DATA PARALLELISM WITH HIERARCHICALLY TILED OBJECTS

BY

JAMES C. BRODMAN

DISSERTATION

Submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy in Computer Science in the Graduate College of the University of Illinois at Urbana-Champaign, 2011

Urbana, Illinois

Doctoral Committee:

Professor David Padua, Chair, Co-Director of Research
Research Assistant Professor María J. Garzarán, Co-Director of Research
Professor William Gropp
Professor Michael Heath
Professor Ron Cytron, Washington University in St. Louis
Abstract

Exploiting parallelism in modern machines increases the difficulty of developing applications. Thus, new abstractions are needed that facilitate parallel programming and at the same time allow the programmer to control performance. Tiling is a very important primitive for controlling both parallelism and locality, but many traditional approaches to tiling are only applicable to computations on dense arrays. This thesis makes several contributions, all in the general area of data parallel operators for the programming of multiprocessors and their current most popular incarnation, multicores. It accomplishes this through the development of Ravenna, a library of data parallel operators for shared-memory systems. Ravenna extends previous work on a data type for dense arrays called the Hierarchically Tiled Array, or HTA.

Ravenna supports arbitrary data types, enabling programmers to write data parallel computations based on other data types such as sets or graphs. Ravenna provides programmers with several mechanisms for tiling data types. In particular for data structures other than dense arrays, it provides a generalized approach called functional tiling. Functional tiling provides programmers with a separation of concerns between implementing a computation and how to tile it. Functional tiling in this way also acts as a tuning mechanism that allows programmers to tune the performance of their codes by plugging in different tiling strategies.

This thesis evaluates the programming model of expressing programs as a sequence of higher level data parallel operators through examining several applications from different domains written in Ravenna. These applications include simple microbenchmarks used to
compare against another shared-memory programming library, a solver for banded linear systems called SPIKE, n-body simulation, clustering, and discrete optimization. The evaluation shows that these programs can be elegantly expressed by the programming model, and that the model’s applicability is not limited to computations based on dense arrays. Particularly, it shows that the resulting programs resemble conventional, sequential programs, simplifying programmer effort and that the available abstractions provided by Ravenna allow programmers to tune in order to obtain good parallel performance.
Acknowledgements

This work would not have been possible without the support of several people. First, I would like to thank my advisors, David Padua and María Garzarán, for their encouragement and advice. I would also like to thank committee members William Gropp and Michael Heath for their feedback and suggestions. Special thanks go to committee member Ron Cytron, who gave me my start in research as an undergraduate.

My friends and fellow group members also deserve recognition for making my time at Illinois enjoyable. I am glad that our research group consists of people who are not just colleagues but also friends.

Finally, I would like to thank my family for their continued support over the many years of my education. They have always been there with words of encouragement during times of stress and with words of praise in times of success.
Table of Contents

List of Figures .............................................................. vii
List of Tables ............................................................... x

Chapter 1 Introduction ..................................................... 1
1.1 Overview ............................................................... 1
1.2 Contributions ......................................................... 5
1.3 Thesis Organization ................................................... 6

Chapter 2 Ravenna, A Library of Data Parallel Operators ........... 8
2.1 The Hierarchically Tiled Array ....................................... 8
2.2 Programming Model .................................................. 11
2.3 Tiling ................................................................. 15
  2.3.1 Functional Tiling ............................................... 17
2.4 Implementation ....................................................... 18

Chapter 3 Comparison to TBB ........................................... 22
3.1 The Intel TBB library ................................................ 22
  3.1.1 TBB operations ............................................... 23
3.2 Parallel Merge ....................................................... 24
3.3 The Game of Life ..................................................... 27
3.4 Average ............................................................. 32
3.5 Substring Finder ..................................................... 35
3.6 Evaluation .......................................................... 36
3.7 Conclusion .......................................................... 39

Chapter 4 Tiled SPIKE .................................................... 43
4.1 SPIKE Variants ....................................................... 46
4.2 Tiled Spike .......................................................... 49
  4.2.1 TU .......................................................... 50
  4.2.2 TA .......................................................... 52
4.3 Evaluation .......................................................... 53
  4.3.1 TU .......................................................... 55
  4.3.2 TA .......................................................... 56
4.4 Conclusion ................................................................. 57

Chapter 5 Computations on Non-Array Datatypes ................. 62
5.1 Barnes-Hut ................................................................. 63
5.2 Clustering ................................................................. 70
5.3 Breadth-First Search ..................................................... 74
5.4 Best-First Search .......................................................... 78
   5.4.1 Different Tiling Strategies ........................................ 81
   5.4.2 Centralized Data Structures ....................................... 83
   5.4.3 Overheads ............................................................... 88
5.5 Programmability ......................................................... 90

Chapter 6 Related Work .................................................. 92
6.1 Summary ................................................................. 96

Chapter 7 Conclusions .................................................... 98
7.1 Conclusions ............................................................... 98
7.2 Future Work ............................................................. 100
7.3 Limitations of the Model .............................................. 100

