it uses. This is similar to DKU's automation infrastructure, but is more of an intermediate level of separation. It requires more effort from the application provider and requires them to learn the memory characteristics of each hardware platform.

DKU does not have a mechanism for such an intermediate level of separation between source and hardware. However, we plan future enhancements to the automated infrastructure that let application programmers add commands that specific specializers understand. Some of these commands will relate to managing work placement. Thus, the application programmer will incrementally improve the performance of their code on chosen hardware, without affecting the original source and without the possibility of introducing bugs.

2 Description

Figure 1 shows the basic structure of code written according to the DKU pattern. A data-structure, representing one atom-ically completed piece of work to perform, flows into a scheduler. The scheduler decides how many pieces to divide that work into. It places its decision into the data structure and hands it to the divider. The divider creates a number of identical data-structures, one for each sub-piece. Into each, the divider places some programmer-defined indication of the boundaries in iteration space (eg, FOR loop's start and end values). This sets the amount of work to do. The divider hands the pieces back to the scheduler. The scheduler may then repeat the process, further sub-dividing each sub-piece (allowing the work-piece hierarchy to be matched to the hardware hierarchy). When the scheduler is satisfied, it passes the leaves of the piece-hierarchy along to the kernel.

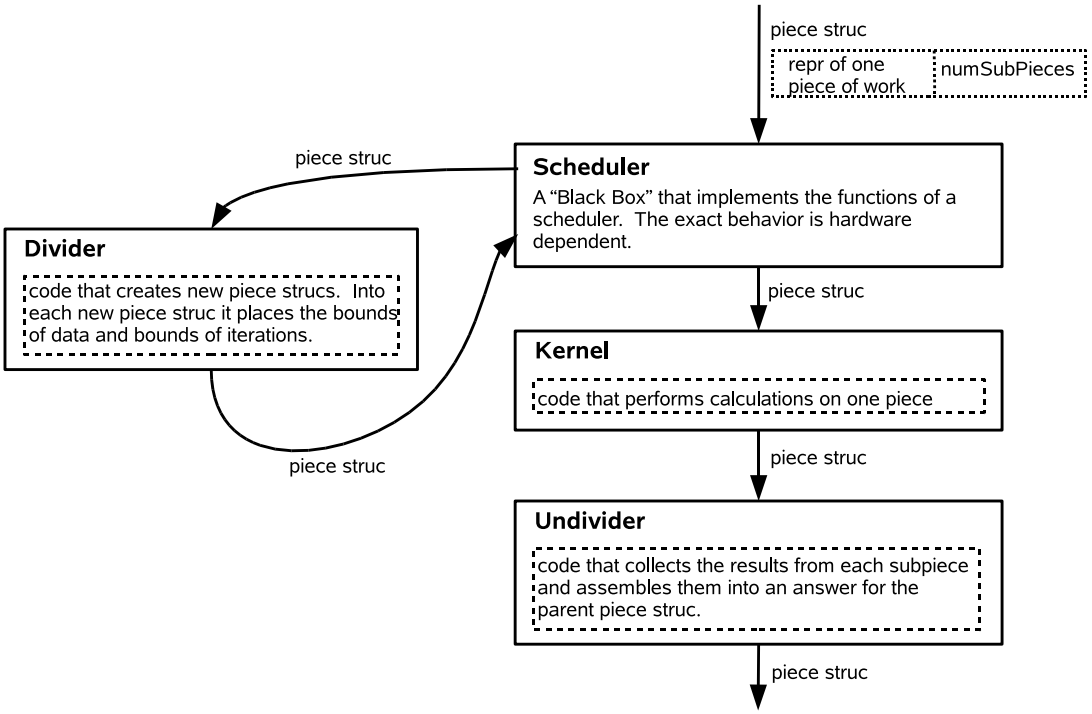

Figure 1. The flow of data through the elements of the DKU pattern

This is where the parallelism comes in to play. Semantically, the kernel code is replicated once for each piece struc. This is equivalent to each piece struc having its own processor performing the kernel just on that one piece. The scheduler controls how many pieces get made, and thus how many copies of the kernel are active at one time. The scheduler is implemented differently on each hardware platform, designed to choose the number of pieces that fits well to the number of processors available and to the network available.

