A Framework to Support Research on Portable High Performance Parallelism

SEAN HALLE
UCSC and INRIA Saclay
seanhalle@yahoo.com

DMITRY NADEZHKIN
Univ of Leiden
nadezhkin@gmail.com

ALBERT COHEN
INRIA Saclay
albert.cohen@inria.fr

Abstract

There is an increasing need for a framework that supports research on portable high-performance parallelism. Such a framework would facilitate two main goals: discovering how to separate application-specific code from hardware-specific code, and supporting research on parallel schedulers and optimizations. The framework would be agnostic to language, however the knowledge gained might inform future language research with better understanding of the boundary between application and hardware.

We propose a first step towards such a framework; to separate application code from hardware specific code, we define a "bi-directional" library interface that has both, library functions implemented by application-code, and library functions implemented by hardware-specific code. We also make the scheduler a first-class entity that is called as a library function. We provide a sample implementation of the framework that can automatically link the two directions and generate executables, from the same source, for each platform being investigated.

To support research on run-time schedulers we provide a pool of instrumented applications and our implementation of the framework to plug research schedulers into. The interface makes an instrumented application appear to the scheduler as a black box that has a number of "knobs" to manipulate, via the interface.

Optimizations that perform Code Structure Transforms are also supported by the application pool and our sample implementation. The researcher is free to use their preferred language to write command-line tools to perform the transforms. The framework would invoke the plugged-in tools on the suite of applications and generate ready-to-run executables for the target platform.

We detail our first installment, which focuses on data parallelism, stating the interface defined so far. We describe the process of instrumenting a set of three applications, Hamiltonian Path (an NP-Complete problem) in Java, H264 Deblocking in C, and Matrix Multiply in Java. We describe the process of implementing, then plugging-in run-time schedulers for three hardware platforms: Multi-core, the Cell processor, and a heterogeneous collection of multi-core. We also share details of our sample implementation that transform tools and run-time schedulers can plug into.

1 Introduction

Solving the compound problem of making it easy for application experts to remain isolated from most hardware architecture details, while writing a single source that runs high performance across widely varying parallel platforms may be larger than a single research group can reasonably handle. A collaborative approach is needed where it is simple and natural for many different groups to collectively accumulate incremental contributions. Some framework is needed to support such collaborative accumulation of research from loosely connected groups working to solve the Portable High Performance Parallelism (PHPP) problem.

By definition, PHPP requires a single source to run high performance on many hardware architectures. Therefore that one source should not encode information specific to one hardware architecture. Hence the essence of solving the PHPP problem is identifying the boundary between application-specific information and hardware-specific information, if one does indeed exist. Once this boundary is identified, then it can be encoded in languages, development tools, and intermediate formats.

An explicit embodiment of the boundary is effectively the "plug" by which a single source is connected to multiple hardware encodings. The existence of such a plug provides the portability feature. The high performance will come from getting the shape of the plug (boundary) right. All communication between hardware and application goes through it, so the plug must enable the operations needed for high performance on the hardware. Some of those operations take place during the process of plugging the two together, while other operations take place during execution of the application.

The process of searching for the boundary involves trying different plug shapes. One popular approach to the search has been to define new languages [25][24][23][10][20][16][18][13][7] and language extensions [15][21][2][26][9][3] which express application information, while hardware information is embedded underneath in the language implementation, to the extent possible. This is a perfectly valid and useful approach. However, it has proven frustratingly slow, and tends to isolate groups, making collective contribution to the search more difficult.

A fair number of other approaches to searching for the boundary have been tried: skeletons[17][14], meta-programming[22], parallelising compilers[8][5] among others. An important effort has been to develop parallel implementations of common programming patterns[11][1][19]. Again, progress has remained slow and research groups fairly decoupled.

A collaborative approach to speed up the search will need an easily changed boundary embodiment. It should keep past plug-shapes working when introducing new proposed extensions or modifications. It should also automate the generation of executables, for testing. But most valuable, perhaps, will be enabling focused research, freeing researchers from the burden of implementing infrastructure, such as by providing a clonable embodiment of the accumulated effort. Individual groups can replicate the most recent version then modify only the one portion their research focuses on. That embodiment should also be easy to modify.

To get a start on such a framework, we propose two patterns: first, the Bidirectional Libraray Interface plus Specialization (BLIS) pattern, and second, the pattern of making schedulers first-class entities in the interface. The bidirectional library pattern decouples application from hardware, while the specialization performs the plugging of each application into hardware encodings. This: makes it relatively easy to accumulate contributions; makes it easy to replicate the accumulation; and makes it easy to modify only the portion a group wishes to do research on. Making schedulers first-class entities is our best guess at a starting embodiment of the boundary, and has yielded the results we share in this paper.

Our first step is for data-parallelism only. We have defined and implemented an interface, instrumented several applications, implemented schedulers for several hardware platforms, and implemented infrastructure that performs some automation. In particular, the infrastructure will automatically specialize any instrumented Java application to run on a shared memory multicore machine, and produce a runnable jar file specialized to the target hardware's number of threads.

In order for this first step to be convincing, we must: demonstrate an array of applications written to the interface, demonstrate that a number of schedulers for different hardware can be written to the interface, and demonstrate that it is easy to link the two. We also must describe the interface, describe our framework implementation, and the process of plugging into it applications and schedulers, with sufficient detail to allow the reader to asses the level of convenience of using the framework.

To support the notion that it is worth taking a look at this first step, we share that we believe that the forms of parallelism supported can be grown to include: stream (pipelining) parallelism; functional-block (component) parallelism; and parallel-library-implementation parallelism. We also believe that it will be possible to support fully general communication among parallel work units, atomic updates of shared data structures, parallel distributed I/O, nested levels of parallelism, and prediction of computation and communication costs, via the bidirectional interface approach.

The growth process envisioned allows a given application to remain compatible with older as well as newer schedulers, insulating the source code from changes in the interface. New applications can implement additions to the interface as needed, and will work with older schedulers, which will automatically ignore the interface additions. Hence to get started on a new application, one is free to implement only a minimal sub-set of the interface, then add extensions as needed for additional performance. Likewise, application developers and hardware specialization implementers are both protected from changes in the interface as the search of the boundary progresses.