Appendix A Additional SPIKE Experiments ......................... 103

Appendix B Additional Barnes-Hut Experiments .................... 117

Bibliography ................................................................. 121
## List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>Hierarchical Tiling</td>
<td>9</td>
</tr>
<tr>
<td>2.2</td>
<td>HTA Indexing</td>
<td>10</td>
</tr>
<tr>
<td>2.3</td>
<td>The tile Operator</td>
<td>19</td>
</tr>
<tr>
<td>2.4</td>
<td>Example with the tile operator</td>
<td>21</td>
</tr>
<tr>
<td>3.1</td>
<td>TBB Parallel Merge</td>
<td>25</td>
</tr>
<tr>
<td>3.2</td>
<td>Ravenna Parallel Merge</td>
<td>26</td>
</tr>
<tr>
<td>3.3</td>
<td>TBB Game of Life</td>
<td>28</td>
</tr>
<tr>
<td>3.4</td>
<td>Ravenna Game of Life</td>
<td>29</td>
</tr>
<tr>
<td>3.5</td>
<td>TBB Calculate State</td>
<td>29</td>
</tr>
<tr>
<td>3.6</td>
<td>TBB GetAdjacentCellState</td>
<td>30</td>
</tr>
<tr>
<td>3.7</td>
<td>Ravenna Calculate State</td>
<td>31</td>
</tr>
<tr>
<td>3.8</td>
<td>TBB Average</td>
<td>33</td>
</tr>
<tr>
<td>3.9</td>
<td>Ravenna Average</td>
<td>34</td>
</tr>
<tr>
<td>3.10</td>
<td>TBB version</td>
<td>35</td>
</tr>
<tr>
<td>3.11</td>
<td>Ravenna version</td>
<td>36</td>
</tr>
<tr>
<td>3.12</td>
<td>Parallel Merge Performance, 8 core Xeon</td>
<td>40</td>
</tr>
<tr>
<td>3.13</td>
<td>Parallel Merge Performance, 16 core Itanium 2</td>
<td>40</td>
</tr>
<tr>
<td>3.14</td>
<td>Game of Life Performance, 8 core Xeon</td>
<td>41</td>
</tr>
<tr>
<td>3.15</td>
<td>Game of Life Performance, 16 core Itanium 2</td>
<td>41</td>
</tr>
<tr>
<td>4.1</td>
<td>Banded System, p=4</td>
<td>44</td>
</tr>
<tr>
<td>4.2</td>
<td>Spike Matrix, p=4</td>
<td>44</td>
</tr>
<tr>
<td>4.3</td>
<td>Spike TU Partitioning</td>
<td>47</td>
</tr>
<tr>
<td>4.4</td>
<td>Spike TA Partitioning</td>
<td>47</td>
</tr>
<tr>
<td>4.5</td>
<td>Data sources for a reduced with 4 blocks</td>
<td>48</td>
</tr>
<tr>
<td>4.6</td>
<td>Tiled SPIKE TU</td>
<td>52</td>
</tr>
<tr>
<td>4.7</td>
<td>Tiled SPIKE TA</td>
<td>54</td>
</tr>
<tr>
<td>4.8</td>
<td>TU Speedups</td>
<td>58</td>
</tr>
<tr>
<td>4.9</td>
<td>Cluster TU Speedups</td>
<td>59</td>
</tr>
<tr>
<td>4.10</td>
<td>TA Speedups</td>
<td>59</td>
</tr>
<tr>
<td>4.11</td>
<td>Cluster TA Speedups</td>
<td>61</td>
</tr>
<tr>
<td>5.1</td>
<td>Spatial Decomposition (a) and Octree (b)</td>
<td>64</td>
</tr>
<tr>
<td>5.2</td>
<td>Barnes-Hut Code</td>
<td>64</td>
</tr>
</tbody>
</table>
5.3 Sequential Barnes-Hut ................................................. 64
5.4 Performance of Tiling Strategies for Barnes-Hut (Xeon E7450) ........ 67
5.5 Performance of Tiling Strategies for Barnes-Hut (Xeon E5405) ........ 67
5.6 Cache Performance (NP=8) ........................................ 68
5.7 Standard Deviation of Avg # of Misses Per Tile (NP=8) ................. 68
5.8 Parallel k-means .................................................. 72
5.9 SELECT.CENTER() ................................................ 72
5.10 Performance of K-means (Xeon E7450) ................................ 73
5.11 Performance of K-Means (Xeon E5405) ................................ 73
5.12 Cache Performance of K-means (Xeon E5405) .......................... 75
5.13 Data Parallel Breadth-First Search ................................ 75
5.14 Data Parallel Breadth-First Search with Tiling ......................... 77
5.15 Tiled Breadth-First Search ...................................... 77
5.16 15 Puzzle Search Graph ......................................... 79
5.17 Data Parallel Best-First Search with Tiling ........................... 84
5.18 Performance of Puzzle Tiling Strategies (Xeon E7450) .................... 84
5.19 Performance of Puzzle Tiling Strategies (Xeon E5405) .................... 85
5.20 Performance of the Initial Configurations (Xeon E7450) .................. 85
5.21 Puzzle Cache Behavior .......................................... 86
5.22 Library Overhead .................................................. 89