Once the kernel completes on each piece struc, the piece is passed along to the undivider, which combines the sub-pieces into an answer for their parent piece.

2.1 DKU pattern in the Object Oriented paradigm

Figure 2 represents both the call structure and the data structure when the DKU pattern is implemented in an object oriented language.

At the top of the figure is a piece struc. The coarse dashed line coming out of it indicates that the piece struc has a pointer to the array seen at the right of the figure. The solid arrow below the piece struc indicates that the struc is flowing into the `performScheduling` method.

A fine dashed line coming out of an object and pointing to a method name indicates a method call invoked on that object. However, `performScheduling` has no fine dashed line pointing to it because it is statically invoked.

The thick vertical line below `performScheduling` indicates all the activity that occurs "during" the method call. The piece struc at the top of the thick vertical line is the piece struc that flowed into the method-call. At the top, the first activity of the `performScheduling` method is to invoke the `Divide` method on the piece struc. The solid arrow coming out of the `Divide` method indicates the flow of return values coming from it. These return values are a number of newly created piece strucs.

The next activity of the `performScheduling` method is to call the `Kernel` method on each of those returned piece strucs. This causes the data to be transformed. The medium-dashed line coming out of each `Kernel` method indicates that the `Kernel` has modified a portion of the array by side-effect.

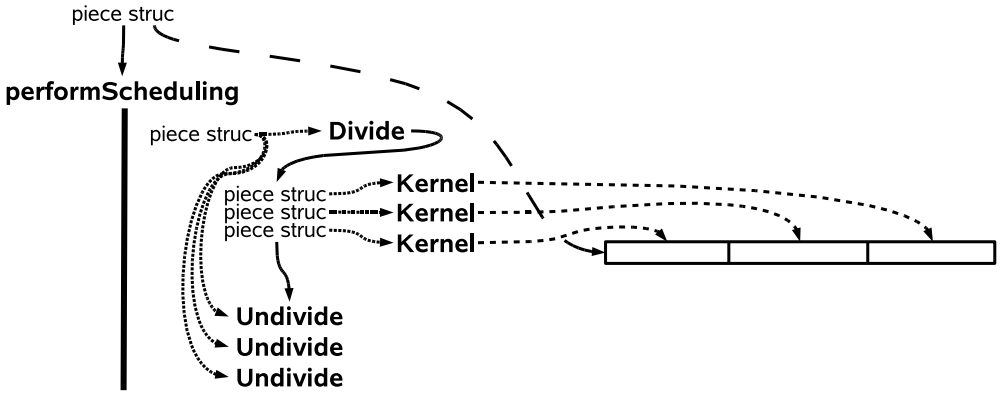

Figure 2. A representation of the call structure plus data structure for the DKU pattern implemented in an OO paradigm. Solid lines represent data flow, coarse dashed lines represent pointers, medium dashed lines represent side effects, and fine dashed lines represent method calls invoked on an object. The thick line below "performScheduling" demarcates the activity that takes place during the method call.

Upon completion of the `Kernel` method, each piece struc flows into the `Undivide` method. The `Undivide` is invoked on the parent of the piece strucs. When all sub-pieces have completed the `Undivide`, the `performScheduling` method is complete.