In Section 2 we describe the big picture of the framework, showing each of the pieces and how they fit together. In Section 3 we describe the interface defined for data parallelism. In Section 4 we describe the three sample applications and how each was instrumented with the interface. In Section 5 we describe the automated infrastructure. In Section 6 we describe the details of the specializers and run-time schedulers in them. In Section 7 we give brief execution-time results demonstrating successful use of the framework for the three hardware platforms and the three applications. Section 8 concludes the paper.

2 Big Picture of the Framework

The "Bidirectional Library Interface plus Specialization" (BLIS) framework is centered around the BLIS Interface, which in turn is based on the notion that scheduling is at the center of parallel execution. The interface makes the scheduler a first-class entity that is directly communicated with by application code. For this paper we define scheduling to consist of three parts: choosing work-units, choosing resources to perform each work-unit, and choosing the timing of starting the work. A work unit is a tuple of a code-snippet, bookkeeping data, and work data. The code snippet is controlled by the bookkeeping data causing it to transform the work-data. Hence bookkeeping data consists of things like iteration-space bounds, while work-data is an ancestor of the computation output.

The BLIS Interface is a set of bidirectional libraries (biLib). Each high level pattern of parallelism can have its own biLib, centered around a scheduler for that kind of parallelism. The first biLib is for data parallelism (Sec 3), while future biLibs will be for stream parallelism, function-unit (component) parallelism, and Library Parallelism (a library of common patterns that have hand-tuned parallel implementations).

Fig 1 shows how a bidirectional library works. It is split into two entities: an application, and a hardware-specific directory. The application implements library functions that things in the hardware directory call. Meanwhile the hardware directory implements library functions that the application calls. The only interaction between the two is via library calls.

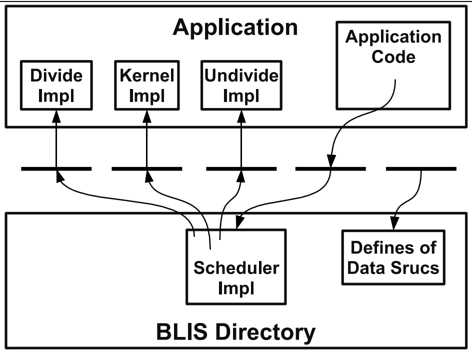

Figure 1. How a bidirectional library works. The application, above, implements app-lib functions, while the hardware directory, below, implements HW-lib functions. The thick lines in the middle represent the interface itself. To run an application on a target platform, the hardware directory for the platform is paired with the application (then, typically, compiled).

Thus, either side can be replaced with a different set of implementations of the library calls and the combination still compiles. In our implementation of bidirectional libraries, the hardware directory is named the BLIS directory.

To support research on PHPP, we chose to make a sample implementation of a framework that automatically performs Specialization. For application development, we made the framework as "turn-the-key" as possible, allowing an application developer to write an application then "press a button" to get an executable for their target machine. Meanwhile, for researchers who wish to focus on one aspect of the PHPP problem, such as Code Structure Transforms (CSTs) or scheduling implementations for specific hardware, we have tried to make it easy to "plug in" such research to the sample infrastructure.

Fig 2 shows the tool chain of our sample BLIS framework implementation. We describe the path of an application passing through the tool chain. First, applications are developed, compiled, and debugged in a sequential environment. The application developer is given the BLIS directory as part of the framework. It contains serial implementations of all the hardware function calls, such as the scheduler call, and definitions of standard data structures.

When an application is complete, its source code is sent to the sample Specialization Server[6], where it is first saved for future use, then sent to each specializer in the server. A specializer is custom to a single hardware platform, and produces an executable, or set of executables, that run native on that platform.

The executables produced by the specializers inside our Specialization Server are saved and later retrieved by the hardware on which they run. To retrieve a stored executable, we supply some form of BLIS client that connects to the Specialization Server and requests the application. The server automatically sends the correct executable for the hardware, and the client makes that executable available to be run.

The hardware-specific side of the BLIS interface is driven by development of specializers. Through this process, the interface will explore the boundary, collecting the most essential "knobs" that schedulers require, and discovering the most essential information the schedulers need from the applications.

In practice, a research team can clone the Specialization Server and modify the local copy. This gives them a fast, responsive development cycle and supplies a ready, reusable pool of applications copied-over inside their clone. The various research teams may modify the applications and the BLIS interface in their local Specialization Server clones. As they discover a need to extend the BLIS interface, or a need to modify an existing library call definition, they can do so. When the team publishes, they can propose that their modifications be adopted into the central Specialization Server.

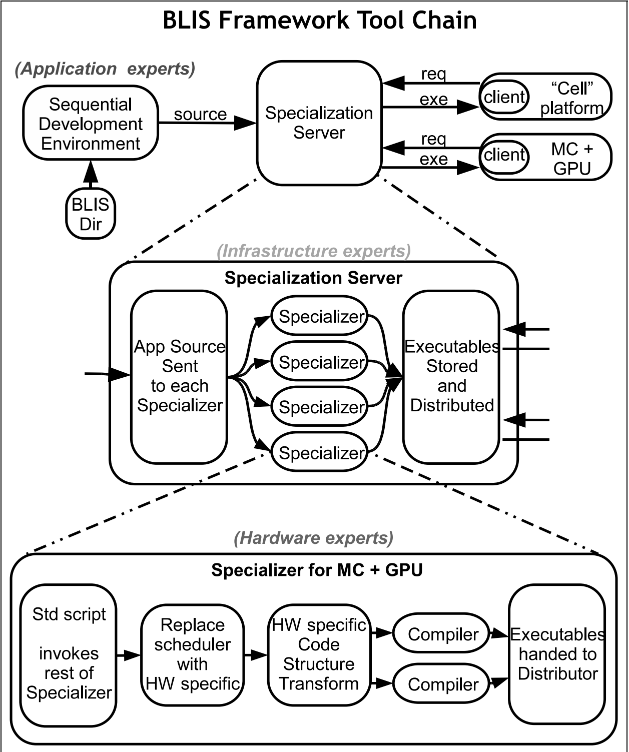
BLIS Framework Tool Chain
(Application experts)
Sequential Development Environment
source
Specialization Server
req
exe
client
"Cell" platform
req
exe
client
MC + GPU
BLIS Dir
(Infrastructure experts)
Specialization Server
App Source Sent to each Specializer
Specializer
Specializer
Specializer
Specializer
Executables Stored and Distributed
(Hardware experts)
Specializer for MC + GPU
Std script invokes rest of Specializer
Replace scheduler with HW specific
HW specific Code Structure Transform
Compiler
Compiler
Executables handed to Distributor

Figure 2.