A.1 TU, N = 131072, K = 16 ........................................... 104
A.2 TU, N = 131072, K = 32 .......................................... 104
A.3 TU, N = 131072, K = 64 .......................................... 105
A.4 TU, N = 131072, K = 128 ...................................... 105
A.5 TU, N = 131072, K = 256 ...................................... 106
A.6 TU, N = 262144, K = 16 ....................................... 106
A.7 TU, N = 262144, K = 32 ....................................... 107
A.8 TU, N = 262144, K = 64 ....................................... 107
A.9 TU, N = 262144, K = 128 .................................... 108
A.10 TU, N = 262144, K = 256 .................................... 108
A.11 TU, N = 524288, K = 16 .................................... 109
A.12 TU, N = 524288, K = 32 .................................... 109
A.13 TU, N = 524288, K = 64 .................................... 110
A.14 TU, N = 524288, K = 128 ................................... 110
A.15 TU, N = 524288, K = 256 ................................... 111
A.16 TU, N = 1048576, K = 16 .................................... 111
A.17 TU, N = 1048576, K = 32 .................................... 112
A.18 TU, N = 1048576, K = 64 .................................... 112
A.19 TU, N = 1048576, K = 128 .................................. 113
A.20 TU, N = 1048576, K = 256 .................................. 113
A.21 TU, N = 2097152, K = 16 .................................. 114
A.22 TU, N = 2097152, K = 32 .................................. 114
List of Tables

2.1 Ravenna Primitives ................................................................. 12
2.2 Ravenna Indexing Operators .................................................. 13
2.3 Ravenna Tiling Primitives ....................................................... 14

3.1 Number of Lines of Code .......................................................... 37
3.2 Performance on Intel Xeon using 1 to 8 cores and 1 tile per core .......... 38
3.3 Performance on Intel Itanium 2 using 1 to 16 cores and 1 tile per core .... 39

6.1 Comparison of Features ............................................................ 97
Chapter 1

Introduction

1.1 Overview

Parallelism is pervasive in modern desktop machines. From microprocessor vector extensions to multicores to GPUs, parallel computing is readily available. The high performance provided by parallel computing enables scientists to perform research to predict weather, model traffic systems, forecast prices for financial markets, model molecular reactions, or test vehicle safety. However, despite all the research put into parallel programming, it remains a significantly more difficult task than sequential programming. Previously, it was not worth the effort for many software developers to parallelize their programs since sequential processors would double performance every eighteen months. Clock speeds of commercial processors now remain relatively flat, but the number of cores available on a chip continues to increase. In order to increase program performance going forward, programmers have no choice but to turn to parallelism. Parallelism unfortunately presents many issues in regards to writing correct programs, introducing new classes of bugs. Consequently, there is still plenty of need and opportunity for new programming notations and tools to facilitate the control of parallelism, locality, processor load, and communication costs and to enable
portability of programs across multiple platforms.

This thesis presents ideas for data parallel operators to be used as a mechanism to facilitate development of parallel programs. Programs can be expressed as a sequence of data parallel operators applied to data types. As has also been observed by many others in the past [20, 30, 8, 14, 36, 4], writing programs as a sequence of data parallel operators applied to aggregates has important advantages over less structured notations.

- **Higher Level of Abstraction** - Expressing computations as a sequence of primitive data parallel operations on data types raises the level of abstraction in several ways. First, parallelism is encapsulated inside of the operators. Second, the details of the underlying architecture are hidden from the programmer. Raising the level of abstraction allows the programmer to write parallel programs without needing to know how the parallelism is implemented on the underlying architecture.

- **Conventional Notation** Programs written this way resemble conventional, sequential programs. This has the advantage that it separates reasoning about the algorithm from reasoning about optimizing for the best performance on the target hardware, simplifying implementation. If the programmer chooses to ignore parallelism or even use sequential implementations of the operators, then he or she can even reason about the semantics of the program sequentially. This provides a simpler model that allows programmers to quickly write code that does not contain errors introduced by parallelism and was the main motivation for the work on autoparallelization [37].

- **Portability** There are many different resources for parallel computing. Programmers can use SIMD units and multicore processors. Distributed memory machines are still popular for large computations. Newer devices like GPUs, Intel’s Larrabee, or the Cell processor provide programmers with mechanisms to exploit large amounts of data parallelism in a single device. Given the diversity of target architectures available, it
would be highly beneficial to programmers if their programs were portable across a wide range of devices. Expressing programs as a sequence of data parallel operators allows for this portability. All that is required is that suitable implementations of the required operators exist for the desired target architectures. This provides the benefit that the majority of programmers can use these primitives to write their programs while allowing the expert programmers to tune the implementations of these primitives to most efficiently use the available resources. For example, consider the array assignment \( A(1:N) = A(2:N+1) + 1 \). This could be implemented in shared-memory using a parallel loop. A distributed-memory implementation would likely partition the data among the different nodes, and then each node would compute its portion of the computation locally. Implementing this operation for distributed-memory requires communication of the values at the borders of these partitions. The implementations for these two platforms is different, but the use of higher-level data parallel operators allows programmers to more easily express their computations by hiding this complexity from the programmer.

- **Scalability** Since the number of processing resources is only going to increase, the primitive operators used must be scalable. Consequently these operators must be parallel, and more specifically, they must be data parallel, or phrased in terms of applying the same computation to different elements of a data type because scalable computations are data parallel.

- **Control of Determinacy**

  Programs expressed as a sequence of data parallel operators are determinate if the operators are pure functions, or have no side effects, and are separated by barriers to ensure that an operator completely finishes before the next one begins. By requiring all operators to be implemented as pure functions, the programmer can enforce de-
terminacy. However, if non-determinacy is allowed or desired, it can be encapsulated inside an operator that permits certain non-determinate semantics.