2.2 DKU pattern in Java

As a concrete example of the DKU pattern, consider the implementation of the DKU pattern in Java illustrated in Figure 3. Here the `class` construct is used to define the data-structure that represents one piece of work, which is called the `MyPiece` class. It implements the `DKUPiece` interface, and has three methods:

1. `divideSelfInto_SubPieces`
2. `performKernelOnSelf`
3. `unDivideASubPiece`

The `divideSelfInto_SubPieces` method uses the `new` keyword to create new sub-piece objects which it places into an array. Although it is not shown in the figure, the DKU pattern includes a standard iteration interface, so one can simply say "myPiece.nextSubPiece". This interface iterates through the sub pieces array.

The data to be worked on is held in this example in an array. Each sub-piece is given a portion of the array to work on. To divide up the work, each sub-piece is given a starting position in the original array, and and ending position. The start and end indexes actually serve two purposes: they mark boundaries in the data, and they also divide the iteration space of the Kernel.

These two purposes are not always so cleanly met. Some uses of the DKU pattern must separately bound data and bound iteration sub-spaces. A good example is in searching a graph. One strategy would be to start the search in the divider, but only go to a certain depth, then save the position. The divider would place one saved position in each sub-piece. The kernel would start from that saved position. In this case, the data is actually the same for all sub-pieces, only the iteration space is different.

The `performKernelOnSelf` method performs the work. In the example code, it has a simple FOR loop. The loop begins at the start-index that is saved in the sub-piece and continues until the end-index saved in the sub-piece. At each index, it performs some work. The result is saved in-place by overwriting the input value. Thus, the original matrix is updated in-place by side-effect. If additional storage is needed for the result, it would be allocated inside the divider, and pointed to in the sub-piece.

A class that implements the DKUPiece interface:

```
public class MyPiece implements DKUPiece
 {
   ADataType[]    origArrayToBeProcessed;

   int       startingIterationValueForThisPiece;
   int       endingIterationValueForThisPiece;

   MyPiece[] subPiecesArray;   //standardized name, holds children pieces

   //  Divider is called by the scheduler
   public void divideSelfInto_SubPieces( int numPieces )
    {
      for( newPiecePos = 0; newPiecePos < numPieces; newPiecePos += 1 )
       {
         newPiece = new MyPiece( this );  //constructor copies pointer to
                                          // origArrayToBeProcessed
         newPiece.startingIterationValueForThisPiece =
            calcNewStart( newPiecePos );  //the index into orig array at
                                          // which new sub-piece starts
         newPiece.endingIterationValueForThisPiece =
            calcNewEnd( newPiecePos );    //the index into orig array at
                                          // which new sub-piece ends

         subPiecesArray[ newPiecePos ] = newPiece;  //add new sub-piece to
       }                                            // children array
    }

   //  Kernel is called by the scheduler
   public void performKernelOnSelf()
    {
      for( currIndex = startingIterationValueForThisPiece;
           currIndex < endingIterationValueForThisPiece;
           currIndex += 1 )
       {
         aDatum = origArrayToBeProcessed[ currIndex ];
         doWorkOn( aDatum );  //do the work on one elem of original array
                              // the result over-writes the original elem.
       }
    }

   //  Undivider is called by the scheduler
   public boolean unDivideASubPiece( MyPiece aFinishedSubPiece )
    {
      incrementNumSubPiecesUndivided();  //only have to keep track of
    }                                    // whether all sub-pieces are done
```

A code snippet that invokes the scheduler. This initiates the DKU activity.

```
   scheduler.processDataIn( myPiece );  // invoke on top-level root piece
```

An implementation of the scheduler. This code is replaced by a specialized version that is written for specific hardware. The hardware specific version creates worker threads, implements communication between them, and synchronization. For single hardware thread machines, this is left as-is.

```
public class Scheduler
 {
   public static void processDataIn( DKUPiece rootPiece )
    {
      rootPiece.performKernelOnSelf();
    }
 }
```

Figure 3.

The `unDivideASubPiece` method is invoked on the parent piece object, and given one of the sub-pieces created by the `divideSelfInto_SubPieces` previously called on that parent piece object. In the example, the result has replaced the input, so the undivider has only the minimum work to do, which is tracking that all sub-pieces have completed.