The tool chain of our sample embodiment of the BLIS framework, with breakouts showing what is inside certain elements. It is designed to encourage general researchers to write applications to the BLIS standard, and to make focused research on one aspect of PHPP convenient. ("MC + GPU" stands for multi-core plus GPU)

This streamlines the process of collaborative sharing of work, collecting the best, in the manner of open source cooperative development.

Such a process will expand experience with application developers who use the BLIS interface, allowing the most convenient form of the BLIS interface to be discovered. Researchers focused on making the BLIS interface convenient for application developers will also have the needs of the hardware-specific code available. They can explore various ways to present to the application the needs of hardware developers by proposing new forms of the BLIS interface, and validate the value of the proposals against the collective experiences of application developers across domains. Because such work will involve changing existing interfaces, it will progress more slowly than specialization research, but more rapidly than the evolution of new languages.

3 The DKU Interface for Data Parallelism

"DKU" is the name of the BiLib (bidirectional library) we propose for enabling portable data-parallelism. It stands for "D"ivider, "K"ernel, "U"ndivider, which are the main library functions that the application implements. The functions are made available to the DKU-scheduler, which uses them to divide work into smaller pieces, execute the kernel on each piece, then collect the individual results into the larger result. The DKU pattern's programming model is shared memory with results communicated by side-effect. However, the shared-memory restriction is lifted by the use of a "bundling quad" interface extension described in Sec 3.5.

When the application's control flow reaches the data parallelism, it calls the scheduler. Normally, at that point, a sequential application would execute a loop nest to process the data. When DKUized, the loop nest is wrapped and becomes the Kernel, while the data that would go to the loop nest is handed instead to the DKU-scheduler.

The scheduler has a choice of whether to exploit the parallelism, or simply pass the data to a serial kernel. A serial kernel is effectively the original sequential loop nest. It takes the data in the application's native form, and processes it sequentially, avoiding overhead.

Figure 3 shows the two cases. On the left, the scheduler decides the data is too small to be profitable and calls the SerialKernel. When the SerialKernel returns, so does the scheduler; the results are communicated by side-effect

On the right, the scheduler decides it is profitable to process the data in parallel and hands it to the Dividable Piece Maker, which packages the data into the DKU pattern's standard data structure. The Dividable Piece Maker is necessary for two reasons: being a set of library-functions, the D, K, and U must have a standard interface, so the data has to be packages into a standard data structure; and there may be dependencies within the data that must be accomodated.

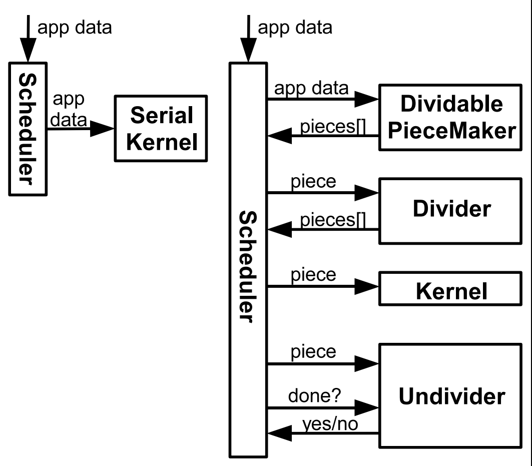

Figure 3.

The call structure of the DKU interface. On the left is the case when the scheduler chooses not to exploit the parallelism. On the right is the case when the scheduler does.

Note that the Dividable Piece Maker turns application-format data into DKU pieces. It is the bridge from application to DKU pattern.

Specification of DKU Java Interface

extend the **DKUPiece** base class
override the methods:

```
performKernelOnSelf()
divideSelfInto_SubPieces( int numPieces )
unDivideASubPiece( DKUPiece subPiece )
```

The Kernel may only access data from instance vars
The Kernel may use pointers to shared data-structs
The Kernel returns results via side-effect
The Kernels in distinct instances must not communicate with each other by side-effect
The Divider creates new instances and places them into the `subPieces[]` array that is in the base class
The Divider fills new instances with values that control the work they perform
The Divider sets the control values based on its own instance's control values (allowing repeated division)
The Divider must carefully choose control values that prevent side effects and distribute all of its instance's work among the sub-pieces
The Divider does its best to create numPieces, but may create any number from 1 up
The Undivider default implementation tracks the completed sub-pieces
The Undivider rarely needs to be overridden

Sidebar:

Above is the minimum set of rules to follow to instantiate a valid DKU instance in a Java application (the Kernel is the implementation of the `performKernelOnSelf()` method, the Divider is `divideSelfInto_SubPieces`, and so on).

The Dividable Piece Maker accepts application specific data and returns the largest freely-dividable work-units (pieces of work) that it can, while still respecting the dependencies. The work-units returned from the Dividable Piece Maker are scheduled in-order, which guarantees that all dependencies in the data are respected.

For each, the scheduler decides how many work-units to sub-divide it into, based on a hardware-specific algorithm, and tells the Divider to perform the division. The scheduler then hands each sub-piece to one of the instances of the Kernel, all of which run in parallel. When a Kernel finishes, the scheduler passes the sub-piece along to the Undivider. When the Undivider indicates that all the sub-pieces are done, the scheduler moves on to the next root piece, until all are complete, then the scheduler returns.

Note that the scheduler is a library function, called by the application. Meanwhile the D, K, U, Dividable-PieceMaker, and SerialKernel are also library functions, called by the scheduler. The scheduler is a black box to the application, while the application is a black box to the scheduler. This generic-in-both-directions is what allows connecting all combinations of applications with scheduler-implementations. The fact that all interactions are through library calls is what ensures that compilation is error free for all combinations.

3.1 The Divider

The D is perhaps the most interesting part of the DKU interface. This is where the strategy of how to parallelize a section of the application is mainly implemented. Often, the only thing divided is the iteration space; a nest of FOR loops is identified and becomes the Kernel; then the nest is modified so that the iteration variables' start and end values come from a DKU-piece data structure.