- **Facilitate Compilation and Autotuning**

The use of higher level data parallel operators can facilitate optimization by the compiler. A compiler that is aware of the semantics of such operators can perform domain-specific optimizations as well as optimizations across operators. The SPIRAL program generation system [45] generates implementations of DSP algorithms. Algorithms are expressed using high level mathematical operators which the compiler understands. When SPIRAL generates code, it is able to make use of its knowledge of the operators to generate optimized code, such as eliminating explicit permutations by propagating the proper array indices through formulas. A simple example of optimizations across operators is the elimination of unnecessary barriers. Consider a simple program that applies one operator to a set of data and then another operator to a different set of data. The programming model places a barrier between these operators, but the compiler can realize that this barrier is unnecessary, merging the two operators and eliminating the barrier when generating code.

When optimizing a program for maximum performance, some parameters may not be known until the program is run or installed on the target machine. In these cases, autotuning is a well-known technique to extract this performance when it cannot be obtained statically through compiler optimizations. By writing programs as a sequence of data parallel operators, the parameters that an autotuning system might explore can be made explicit as parameters of the operators. One example of such a parameter is the tile size for linear algebra routines. For example, ATLAS [50] is a system that uses autotuning to automatically generate high-performance BLAS routines. It performs a search to choose tile sizes that optimally exploit the target system’s caches. Another
possibility is to autotune the implementations of the operators themselves. Combining autotuning techniques with data parallel operators facilitates autotuning the whole program.

1.2 Contributions

This thesis makes several contributions, all in the general area of data parallel operators for the programming of multiprocessors and their current most popular incarnation, multicore. It accomplishes this through the development of Ravenna, a library of data parallel operators for shared-memory systems. Ravenna extends previous work on a data type for dense arrays called the Hierarchically Tiled Array, or HTA. The initial implementation of the HTA targeted distributed-memory machines and was built on top of MPI. However, the notation consists of data parallel operators that are amenable to any platform for which suitable implementations exist. Due to the rising importance of multicore/manycore architectures, Ravenna targets this class of machine. The HTA data type was only applicable to computations based on dense arrays. Ravenna can support arbitrary data types, enabling programmers to write data parallel computations based on other data types such as sets or graphs.

Codes expressed as a sequence of data parallel operators are portable across any platform for which suitable implementations of the necessary operators exist. Ravenna demonstrates this through the example of a parallel solver for banded linear systems. The resulting implementation is both cleaner than its counterpart implemented using Fortran and MPI and obtains comparable performance. The same code written using the Ravenna library can be executed either on shared-memory or on distributed-memory using the HTA library.

Tiling is a very important primitive for controlling both parallelism and locality. It is essential for obtaining the best performance for numerical applications, and its use for dense
array computations was described in [2]. Computations for structures other than dense arrays can also benefit from tiling once the data types and associated operators are extended with tiles. New primitives are necessary for properly tiling different data structures. Ravenna provides programmers with several mechanisms for tiling data types. In particular for data structures other than dense arrays, it provides a generalized approach called functional tiling. Functional tiling provides programmers with a separation of concerns between implementing a computation and how to tile it. Functional tiling in this way also acts as a tuning mechanism that allows programmers to tune the performance of their codes by plugging in different tiling strategies.

Next, this thesis evaluates the programming model of expressing programs as a sequence of higher level data parallel operators through examining several applications from different domains written in Ravenna. These applications include simple microbenchmarks used to compare against another shared-memory programming library, a solver for banded linear systems, n-body simulation, clustering, and discrete optimization. The evaluation shows that these programs can be elegantly expressed by the programming model, and that the model’s applicability is not limited to computations based on dense arrays. Particularly, it shows that the resulting programs resemble conventional, sequential programs, simplifying programmer effort and that the available abstractions provided by Ravenna allow programmers to obtain good parallel performance.

1.3 Thesis Organization

The rest of the thesis is laid out as follows. Section 2 presents Ravenna, a library of higher-level data parallel operators for shared-memory systems. Section 3 presents a comparison of Ravenna to Intel’s Threading Building Blocks, a library of task parallel primitives for shared-memory programming. Next, Section 4 presents an example of using data parallel
operators for sparse computations with the implementations of clean, high-performance, and portable solvers for banded linear systems implemented using the Ravenna library. Section 5 illustrates how Ravenna can express data parallel computations on non-array data types through examples of n-body simulation, clustering, and discrete optimization and also studies how functional tiling can be used to optimize the performance of programs. Section 6 discusses related work and compares Ravenna to several other approaches. Finally, Section 7 presents conclusions and future work.
Chapter 2

Ravenna, A Library of Data Parallel Operators

This chapter presents Ravenna, a library of higher level data parallel operators for programming shared-memory multiprocessors. Ravenna provides several data parallel operators that operate on tiled data types. It provides several tiling abstractions to the programmer that enable expressing many different kinds of computations. Ravenna is based on an earlier library for data parallel computations on dense arrays called the Hierarchically Tiled Array, or HTA. It extends this work by supporting new tiling abstractions for shared-memory systems as well as adding support for non-array data types.

2.1 The Hierarchically Tiled Array

The Hierarchically Tiled Array [3, 18], or HTA, data type extends earlier work on data parallel array languages with explicit tiling. An HTA object is a tiled array whose elements can be either scalars or tiles. HTAs can have several levels of tiling, allowing them to adapt to the hierarchical nature of modern machines. Figure 2.1 presents two examples of how HTAs
can exploit hierarchical tiling. For example, tiles in the outermost level can be distributed across the nodes in a cluster; then, the tile in each node can be further partitioned among the processors of the multicore node. Likewise, an array could have several levels of tiling to block the computation onto the different levels of cache found on a processor.