To illustrate the undivider in the graph search example, the undivider would look at the results from each piece, and check if any of them was successful in the search. This is how the individual search results would get combined into a single over-all search result. Future work might add iteration-control syntax that would allow the undivider to communicate with the scheduler. This would allow all sub-pieces to be stopped once one of them found a match.

The Scheduler controls all actions; the only way for an application to make anything in a DKU pattern take action is through the `Scheduler` class. This has some benefits; for example it enables a very simple specialization technique in which the `Scheduler` class is simply replaced by a hardware-specific version.

The pattern defines names that must be present in an instance of the DKU pattern. The DKUPiece interface defines standard method names. In addition, many instance variables are required to have standard names and uses, such as: `subPiecesArray`, `numSubPieces`, and `numSubPiecesFinished`. Finally, there must always be a `Scheduler` class, which is the only class that invokes any of the standard DKU methods (outside of the classes in the DKU pattern itself).

Having fixed names enables a specializer to identify the DKU pattern within the source. A specializer can perform more in-depth optimizations this way. For example, it could collapse data-structures passed to the Kernel down onto a flat array, thereby enabling advanced optimization techniques.

2.3 Specialization

What makes the DKU pattern interesting is the ability to automate back-end specializations. Each specialization transforms the source and compiles down to efficient code for specific hardware. Specialization can be performed for many classes of hardware, including shared memory and distributed memory machines, and even for new generation GPUs.

The ability to run high performance across a variety of hardware platforms relies on the fact that the behavior of the Pattern itself changes for each kind of hardware. It also relies upon the semantic information the Pattern conveys.

The high performance is accomplished by extending the split-compilation model. In this model, a front-end compiler performs all syntactic and grammatic checks, and accumulates all libraries and linked code. Meanwhile, a back-end compiler, or "specializer" transforms the code to be efficient on particular hardware. The source developer uses the front-end compiler, while some other entity performs the specialization.

Each specializer is written for one specific hardware platform, and uses techniques that enhance performance on that kind of hardware. For example, for GPUs, a custom Scheduler class is written that performs the division on the host processor. The custom scheduler includes the code that controls the GPU hardware, loading it with the Kernal and pointing it to the array of piece data-structures.

The specializer we wrote for our proof of concept implementation includes a Scheduler class and worker-thread classes. The scheduler creates one worker for each hardware thread during intialization. When the scheduler is invoked to perform work, it is handed a DKUPiece and calls the divider on that piece, telling it to make the same number of pieces as the number of worker threads. The scheduler then hands the pieces to the worker threads. The worker threads hand the completed pieces back, at which point the scheduler calls the undivider on the parent of the piece until all pieces are complete.

The specializer script inserts the number of hardware threads into the Scheduler class before swapping the specialized scheduler in place of the original one. This simple mechanism, swapping directories, to perform specialization is shown, in the results section, to be effective.

2.4 DKU Infrastructure

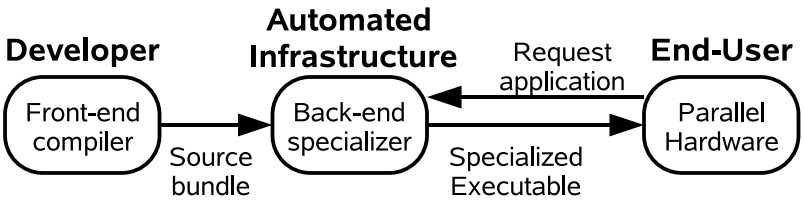

Figure 4. Tool flow of an application

The tool flow of an application is depicted in Fig 4. It shows the application developer using the front end compiler to create a source bundle; this feeds the automated infrastructure that specializes the source; this sends the appropriate executable to the hardware of a user who requests an app.