As a result, the process of dividing is reduced to placing start and end values into DKU-pieces; each piece gets a different portion of the original iteration range. When a K receives such a DKU-piece, it sets the start and end values in each of its loops from the piece, then performs the loops. Each piece has a distinct set of start and end values, so each K performs a distinct portion of the work. The application programmer ensures the pieces are independent. The independence restriction will be lifted in the next release of the DKU biLib, which will include inter-piece communication.

This basic technique of identifying loop nests, then dividing the loop-bounds among DKU pieces was used in both dense matrix multiply and H264 deblocking. However, Hamiltonian Path was included in the application set to demonstrate more creative opportunities for using the DKU biLib. There, the divider turns the control flow into data, then divides that "control" data. Section 4.3 gives details of how this was done.

An important point is that the divider is not required to produce the number of pieces the scheduler asks for. It simply does its best. The scheduler always checks how many pieces it actually got back and proceeds accordingly.

3.2 The Kernel

The K is pure application code. It is normally a loop nest that has been slighly modified such that the ranges of its iterations come from values taken from a DKU piece structure. During the DKU-ization process, the Kernel code normally has only minimal changes from the pre-DKU code. Often only a wrapper is required.

The rules for a Kernel are straightforward: all variables must be local to the kernel, all data touched is reached via the DKU piece that is passed to the kernel (pointers are allowed), and all results are reachable from the DKU piece. Isolating the kernel from the environment in this way allows it to be run in remote memory spaces without modification.

Kernels are allowed to work on shared data-structures, and to have loop-carried dependencies. The complications are handled by the Dividable Piece Maker, described in section 3.4, which is tasked with identifying independent pieces within the data/iteration-space. This flexibility expands the number of applications that can successfully use the DKU interface.

In C, the Kernel has the additional restriction of using only data-types defined in the BLIS-supplied headers for DKU. "int" becomes "int32" and so forth, to ensure data sizes are the same (except for pointers) in both local and remote memories. Data-structures used in the Kernel cannot have align statements in them (these will be supplied by the specializer if they are needed for the target hardware). Finally, the bundling quad (Sec 3.5) must handle changes in data-structure sizes that are due to differences in pointer sizes.

3.3 The Undivider

The U is usually the simplest of the three. In many cases, it simply acts as a barrier, counting the completed sub-pieces until all have been accounted for. However, sophisticated specializers that perform source-to-source transforms are free to analyze the application code and substitute more relaxed synchronization. The DKU interface identifies for such a specializer where the barrier is located in the code (the U), and where the dependencies among root pieces are encoded (in the Dividable Piece Maker).

3.4 The Dividable Piece Maker

The Dividable Piece Maker packages application-specific data structures into DKUPiece standard data structures and enforces dependencies within the data. For example, in H264 deblocking, a macro block depends on having its neighbor above and neighbor to the left being already finished. The Dividable Piece Maker encodes these dependencies. It is handed the original data structure, and returns maximally-sized independent DKUPiece structures that can be freely sub-divided. Each piece it hands back must be processed in sequence.

The Dividable Piece Maker written for H264 deblocking is given a frame of data and slices it into diagonals of macro-blocks, handing back each diagonal as a single piece. Each diagonal can then be freely sub-divided, with its macro-blocks distributed among the sub-pieces.

3.5 Extension for Distributed Memory: The Bundling Quad

The bundling quad is an extension to the basic DKU interface that enables specialization to distributed memory machines. As the name implies, it consists of four functions: bundleInputs, unbundleInputs, bundleResults, and unbundleResults. Their effect is to make remote execution look as though it happened locally.

Each bundling function can be thought of as a "port" for sending between local memory and remote memory. They are only invoked by schedulers on distributed memory machines, so impose no overhead on shared memory.

BundleInputs is run locally, gathers all data that will be touched by the Kernel, and returns a pointer to the bundle. The scheduler sends this bundle, via its specializer-inserted, hardware-specific, communication infrastructure, to a remote scheduler or to a remote communication stub.

In the remote memory the remote scheduler calls unbundleInputs, then the Kernel, then bundleResults. UnbundleInputs unpacks the data and returns a DKUPiece. The DKUPiece is handed to the Kernel, which produces results that are reachable from the DKUPiece as per the standard for Kernels. Then bundleResults is handed the DKUPiece. It packs all result data into an array (or object) and returns a pointer. The remote scheduler then sends the result bundle back.

The specializer-inserted communication infrastructure that runs in local memory then receives the bundled result data. The local scheduler hands it to unbundleResults which modifies the local copy of the DKUPiece, making local memory look exactly the same as if the Kernel had run locally.

This set of functions cleanly hide remote execution from both the local application and the Kernel. The application is free to use global variables, shared data structures, and so forth. In practice it has been surprisingly easy to implement the bundling quad.

4 The Applications

We have chosen a set of three applications to demonstrate using the DKU interface. Two of these are implemented in Java, and are more "benchmark" code, while the third is taken from a real application written in C. The benchmarks are dense matrix multiply and a Hamiltonian Path solver (an NP-Complete problem). The real application is H264 deblocking, which was previously optimized by hand for high performance on serial processors.

4.1 Deblocking Filter for H264

This is the most interesting, from a real-world perspective, as it was taken from an active project [12] and optimized for high speed on serial hardware. As such, it breaks many abstractions natural to the application. For example, the "macro block" is a natural unit of data. In an object oriented program, one would include all the data for a macro block in a single object. However, in the C code, the data is flattened onto linear arrays then accessed by calculating addresses within the arrays (the arrays are static variables).

For many parallel programming paradigms, the global, shared, flattened, arrays and address arithmetic would be problematic, due to side-effects, incompatibility of single shared arrays with the language's parallel model, and so forth. For DKU, it is handled via the Dividable Piece Maker and the bundling quad, without modifying the core of the original application code, and without undue effort. The details of how this was done are too involved for the space available; the best way to see them is via the code on the website [4].

The dividable piece maker we implemented divides a frame, consisting of macro blocks, into diagonals, making each diagonal a separate root DKUPiece. Each macro block requires its neighbor above and neighbor to the left to be completed before it is calculated, so all the macro blocks on a 45 degree diagonal are independent from each other and all free to calculate once the preceeding diagonal is complete. The divider simply assigns the macro blocks from a diagonal to sub-pieces.