The HTA data type makes tiles first class objects that are explicitly referenced and extends traditional Fortran 90 style array operations to function on tiles. Figure 2.2 illustrates the ways in which HTAs can be indexed. HTAs permit indexing of both tiles and scalars. One uses () to refer to tiles and [] to refer to scalar elements. This way, A(0,0) refers to the top left tile of HTA A, and A(0,1)[0,1] refers to the element [0,1] of the top right tile of HTA A. Also, HTAs support the triplet array notation in order to index multiple scalars and/or tiles, as shown in Figure 2.2 when accessing the two bottom tiles of A by using A(1, 0:1). Since the HTA has been implemented as a library for C++, the triplet notation is actually represented by a Triplet object. Consequently the previous example is more accurately rendered into code as A(1, Triplet(0,1)). The simpler : notation will be used in the remainder of this thesis for simplicity. Scalars can also be accessed in a flattened fashion.
that ignores the tiling structure of the HTA, as shown in the example when accessing the element A[0,3]. This flattened notation is useful for several tasks. Examples include initialization, calling functions from other libraries, gradual transformation from sequential to parallel code, or operations like global pivoting that ignore the tiling structure.

The HTA library implements several data parallel operators that can be applied to HTA objects including element-by-element operations, maps that apply a user-specified function to each tile of an object, reductions, and others. HTAs generalize the notion of conformability of Fortran 90. When two HTAs are used in an expression, they must be conformable. Specifically, they are conformable if they have the same tiling structure and tile sizes. Also, HTAs are conformable with scalars, conceptually replicating the scalar to match the dimensions of the leaf tiles of the HTA.

HTA programs are expressed as a sequence of data parallel operators applied to HTA objects. Operators are separated by barriers, guaranteeing that each operation will complete before the next one can begin. HTA programs appear sequential to the programmer as all parallelism is encapsulated inside the operators. The tiling of HTA objects, specifically the numbers and sizes of tiles, is chosen by the programmer both to control the granularity of parallelism and to enhance locality.
The HTA data type was initially implemented as libraries for both C++ and MATLAB. The C++ library targeted distributed-memory using MPI.

2.2 Programming Model

As is the case with the HTA library, the programming model for Ravenna consists of two parts: tiled objects and data parallel operators. Programs are expressed as a sequence of data parallel operators applied to tiled objects, where the operators are separated by an implicit barrier. Tiled objects are objects whose elements have been distributed across the tiles of the object such that an element only belongs to one tile. For arrays, this is equivalent to using hyperplanes to divide the array into chunks. For other data types like sets, this requires a function to map elements to tiles. Tiling in Ravenna can be hierarchical, that is, objects can have several levels of tiling. This is beneficial as modern machines are hierarchical. Tiles are first class objects, that can be explicitly referenced in the program. Although the concept of tiles also exists in languages such as HPF [20], among others, in the form of distributions, tiles were only used by the compiler, and could not be referenced by the programmer. This, many times, hindered the process of program optimization, as the programmer had to rely upon the compiler to make good decisions. The Ravenna approach raises the level of abstraction, while still giving the programmer control over performance since he or she knows how data are mapped to tiles and thus can easily reason about where communication occurs. Directly referencing tiles allows the programmer to do assignment between tiles, which will result in communication. It also facilitates the expression of cache oblivious algorithms, where tiles are dynamically re-partitioned as the algorithm evolves or block algorithms that map naturally to tiles. Finally, it is important to have flexible mechanisms to tile objects, as not all the computations can be mapped to simple distributions. This is further discussed in Section 6.
### Primitive Operators

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
<th>Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>+ - * /</td>
<td>Applies a function, such as '+', to each element of an object or corresponding elements of conformable objects.</td>
<td>Usage: $a + b$</td>
</tr>
<tr>
<td>=</td>
<td>Assigns the contents of one or more tiles to another. Requires the left-hand side and right-hand side to be conformable.</td>
<td>Usage: $a = b$;</td>
</tr>
<tr>
<td>map(func(...), ...)</td>
<td>The arguments are a programmer-defined function <code>func()</code> and any number of conformable tiled objects. Map applies <code>func()</code> in parallel to each tile of the object and the corresponding tiles of any additional arguments. Assumes that the computation in a tile is independent of the computation in other tiles.</td>
<td>Usage: <code>a.map(SomeFunction(), b, c, 0)</code>;</td>
</tr>
<tr>
<td>reduce(op, d)</td>
<td>Applies an operation, <code>op</code>, on an object to produce an object of lesser rank where <code>d</code> is the dimension along which <code>op</code> is performed. Simple examples include summing all the elements in a 1D tile or summing all the elements in the columns of a tile of a 2D array.</td>
<td>Usage: <code>a.reduce(plus&lt;double&gt;(), 0)</code>;</td>
</tr>
<tr>
<td>spread(d, num)</td>
<td>Replicates a tile <code>num</code> times in the dimension <code>d</code>.</td>
<td>Usage: <code>a.spread(0, 4)</code>;</td>
</tr>
<tr>
<td>mapReduce(mfunc(), rfunc())</td>
<td>Takes two programmer-specified functions, <code>mfunc()</code> and <code>rfunc()</code>. It applies a <code>mfunc()</code> to each tile and produces (key, value) pairs. The <code>rfunc()</code> is then applied over the values based on their respective keys. This implements the operator described in [11].</td>
<td>Usage: <code>a.mapReduce(count(), sum());</code></td>
</tr>
</tbody>
</table>