The DKU pattern relies upon infrastructure to perform specialization. The DKU pattern was designed to work this way, in tandem with automated specialization, so it provides the semantic information that advanced specialization and scheduling needs. It also is designed to make automated specialization convenient.

The infrastructure's purpose is performance portability. The DKU pattern itself performs no actions, and its semantics are non-functional; they are performance-related semantics. Thus, the DKU pattern has no value without the infrastructure that takes advantage of what the pattern offers.

Performance portability means specializing a single source to multiple hardware platforms. This, of course, requires hardware knowledge. The three situations in which hardware knowledge is present are:

1. During distribution to the end-user
2. During installation by an end-user
3. During the run of the program

All three situations are good candidates for places to perform specialization. We have chosen, for our proof of concept, to implement the first choice, with a web-based infrastructure.

2.5 Implementation

Our proof-of-concept implementation includes a matrix multiply program written, in Java, according to the DKU pattern, as well as web-based infrastructure. The infrastructure is a website[8] that receives a source tar ball, specializes it, then distibutes .jar executables.

The matrix multiply program follows the DKU pattern for Java, which stipulates a structure for the source directory tree. In this structure, the application developer places all DKU infrastructure related classes in a single directory, named "DKU". Hence the DKUPiece interface and Scheduler class files are placed in this directory. This is a source-code requirement that our implementation relies upon.

The web infrastructure has a web server and many scripts. One set of scripts receive source and specialize it, a separate script distributes the resulting specialized executables. The specialization is performed by a set of scripts with one script for each hardware platform. Each specialization script comes with a DKU directory that contains a Scheduler class and helper classes that are written specifically for one hardware platform.

The Receiver script, upon receipt of new source, calls all the specialization scripts. Each specialization script swaps its custom DKU directory for the original one, then calls the Java compiler. This is the essential step that modifies the behavior of the code. The .jar resulting from compilation is handed back to the receiving script, which moves it to a staging area and adds information about it to a "database" text file. The information will be used by the Distribute script and includes the program name, the .jar file name plus path and which hardware platform the .jar file is a specialization for. This completes the receive and specialize process.

Meanwhile, clients request programs they want to install and run. Each request identifies the hardware platform of the client and the desired program's name. The Distribute script in the web infrastructure is invoked when the server receives such a request. The script looks in the text database file to find the appropriate .jar file. It uses the .jar name and path info to get the .jar file, then sends it to the client.

3 Setup and Method

We evaluate the DKU pattern in unison with its web infrastructure, which performs automatic specialization. The evaluation is performed on the Matrix Multiply algorithm, for which a single high performance DKU version is written in Java 1.6. This one program is then automatically specialized, using the web infrastructure, to 2 versions, a serial version and an 8 threaded version that uses Java 1.6's (relatively) high performance concurrency constructs.

Both versions are run on an 8 hardware thread machine containing 2 quad core Xeon E5345 processors at 2.33GHz, and on a single hardware thread Centrino processor at 1.5GHz. Timing is collected via Java's nano-second precision timer plus printf.

4 Results

To show that the automated specialization successfully tunes parallelism to hardware characteristics, we run our DKU-ized Matrix Multiply program, in Java, on two different machines. One has a single hardware thread, the other 8 hardware threads. We run on a selection of matrix sizes, and plot the resulting running time and estimate the time lost due to parallelism overhead. The overhead is spent inside the JVM and OS's implementations of threads, communication primitives, and synchronization.

The overhead of using the DKU pattern is not reported, because it is too small to accurately measure. The DKU pattern adds only a single extra method-call over a non-DKU version, when specialized to a single threaded processor. This extra call happens when the single-thread scheduler is called, which does nothing except in-turn call the kernel. It calls the kernel on the entire input data-structure, so no division or undivision is performed, so the overhead is just the unneeded call to the scheduler. On the system we have available for testing, the overhead of this method relay is too small to separate from noise.