4.2 Dense Matrix Multiply

Nothing special was done in the sample implementation of matrix multiply we supply. The divider slices the two input matrices into "strips". A single DKUPiece contains a pair of matrices to be multiplied. When such a pair is divided, its left matrix is sliced horizontally into clusters of rows, while the right matrix is sliced vertically into clusters of columns. Each combination of clusters is made into a separate sub-piece. This has the side-effect of discretizing the number of sub-pieces the divider is able to make. Note, however, that the division doesn't move any data in the matrices around. Rather, it calculates start and end rows for the left matrix, and start and end columns for the right.

4.3 Hamiltonian Path

In the Hamiltonian Path problem, one finds a path that visits every node of a graph exactly once, or else determines that no such path exists. The standard approach is to use back-tracking, which, when one examines the control flow, performs a search that has the shape of a tree. We turned the back-tracking algorithm into an iterative algorithm that represents the current point in the search as a partial-path. There is a one-to-one correspondence between each possible partial-path and a node in the search tree.

The property we exploit is that in a tree, given two nodes neither of which is an ancestor of the other, those two nodes have no descendants in common. So, if search proceeds independently from those two nodes, the two searches will remain independent and will duplicate no work.

The divider finds a set of such nodes by traversing the search tree breadth-first. It accumulates the nodes it visits, removing a node when it explores its children. Meanwhile the Kernel is written to search depth-first to increase the chances of early termination. The undivider, in essence, performs a logical OR of all the answers. If they're all null, the total answer is null. Otherwise, it takes the first path returned to it as the final answer.

The search tree is typically exceptionally unbalanced. To handle this, we introduced a "re-divide" interface extension. The scheduler signals a running Kernel to stop for re-division, then hands the partially-completed piece to the ReDivider. For Hamiltonian Path, we exploit the re-entrant nature of the divider, and implement the redivider as a few "fix ups" that handle the differences between depth-first and breadth-first search.

In the case that one of the sub-trees finds a path quickly, we have also added an extension to the DKU interface that allows the undivider to tell the scheduler to early-terminate the un-finished DKU-pieces.

5 Specialization Infrastructure

In general, specialization can be performed in many different ways: by hand, by a build-script, or by some more general infrastructure. To support research, we have chosen to try to minimize the effort required by a researcher who wants to focus on one step of specialization such as Code Structure Transforms or specific scheduling algorithms. Hence our sample BLIS implementation has sample code of infrastructure. It has a BLIS directory with serial implementations that is supplied to the application developer, a sample Specialization Server, and a sample BLIS Client.

5.1 Application Development

Figure 2 shows that during development, the application source includes an inserted BLIS directory that is supplied as part of our framework. The BLIS directory has a DKU sub-directory that has all standard data-structure definitions and a sequential implementation of the DKU-scheduler. By being serial, the sequential-scheduler allows development to proceed in a serial environment, such as the application developer's favorite development environment.

When the D, K, U, and DividablePieceMaker are being debugged, an alternate implementation of the scheduler is swapped in, which calls the D, K, U, and DividablePieceMaker in a serial fashion. This pattern of swapping in different scheduler implementations leaves room for more sophisticated debugging machinery. For example, a serial scheduler could stimulate timing bugs by performing repeatable inter-leaving and simulating distributed memory.

Later, during specialization, the application will be linked, in turn, to one or more hardware-specific scheduler implementations. In our framework implementation, the hardware specific scheduler's source is packaged inside a specializer module.

The use of a BLIS directory during app-development provides additional benefits. As mentioned, our BLIS implementation allows given application code to remain compatible with both future and past schedulers as the interface is extended, and modified. The use of a BLIS directory enables this. When an extension is accepted into the BLIS interface, then a new BLIS directory containing serial implementations of all the interfaces, including the new ones, is made available for developing applications, and for developing schedulers. Meanwhile, the BLIS directories inside each application and inside each specializer already in the central Specialization Server are merged with the new directory, such that "do nothing" versions of missing interface implementations are added. Hence, all interfaces "work".

The caveat is that schedulers written to new versions of the interface will not necessarily do anything interesting with applications written to old versions and vice versa. This scheme only ensures that old apps still work the same with old schedulers and new works with new, and no compilation errors happen when new is mixed with old.

5.2 Our Specialization Server Implementation

The middle of Figure 2 shows that the server in our sample implementation sends the source to each specializer module. A specializer is self-contained and is invoked by a script placed in a designated directory. The server simply calls all scripts in that directory. Plugging in a new specializer module consists of placing the module's entry-script into that directory.

The entry script is handed a copy of the application code-base. From there it invokes other scripts in the specialization module. The first script typically deletes the DKU directory that came with the app and replaces it with the DKU directory packaged inside the module. Next, if the specializer has any code-structure transforms (CST), a script invokes those. When transform completes, a script invokes the compilation process. Finally, a script packages the resulting executables and sends the package, along with information about the specializer, the application, and the target hardware, to the Server's distributor.

The Server's distributor accepts executable bundles from specializer modules, and also listens for requests from clients running on end-platform hardware. To run an application on a given machine, a client on that machine is used to get the application. It sends information about the hardware and the desired application. The Distributor sends back the appropriate executable. The client unpacks the executable and makes it available to the OS to be run. The client may simply be a human using FTP, a script or a test harness.

5.3 Client on end-hardware

We chose to include a client on end-hardware to improve support for the search for the boundary between application and hardware. The search needs a broad base of applications; to encourage application development we provide "turn-key" infrastructure that automates the entire toolchain, all the way to invoking an application on target hardware via a BLIS Client. The right of the top of Figure 2 shows that a client sends detailed information about its platform to the Specialization Server, potentially including cache sizes, number of cores, operating system settings, and so forth. The server hands the client the executable best suited to the hardware. Our sample code implements the client as a human using FTP and a web-page interface to the data base.

A more interesting client would be one that runs on a mobile device, as envisioned for the OMP (Open Media Platform) project[5] which supported this work on the BLIS framework. Here, the client requests the media-components required to play a particular content stream, as needed, and may even request according to resource usage and quality of experience offered.

6 Specializers and the Schedulers Inserted by Them

The heart of the BLIS framework is the specializers that produce the executable images.

There is only one formal requirement for specializers and schedulers: Specialization must be performed, and the code output from Specialization must produce the same result as with the sequential scheduler. Beyond this, they are free. For instance, they don't even have to use the library interface call to the scheduler; the specializer could perform a Code Structure Transform that turns the call to the Scheduler into an OS call to start the Kernels, and so on.