Table 2.1: Ravenna Primitives
Table 2.2: Ravenna Indexing Operators

<table>
<thead>
<tr>
<th>Indexing Operators</th>
</tr>
</thead>
<tbody>
<tr>
<td>()</td>
</tr>
<tr>
<td>Indexes a single tile or a range of tiles of the object.</td>
</tr>
<tr>
<td><em>Usage</em>: <em>a</em> = <em>b</em>(0,0);</td>
</tr>
<tr>
<td>[]</td>
</tr>
<tr>
<td>Indexes data elements inside of a tile.</td>
</tr>
<tr>
<td><em>Usage</em>: <em>int</em> <em>x</em> = <em>a</em>[0,0];</td>
</tr>
<tr>
<td>Triplet</td>
</tr>
<tr>
<td>Allows programmers to index ranges of tiles or elements.</td>
</tr>
<tr>
<td><em>Usage</em>: <em>a</em> = <em>b</em>(Triplet(0,3))[Triplet(0,n-1)];</td>
</tr>
</tbody>
</table>

Table 2.1 lists the primitive operators provided by Ravenna. Operators are applied in parallel across all the tiles of an object. Operators are aware of tiles and any hierarchy they might have. Operations should be defined so that computation on a tile is independent of computation on others. Thus, tiles in Ravenna correspond to the logical unit of independent computation. Basic arithmetic operators operate on corresponding elements of conformable objects. Consider the statement *A* = *B* + *C*. If *B* and *C* have the same tiling structure and tile sizes, then each element of each tile of *B* is added with its corresponding element of each tile of *C*. Another possibility is that *C* is a single tile of the same size as the tiles of *B*. In this case, each element of every tile of *B* is added to its corresponding element in the single tile of *C*. Assignment has similar behavior to the basic arithmetic operators. However, assignment can imply communication between tiles. In the distributed-memory HTA library, this communication translates into calls to MPI. In Ravenna, the data is directly accessed.

`map` is applied to tiled objects and takes a function object as input. This object must have an overloaded () operator that takes tiled objects as parameters. `map` calls this object’s overloaded operator on each tile of the object on which it was called. For example, *A*.map( *Functor()* , *B* , 0) means that *Functor*’s overloaded () operator accepts two tiles, one of *A* and one of *B*. The optional 0 parameter states that *Functor* is applied at level 0 of the tiling.
Ravenna Tiling Operators

Creation

`create(# levels, (Tuple, ...))`

Used to create tiled objects. The programmer specifies the number of levels of tiling
that the object will have as well as a sequence of tuples that dictate the sizes of
tiles at each level. An object with 0 levels of tiling corresponds to a regular array.
The example usage shows how to create a tiled array of 2x2 tiles of N x N elements.
The type of the elements and the number of dimensions are template parameters.
Usage:

```c++
Tuple<int, 2>::Seq tiling = (Tuple<2>(2, 2), Tuple<2>(N, N));
a = TiledArray<int, 2>::create(1, tiling);
```

Dynamic Partitioning

`addPartition(Tuple loc)`

Used to dynamically split a tile at runtime. `loc` specifies the location in the tile
where the split will occur. The usage examples shows how to split a 1D tile `A`
of 10 elements into two tiles, one of four elements and one of six.
Usage: `a.addPartition(Tuple<1>(4));`

`rmPartition(int line)`

Used to dynamically combine tiles at runtime. `line` refers specifies the partition line
to remove. The usage example shows how to undo the operation completed by the
example for `addPartition`. In this case, there is only a single partition line since
there are only two tiles.
Usage: `a.rmPartition(1);`

Functional Tiling

`tile(func())`

Used by functional tiling to rearrange all the elements of an aggregate among the
tiles so that each tile contains the elements that map to it. `func` is a function object
with an overloaded `()` operator that for a given element returns the tile to which
it maps.
Usage: `a.tile(TilingStrategy());`

Table 2.3: Ravenna Tiling Primitives
where the leaf tiles that contain the actual data elements are found. If A had another level of tiling, one could call map at level 1. This would call Functor with a tile of A and a tile of B. Each tile of A would have additional tiles since it has another level of tiling. This type of usage of map is useful expressing computations that might use a spread operator without replicating data.

mapReduce is not the same as applying a map and then a reduce to an object. Consider the following example. Let V be a tiled vector where each tile contains the words of a page of a text. Let the programmer-specified function count take a tile of V and return a tile that consists of tuples of the form (word, # of occurrences). Let sum be a function that takes two tiles and returns a single tile that contains tuples that represent the sum of the number of occurrences of each word in the tiles. One can now count the number of occurrences of each word in V with the statement V.mapReduce( count(), sum() ).

Section here about indexing operations and citing 2.2. Ravenna also provides several indexing operators that allow programmers to access either tiles or data elements of objects. The () operator is used to index tiles of an object, while the [] operator is used to index the elements of a tile. If [] is applied to a tile that is not a leaf tile, as in it has additional levels of tiling, then [] is used to access elements in a flattened fashion, that ignores the lower levels of tiling. The Triplet object is used to represent index ranges when accessing tiles or elements. For example, a(Triplet(0,N-1)) would refer to the first N tiles of a.