What we do measure is the effectiveness of our specializations. Because the DKU pattern only makes semantic information available, it has no intrinsic performance. A specializing back-end compiler must be written that uses the semantic information. It is the performance of this specialization that is being measured. In essence, the DKU pattern has less structure than an extension to a language would impose, being merely an organization principle that makes semantic information available. It is the effectiveness of *using* the semantic information that the tests measure.

4.1 The Numbers

Number of Thds Specialized to	9 x 9 t in msec	81 x 81 t in msec	162 x 162 t in msec	324 x 324 t in msec	648 x 648 t in msec	1296 x 1296 t in msec
1 Thread	0.28	28	31	291	2,270	19,100
8 Thread	1.4	35	16	38	275	2,400

Table 1. To show that the specialization works, on an 8 hardware-thread machine, this table shows the running time for several sizes of matrix. Two different specializations of the same Matrix Multiply source are run. Shows the minimum time, chosen from among ten runs.

The results on the eight hardware-thread machine, seen in Table 1, show that on large matrices, the version specialized to eight threads performed 7.9 times faster than the version specialized to a single thread. This indicates the success of the automatic specialization scheme.

Number of Thds Specialized to	9 x 9 t in msec	81 x 81 t in msec	162 x 162 t in msec	324 x 324 t in msec	648 x 648 t in msec	1296 x 1296 t in msec
1 Thread	0.11	8.7	56	447	3,450	30,700
8 Thread	3.7	57	102	403	3,860	31,100

Table 2. To show that specialization has value, and to estimate thread overhead, on a single hardware thread machine, this table shows the total running-time on two specializations of the Matrix Multiply program. Results are given across several matrix sizes, for a specialization to a single thread and a specialization to eight threads. Shows the minimum time, chosen from among ten runs.

The results on the single hardware thread machine, seen in Table 2, show that the eight thread version is now slower (with one anomaly). The slowdown is due to the OS's thread scheduler overhead and the JVM's synchronization primitive overhead. There may also be memory hierarchy effects at work.

The net effect is that if one simply wrote an eight threaded version, it would run slower on a single hardware-thread machine. Conversely, if one simply wrote a single threaded version it would run slower on an eight threaded machine. It is therefore valuable to have the automated infrastructure that generates both versions and sends the single threaded version to the single hardware-thread machine and sends the eight threaded version to the eight hardware-thread machine.

4.2 Proposed OS modification for higher performance

The DKU pattern can potentially save the time lost to synchronization. The large discrepancies on small matrix sizes on both machines are due to thread overhead. Most of this time is lost in the implementation of the JVM and OS.

DKUs semantics enable replacing the OS's thread scheduler with a more efficient scheduler. The DKU pattern encodes independent, atomic, units of computation. Each of these atomic units of computation can be scheduled directly onto a hardware thread and allowed to run to completion. Following this notion, the OS's "blind" thread scheduler would be replaced by a very simple DKU-aware scheduler.

The DKU scheduler would assign atomic units of computation directly to hardware threads. To start a computation, it would load one register with a pointer to a piece-struc then jump to the Kernel's code (rather than a call instruction). When the computation finished the Kernel would jump directly to the scheduler. The scheduler would already have the pointer to the piece-struc, and so it has all the information the Undivider needs. Hence, no register saves nor restores would be performed, no stack manipulations, and most importantly, no synchronizations would be performed. The times seen in Tables 1 and 2 suggest how much improvement could be realized by using this scheme over using the "blind" thread scheduler in the OS.

If specialization were placed into the OS, then such a DKU aware scheduler could be used in place of a blind thread scheduler. The OS would have to add a command to explicitly perform installation of new programs. The installation command would call the specializer, which would instrument the Kernel to end with a jump to the scheduler. The scheduler would be a permanent fixture of the OS kernel.