For our automated infrastructure, specializer modules often have two parts: a BLIS directory that contains pre-written scheduling + communication code, and a script that first replaces the serial BLIS directory, that came with the application source, with the hardware-specific one, and then compiles the result. The pre-written scheduler in the new BLIS directory makes calls to the D, K, U, etc implemented in the application, while the application calls the scheduler. Such a specializer module does not modify the calls, so there are no compilation errors. Changing the contents of the BLIS directory has only changed the implementations of both directions of library call.

In the following sub-sections, we describe some details of our sample specializer modules and the schedulers packaged inside them, for each of our test hardware platforms.

6.1 Java specializer and scheduler for Shared Memory Multicore

Our sample specializer module for Java on shared memory multi-core machines has no Code Structure Transforms. It only removes the serial DKU directory and replaces it with one containing our pre-written scheduler for multi-core machines.

We didn't concentrate on performance, as our intent is to demonstrate how to make a scheduler, not to try to make an interesting one. The scheduler creates one worker-thread for each core, and communicates with it via a pair of one-way queues.

A separate `DKUScheduler` object is created in the application's main thread. Then its `scheduleAndPerformWorkOn` method is called and handed an `Object`. The method uses the `DividablePieceMaker` to turn the `Object` into an array of `DKUPiece` objects, then loops through the objects invoking the divider method of each one. It tells the divider to make the same number of pieces as the number of worker threads. Then it loops through the sub-pieces, sending each to a worker thread, round-robin. The worker threads take DKUPieces out of the queue, call the Kernel method of the DKUPiece, which produces results by side-effect, then put the piece into the return queue back to the scheduler. The scheduler calls the undivider method on the parent piece, passing it each sub-piece until the parent says all sub-pieces are accounted for. The scheduler then loops to the next DKUPiece in the array, and returns when all are done. There is significant room for scheduler improvement.

6.2 Java specializer for Heterogeneous Networks of Machines

The scheduler in this specializer module has two levels that run in separate JVMs: one scheduler is part of the application executable, and a second scheduler is in a stand-alone "worker" that runs on each machine in the heterogeneous collection. A worker accepts pieces from all applications running on the collection.

Two levels of division are performed, adaptively. The first level of division is performed by the application-scheduler. It reads a config file of the machines available, and makes enough pieces that it can hand each machine a number proportional to its processing power. The second division-level is performed inside each worker.

The worker running on a given machine is implemented specifically for that machine, so it potentially performs division and scheduling in its own way. Our implementation is for a collection of shared memory multicore machines, so the workers are all implemented the same, but they differ in the number of threads they schedule onto. The workers divide the pieces they receive, to end up with close to the same number of sub-pieces as there are hardware threads in the machine.

This demonstrates a useful property of the DKU interface, that division is re-entrant, so it can be performed repeatedly on sub-pieces and sub-sub-pieces, etc. The choice of further sub-division is left to the receiver of a piece.

6.3 C specializer for Cell BE Processor

The Cell BE hardware makes this specializer more interesting. The scheduler code is split into two parts, one part that runs on the two PPUs, and a second part that runs on the SPEs. As with the other schedulers, this is sample code to show how to make a scheduler for the Cell that uses the DKU interface, without special effort to gain performance.

The Cell has two different instruction sets inside it, one for the PPU, a second for the SPE, so the specializer must create two separate code bases and call two separate compilers (with help from the SDK). The functions from the application that execute on the SPEs are `unbundleInputs()`, `Kernel`, and `bundleResults()`, so these must be copied out of the application code-base and into the DKU directory where they are inserted into SPE code templates. We performed the copy and insertion by hand rather than writing scripts. It takes about 10 mins to specialize the generic source to run on the Cell.

In the C version of DKU, before calling the schedule function the first time, `schedulerInit()` is called. For the Cell, this init uploads the SPE code templates, with their inserted application functions, and starts them running. The SPEs then loop looking for work.

The PPU scheduler has the same architecture as the Java and C shared memory versions, with the difference lying in the worker threads. Rather than performing the work itself, the worker thread instead finds a free SPE and transfers the work to it.

This requires the worker thread to first call `bundleInputs`, then tell the SPE the address of the bundle. The SPE uploads the bundle and runs `unbundleInputs` in its local memory, hands the created DKUPiece to the Kernel, then calls `bundleResults` on the completed DKUPiece. The SPE then copies the result bundle to the PPU memory and notifies the worker threads. The worker that gets the notification pairs the returned result bundle with the original DKUPiece in PPU memory and runs `unbundleResults`, which moves data out of the bundle. After the data-moves, PPU memory looks just like the work had been performed locally.

The specializer has to handle machine details, such as the difference in pointer sizes between the PPUs, at 64bits, and the SPEs at 32 bits. It gets help from the application and the DKU standard.

Recall that the Kernel is defined to only touch data that is reachable from a DKUPiece, while `bundleInputs` is defined to gather all data a Kernel will touch from a given DKUPiece. This hides global variables and shared variables from the scheduler code.

Native data size differences are hidden by `#define`s. The DKU.h file `#define`s standard size data types, for example `int32` and `uint8`. The application uses these for all data touched by the Kernel, instead of `int` and `char`. This allows the specializer to include a hardware-specific `DKU.h` file in its DKU directory. The `#defines` are written as part of the specializer, possibly using macros, and make the compilation performed inside the specializer use the right assembly instructions to be consistent with the defined size.

Pointer size differences and endian differences are handled by `#define`s as well, two for pointer sizes in local and remote memories, and two for the endianness in each. The bundling quad is written, by the application programmer, to use these `#define`s in the `DKU.h` file to construct a bundle with the right sizes and byte orderings for remote memory. Pointer arithmetic and normal arithmetic then function correctly inside a remote Kernel binary, which was compiled to a different ISA, without the application ever knowing what that ISA might be. Further, `if` statements in the bundling quad, that check the `#define` values to decide which rearrangements of data to perform, are optimized away, leaving only the operations required for the actual combination of ISAs to be compiled and run.

Lastly, the application should not use align statements in data structures that are copied by `bundleInputs`. If such statements are required, then `bundleInputs` is written to perform a primitive-by-primitive copy into the bundle. Meanwhile `unbundleInputs` is written to create a new data structure and copy the individual primitives into it. With this approach, the two compilers should correctly handle differences in alignment on the local and remote machines.