2.3 Tiling

The key difference between Ravenna and other data parallel libraries is tiling. Tiling is an important primitive with many benefits. It is a natural way to express data distribution. It also lets programmers control the granularity of their computations by adjusting the tile size. Tiling is also a natural primitive for exploiting locality. By controlling how data are
partitioned into tiles, programmers can tune their computations to obtain the best performance on the desired target platform. This has several advantages for the programmer. Domain programmers need not concern themselves with how to tile an object. They need only be aware that an object is tiled and write their programs by specifying a series of data parallel operators that are applied to tiled objects. Expert programmers can then implement different tiling strategies in order to tune the performance of the computation to the target platform. In this way, tiling provides a separation of concerns between algorithmic correctness and the process of program optimization. Table 2.3 shows the primitives that Ravenna provides for creating and modifying the tiling of objects.

The most basic of Ravenna’s primitives is tile creation. The create operator allows programmers to create tiled objects that have a programmer-specified tiling structure. Programmers specify the number of levels of tiling, the number of tiles per level, and the type of elements that exist at the leaves such as arrays, sets, primitive data types, or others. The desired tiling structure is treated as an input to create, so programmers can treat the tile size as a parameter whose value only needs to be determined at the time of creation. For objects such as dense arrays, creating a tiled object is simple as they have an implicit way to tile them that can be thought of as splitting the array with hyperplanes.

Another way of tiling an array is to use dynamic partitioning. The creation mechanism previously described is sufficient to express many regular numerical computations. However, it is an insufficiently powerful abstraction to represent less regular computations. Examples include computations whose tiling is data-dependent or can evolve throughout the computation. Dynamic partitioning allows programmers to split and combine tiles of arrays at runtime. Similarly to creating the initial tiles, dynamic partitioning of arrays can be viewed as splitting a tile by a hyperplane. An example of dynamic partitioning is shown in Section 3.2 with a code that performs the merging of two sorted sequences in parallel. This example uses dynamic partitioning to create nested parallelism that divides the input arrays
into smaller and smaller tiles until a suitable size is reached where the merge operation can be performed sequentially.

2.3.1 Functional Tiling

Ravenna’s creation and dynamic partitioning primitives provide good abstractions for tiling many computations. For example, the ordering of the elements in an array facilitates tiling because all that is needed is to specify the chunk of consecutive elements that belong to the same tile. Ravenna, however, provides support for data types other than arrays, such as graphs, trees or sets. These data types are not ordered, or at least do not have a single order. Higher level abstractions are therefore needed to tile these data types for some applications. Towards this end, Ravenna provides a new abstraction called functional tiling. Programmers specify a function that defines the tiling for an aggregate. This function maps each element to the appropriate tile. Just as tiles in Ravenna are first class objects that programmers can use, functional tiling turns the specification of tiling itself into a first class object.

Functional tiling in Ravenna is achieved through a communication operator called tile. As seen in Table 2.3, tile is applied to tiled objects and takes a function that maps elements to tiles as input. The operator applies the function to each element across all the tiles in parallel, determining those that map to different tiles. These elements are then communicated to the tiles to which the function maps them. Conceptually, this operator can be viewed as a specialized application of MapReduce. The map applies the tiling function to each element of the object, creating tuples of the form (tile #, element). The reduce then applies set union to all the elements that map to the same tile, resulting in each tile containing only the elements that map to it.

The tile operator can be used to provide the initial partitioning according to the tiling function, rearrange the data if another tiling is desired, or ensure that the tiled object represents the tiling specified by the supplied function when the computation creates new
data elements that might need to be mapped to the appropriate tile. An example of this is shown in Figure 2.3. Here, an object has three tiles. Each tile contains an element whose shade indicates the tile to which it maps. An operation is now performed on each tile, generating new data. Some elements are in the correct tiles while others are not. tile is then applied to the object, remapping the elements to their correct tiles. Chapter 5 presents a more concrete example of the use of the tile operator by a data-parallel breadth-first search computation. This computation partitions the vertices of a graph into tiles. Each iteration, every tile processes the vertices it contains and identifies new neighbors to explore. These neighbors may map to different tiles than the one whose vertex found them. In this case, the tile operator is used to notify the appropriate tiles of the neighbors to process in the next iteration.

Note that functional tiling is itself a generalization of Ravenna’s other tiling primitives. For example, tiling of arrays could be represented by a tiling function that takes array indices as input and calculates the destination tile based on the desired block size. This would allow programmers a simple interface for non-homogeneous tiling of arrays, something not supported by the create operator. Likewise, dynamic partitioning could be represented by a function that splits a range around a given pivot. These other tiling primitives are still useful abstractions to provide to programmers when they are sufficient to express the desired computation as they can do so with lower overheads than functional tiling. However, functional tiling has greater flexibility, as it can be applied to any data structure and enables the expression of more computations.

2.4 Implementation

Ravenna is built as a library for C++. The implementation is based on a sequential implementation of the Hierarchically Tiled Array [25], and it targets shared-memory multi-
processors, due to their increasing importance. However, porting array programs written in Ravenna to run on top of MPI using the HTA library only requires changing the header file for the library to include `htalib_mpi.h` instead of `ravenna_shmem.h`. However, Ravenna programs using functional tiling or dynamic partitioning will only run using the Ravenna library, as neither functional tiling nor dynamic partitioning is implemented in the HTA library. Parallelism is implemented in Ravenna using the Intel Threading Building Blocks (TBB) [40]. This gives the library flexibility to use the work-stealing scheduler that has been built into TBB as well as their affinity partitioners. Many operators have a straightforward parallel implementation. An example of such an operator is `map`. The sequential implementation iterates over all the tiles of an HTA and applies the programmer-specified function to each tile. A parallel implementation simply transforms this loop into a parallel loop. Similarly, `reduce` is implemented with TBB’s reduction operators.