The DKU scheduler, being inside the OS, would take direct control of hardware threads. This would eliminate the blind thread-scheduler's needless-for-DKU decision making, eliminate register saves and restores between Kernel calls, eliminate stack manipulations, and eliminate synchronization operations, all of which are unneeded for DKU programs. The result would be reduced running time on a given matrix size and a much smaller break-even size.

5 Future work

In future work we will attempt to implement a specializer for MPSoC chips with distributed memory. We plan to implement on an MPSoC from ST that is targeted in the OMP project. This specializer will implement its own hardware level thread scheduler, which will take control away from the OS's thread implementation.

We also plan to add a running-time estimator, in which the application programmer or profiling tools indicate the running-time complexity of the kernel. Executables will use this at runtime to decide the number of pieces to divide data into, thus adjusting for break-even data size.

Finally, we plan to introduce custom syntax embedded within comments to help when specializing to distributed memory hardware, and to aid in collapsing data-structures down to flat arrays. This will be especially helpful for object oriented programs, which inherently use extensive indirection. The synatax should allow the elimination of many of the pointers, while the code is still written in a natural object oriented way.

6 Conclusion

We have proposed a pattern that is embedded into any language, to express data parallelism. The pattern enables an automated tool that specializes source code to multiple hardware platforms. A given specializer works for all source code that follows the pattern. Source that follows the pattern works with all specializers. When new hardware is introduced a specializer is written for it, and all of the "dusty decks" are run through the new specializer, making all the pre-existing sources run high performance on the new hardware.

The source is written by application people, while the specializers are written by hardware people. This cleanly separates application knowlege from hardware knowlege and makes a single source run high performance on all hardware that has a specializer. To coin a phrase, "write once, run high performance anywhere".

7 References

[1] CILK homepage. http://supertech.csail.mit.edu/cilk/.

[2] Titanium homepage. http://titanium.cs.berkeley.edu.

[3] Philippe Charles, Christian Grothoff, Vijay Saraswat, Christopher Donawa, Allan Kielstra, Kemal Ebcioglu, Christoph von Praun, and Vivek Sarkar. X10: an object-oriented approach to non-uniform cluster computing. In *OOPSLA '05: Proceedings of the 20th annual ACM SIGPLAN conference on Object oriented programming, systems, languages, and applications*, pages 519–538. ACM, 2005.

[4] M Cole. *Algorithmic skeletons: Structured management of parallel computation*. Pitman, 1989.

[5] J. Darlington, A. J. Field, P. G. Harrison, P. H. J. Kelly, D. W. N. Sharp, and Q. Wu. Parallel programming using skeleton functions. pages 146–160. Springer-Verlag, 1993.

[6] Kayvon Fatahalian, Daniel Reiter Horn, Timothy J. Knight, Larkhoon Leem, Mike Houston, Ji Young Park, Mattan Erez, Manman Ren, Alex Aiken, William J. Dally, and Pat Hanrahan. Sequoia: programming the memory hierarchy. In *SC '06: Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, page 83. ACM, 2006.

[7] Dominique Ginhac, Jocelyn Serot, and Jean Pierre Derutin. Fast prototyping of image processing applications using functional skeletons on a mimd-dm architecture. In *In IAPR Workshop on Machine Vision and Applications*, pages 468–471, 1998.

[8] Sean Halle and Albert Cohen. DKU infrastructure server. http://omp.musictwodotoh.com.

[9] Ken Kennedy and John R. Allen. *Optimizing compilers for modern architectures: a dependence-based approach*. Morgan Kaufmann Publishers Inc., 2002.

[10] P Palatin, Y Lhuillier, and O Temam. Capsule: Hardware-assisted parallel execution of componentbased programs. In *In Proceedings of the 39th Annual International Symposium on Microarchitecture*, pages 247–258, 2006.

[11] R. Stephens. A survey of stream processing, 1995.

[12] Wikipedia. MPI wikipedia page. http://en.wikipedia.org/wiki/Message$_{\text{P}}$assing$_{\text{I}}$nterface.