7 Experimental Results

We stress that the schedulers tested here are sample code, written to show how it is done, with no interesting features for performance. The results serve to show that the bidirectional interface pattern works; performance is placed by the BLIS interface into the hands of the specializer implementors and application implementors.

When measuring overhead, it can be subtle determining which overhead should be counted against the interface, and which is intrinsic to parallelism. We adopt this question as the defining test to determine what is overhead of BLIS vs overhead intrinsic to parallelism:

> "would this overhead occur in the application, if it were written by hand using the scheduling primitives available on the hardware?"

In most cases, the answer to the question is yes, the overhead would also occur in a hand-coded version, which makes the overhead intrinsic to parallelism. The one set of overheads that are clearly unique to using BLIS are the `call` instructions of the interface calls.

However, some overheads may be made worse by using BLIS, but by how much is implementation dependent. These are: loss of parallelism opportunities due to BLIS's choice of interface pattern; additional think time in a scheduler due to lack of needed information or lack of a needed "knob" to manipulate application-specific quantities; time lost in the barrier implied by the undivider; and added overhead caused by the form of sending signals between the scheduler and a running Kernel that has been implemented to communicate.

We have no general solution to measure the worsening of these overheads resulting from the BLIS form. However, for some actions we measure times in the schedulers, which admits intuition about how efficient the BLIS code is.

In the experiments, we used a 2 core laptop, denoted “1x2”, a 2 socket by 4 core each, denoted “2x4”, a 4 socket by 4 core each, denoted “4x4”, a heterogeneous network of them connected by 100Mbit LAN, denoted “Het”, and the Cell BE in a PS3. Of note, the heterogeneous network demonstrates the use of the re-entrant feature of the Divider, as mentioned in Sec 6.2.

Matrix Multiply in Java: Figure 4 shows efficiency on Matrix Multiply in Java, when run on three different multi-core machines, plus the heterogeneous collection of them. The break even size is around 100x100 double precision on the multicore machines. However, on the heterogeneous collection, the amount of data sent over the 100 MBit LAN limits performance. Break even is around 200x200, where serial Kernel time is that on the machine the app is lauched from. On matrix multiply, the comp-to-comm ratio grows linearly with matrix size, causing percent ideal speedup to increase linearly with matrix size.

The Slow-down for small matrix sizes can be avoided by the addition of an interface for execution-time prediction. When added, the schedulers will be able to avoid the parallelism overhead by instead calling the serial Kernel when a slow-down is predicted.

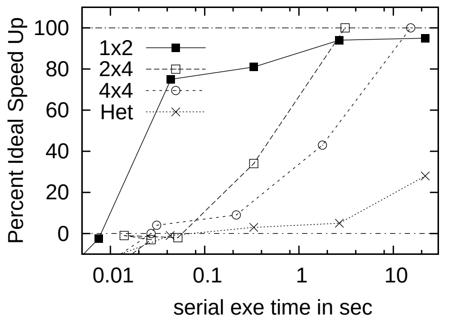

Figure 4. Matrix Multiply in Java: The percentage of the ideal speedup achieved on each of four machines vs the serialKernel execution time. Break-even exe time on a machine is found at the 0% point (serial exe time = parallel exe time). The size of matrix at succesive points along one curve: 9x9, 81x81, 162x162, 324x324, 648x648, 1296x1296

$$\text{PercentISU} = \frac{\frac{\text{Tser}}{\text{Tpar}} - 1}{\text{p-1}} \cdot 100$$

Hamiltonian Path in Java: Hamiltonian Path is used to demonstrate several things: the interface does evolve as needed; the interface admits high-efficiency; and complex problems can be expressed within the DKU interface. Hamiltonian Path is a complex problem, its running time is inherently not predictable, as far as is currently known, so the running-time of an input-graph in a serial thread is the base quantity, rather than size of the input-graph.

Table 1 shows the evolution of the interface: two input-graphs are run on a 4x4 core machine four times. Each time is with a different implementation of the scheduler: serial, one-time division, redivision, and redivision plus early termination. The dramatic changes in running time show the value of being able to incrementally add features to the interface. The fact that the same application source is run on all four versions of the scheduler demonstrates the point that “old” schedulers can be successfully used with “new” applications. This backwards-and-forwards compatible feature of the framework has practical value for researchers.

The odd timings are due to the nature of the search Kernel: it explores the search tree depth-first until it finds a solution, then stops. So the existence of a solution allows early termination, while on a graph with no solution, the full search tree must be explored. Hence, dividing the search tree of a graph can separate a solution into one piece, while the other pieces have no solution. The solution-containing piece finishes early while the others fully explore their sub-trees. This is why exe time increases from sequential to one-time divide: a previously hidden sub-tree with no solution is placed into its own piece, and all the other pieces wait for that one to complete.

Ham. Path	Sched:	Serial	One Div.	Rediv.	Rediv. + Early Term.
	Solution	140s	272s	43s	0.012s
	No Soln.	52s	21s	3.4s	3.3s

Table 1. Four versions of the scheduler, each on two input-graphs run on a 4x4 core machine. The top row shows running times for the graph that has a solution, while the bottom row is for a graph that has no Hamiltonian Path in it.

This behavior drove the addition of the redivide and early-terminate interface extensions. In the “redivide” interface, the Kernel stops when signaled by the scheduler, and the work remaining in the piece is divided then handed out to idle workers. The table shows that, for this input-graph, redivision gives near linear speedup over one-time divide. The deviation from fully linear is due to working on the orignal pieces, that finished early. Notice that it would finish even faster if the long-running pieces could be stopped as soon as a solution is found. This is what the Early-terminate interface provides. Its use gives the surprising speedup seen in the top of the last column. However, when no solution exists in the input graph, the speedup remains linear as seen just below, in the last column of the second row.

Figure 5 shows efficiency: the same input-graphs are run on all the test machines, in a single thread first then on multiple threads. The input-graphs all have no solution, so the amount of work stays constant. The plot shows that the speedup approaches perfect speedup on all machines once the serial time is large enough. This shows that reasonable performance can be achieved even with very simple schedulers.

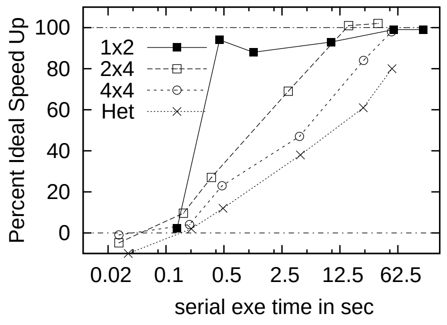

Figure 5. Hamiltonian Path in Java: The percentage of the ideal speedup achieved on each of the multicore machines vs the serialKernel execution time. All input graphs are no solution types. The Redivide + Early Terminate scheduler was used. Exe time is the same for serial and parallel execution at the 0% SU point.

Table 2 shows Cell BE results for deblocking of a cell-phone size screen. All of the parallel configurations using the SPEs caused slowdown due to communication delays. As seen by the Kernel time vs Comm time, deblocking performs very little work on each byte transferred, so communication latency dominated. However, in a system with other sources of parallelism, the latency might be overlapped.

This illustrates the importance of the serial kernel. As seen here, not all sources of parallelism exposed in an application are exploitable on all hardware. The serial kernel's speedup of 2.3x over the 6 SPE configuration shows that the flexibility to decide at run-time whether to exploit the parallelism can significantly affect performance.

H264 D.F. on Cell BE	Serial PPE	1 SPE	3 SPE	6 SPE
Total time	2.0ms	16ms	8.9ms	4.7ms
Kernel time	n/a	2.2ms	0.7ms	0.3ms
Comm time	n/a	13sms	7.6ms	4.2ms

Table 2. Cell results on H264 deblocking. Total time is for one frame of 320 x 200 pixels. Kernel time is the portion of that due the Kernel. Comm time is the portion due to the bundling quad plus queues plus DMA. The "missing" time was spent in the scheduling code on the PPU. "Serial" means the serialKernel was run on the PPU, while "X SPE" means the parallel version was run using that many SPEs.

These Cell results show how the framework can be useful. The hand-coded versions of H264 achieve parallel speedup, by performing multiple steps on the same data once it is in an SPE. To close the gap between hand-coding vs machine-independent source, greater flexibility is needed in automatically arranging the sources of parallelism exposed in the portable source. This framework helps in exploring how to express the needed flexibility, and in exploring how to write a specializer and a scheduler for the Cell that take advantage of that expressed flexibility. Productivity gains in doing that kind of research are the goal of the proposed framework.

8 Conclusion

We have shown a framework to support research on portable high performance parallelism. It uses the pattern of bi-directional library calls to successfully separate application code from hardware specific code, while enabling mix-and-match of applications to hardware. The hardware scheduler is called as a library function, and interacts with "knobs" that the application makes available as reverse-library functions.

We showed a bidirectional library interface for data parallelism, three applications instrumented with it, and results of automated specialization of the applications to shared memory multicore machines. The applications were written and debugged in a sequential IDE in Java and C; then the sources were specialized, automatically or by hand, to the Cell BE, to various multi-core, and to a heterogeneous collection of multi-core. The performance results show that this approach works, and anecdotal evidence of the time we spent on implementation of the specializers indicates that the proposed framework may be efficient and helpful for research on portable parallelism.

References

[1] Berkeley Pattern Language. http://parlab.eecs.berkeley.edu/wiki/patterns.

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

[3] CnC homepage. http://software.intel.com/en-us/articles/intel-concurrent-collections-for-cc/.

[4] Deblocking Filter code. http://dku.svn.sourceforge.net/viewvc/dku/branches/DKU_C__Deblocking__orig/.

[5] Open Media Platform homepage. http://www.openmediaplatform.eu/.

[6] Sample BLIS Code. http://dku.sourceforge.net/SampleCode.htm.

[7] Scala homepage. http://www.scala-lang.org/.

[8] Suif parallelizing compiler homepage. http://suif.stanford.edu.

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

[10] Unified Parallel C homepage. http://upc.lbl.gov/.

[11] K. Asanovic, R. Bodik, B.C. Catanzaro, J.J. Gebis, P. Husbands, K. Keutzer, D.A. Patterson, W.L. Plishker, J. Shalf, S.W. Williams, et al. The landscape of parallel computing research: A view from berkeley. *Electrical Engineering and Computer Sciences, University of California at Berkeley, Technical Report No. UCB/EECS-2006-183, December*, 18(2006-183):19, 2006.

[12] Arnaldo Azevedo, Cor Meenderinck, Ben Juurlink, Andrei Terechko, Jan Hoogerbrugge, Mauricio Alvarez, and Alex Ramirez. Parallel h.264 decoding on an embedded multicore processor. In *HiPEAC '09: Proceedings of the 4th International Conference on High Performance Embedded Architectures and Compilers*, pages 404–418, 2009.

[13] G.E. Blelloch, J.C. Hardwick, S. Chatterjee, J. Sipelstein, and M. Zagha. Implementation of a portable nested data-parallel language. In *Proceedings of the fourth ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 102–111. ACM New York, NY, USA, 1993.

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

[15] 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, 2006.

[16] D. Gelernter. Generative communication in Linda. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 7(1):80–112, 1985.

[17] 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.

[18] C. Lin and L. Snyder. ZPL: An array sublanguage. *Lecture Notes in Computer Science*, 768:96–114, 1994.

[19] T. Mattson, B. Sanders, and B. Massingill. *Patterns for parallel programming*. Addison-Wesley Professional, 2004.

[20] J. McGraw, SK Skedzielewski, SJ Allan, RR Oldehoeft, J. Glauert, C. Kirkham, B. Noyce, and R. Thomas. SISAL: Streams and iteration in a single assignment language: Reference manual version 1.2. *Manual M-146, Rev*, 1.

[21] 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.

[22] Jocelyn Serot and Joel Falcou. Functional meta-programming for parallel skeletons. In *ICCS '08: Proceedings of the 8th international conference on Computational Science, Part I*, pages 154–163, 2008.

[23] D.B. Skillicorn and D. Talia. Models and languages for parallel computation. *ACM Computing Surveys (CSUR)*, 30(2):123–169, 1998.

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

[25] Wikipedia. HPF wikipedia page. http://en.wikipedia.org/wiki/High$_P$erformance$_F$ortran.

[26] Wikipedia. MPI wikipedia page. http://en.wikipedia.org/wiki/Message$_P$assing$_I$nterface.