Ravenna adds support for arbitrary data types. The HTA library only contained support for tiled arrays. These were represented in memory by a flat array. The library then maintained additional information that mapped leaf tiles to the appropriate offsets in the flattened array. This solution was acceptable for primitive data types such as integers or doubles. Ravenna implements support for non-primitive data types in a generic fashion, in
order to benefit from many operators provided by the Standard Template Library. Instead of providing our its implementations of different data structures such as sets or graphs, Ravenna allows the programmer to specify the desired container class for the leaves of the tiled object since the choice of data structure is an important consideration for program performance. A tiled set, for example, is thus represented inside the library as an array of set containers. This design decision also provides support for data types whose size can dynamically grow or shrink during execution. Currently, the implementation uses static tiling to define the initial tiling structure for a tiled object, that is, the programmer specifies the type of object and numbers and levels of tiles. The programmer can then use functional tiling to communicate data between tiles when appropriate.

Support for functional tiling inside the library comes from the implementation of the tile operator. This operator is applied to tiled objects and takes a programmer-specified function object as input. This object contains two methods. First, it must overload the C++ () operator similarly to those used by map. This operator is applied to each element in the tile and returns the tile to which it maps. The current implementation only supports a single level of tiling, but this could be extended to return a tuple instead of a scalar in order to support multiple levels of tiling. The object passed to tile must also contain an insert method. This is necessary in order to call the appropriate method for inserting elements into the programmer-chosen data type used to store the elements in the leaves. The current implementation of tile is as follows. Each tile of the object on which tile is called is associated with an array of "buckets". Each tile iterates through its elements in parallel, applying the function object’s () operator to each element and placing it into the correct bucket dictated by the result of the function object’s (). Each tile then waits at a barrier until all tiles have placed their elements into buckets. Next, each tile iterates in parallel through all the buckets that correspond to itself, calling the function object’s insert method on every element inside a bucket. Note that while the current implementation of
a.map( op1(), b);
b.tile( func() );
b.map( op2(), ...);

Figure 2.4: Example with the tile operator

tile only targets shared-memory, a distributed-memory implementation would only require each processor to gather all the buckets that are associated with the tiles mapped to it.

Operators in Ravenna end in barriers. These barriers should have little impact in performance as long as each tile can perform sufficient computation. Programmers can recognize operations with little computation and manually combine them into one call to map in order to reduce the overhead of synchronization. This, of course, requires that no dependences are violated by fusing operators. If Ravenna had a static compiler or JIT that could identify such dependences, it could potentially eliminate some of these barriers for the programmer.
Chapter 3

Comparison to TBB

Once the Ravenna library had been implemented, a comparison was performed of the performance and programmability of simple applications written using the library against codes written using Intel TBB, the library upon which Ravenna is built. The goal of this evaluation was not to see if programs written in Ravenna beat the performance of TBB programs, but rather if they obtain roughly equal performance with less programmer effort.

3.1 The Intel TBB library

The Intel Threading Building Blocks (TBB) [41] is a library for shared-memory programming. It does not base the specification of parallelism on data operations that are inherently parallel, which is the HTA approach. Rather, parallelism is achieved by defining tasks that can be performed concurrently. The task scheduler then maps tasks to available hardware threads. When there are more threads available than tasks, it can split an existing task in several smaller tasks.
3.1.1 TBB operations

The element-by-element operation, reduction, and scan constructs are implemented in the TBB library using the `parallel_for`, `parallel_reduce` and `parallel_scan` algorithm templates respectively. The TBB library also includes the algorithm templates `parallel_do`, which supports unstructured workloads where the loop limits are not known at the beginning of the loop, and `pipeline`, which is used when there is a sequence of stages that can operate in parallel on a data stream.

The `parallel_for`, `parallel_reduce` and `parallel_scan` algorithm templates accept two basic parameters: a `range` defining loop limits, and a function object representing the body of the parallel loop. This object overloads the () operator and defines the operation to be performed on the range passed as input. The range is split recursively into subranges by the task scheduler and mapped onto physical threads. The TBB library provides standard ranges, such as `blocked_range`, which expresses a linear range of values in terms of a lower bound, an upper bound, and optionally, a grain size, which is a guide for the amount of work performed by a task.

Programmers can define their own range classes implementing specific policies to decide when and how to split or how to represent the range. An example of a programmer-defined range will be shown in Section 3.2.

Some additional features present in the TBB library are a scalable and efficient memory allocator for multithreaded programs, mutual exclusion structures for explicit thread synchronization, support for atomic operations on primitive data types, and thread-aware timing utilities.

The codes used in this comparison were taken from the chapter 11 of [41], which contains examples of parallel implementations of algorithms using TBB [47]. These codes were chosen because at the time the experiments were done there were very few codes available written using TBB, and they cover a diversity of parallel computation patterns. The following
sections describe them and highlight the key differences between the TBB and Ravenna implementations.

### 3.2 Parallel Merge

This code merges, in parallel, two sorted sequences into an output sorted sequence. The algorithm operates recursively as follows:

1. If the sequences are shorter than a given threshold, they are merged sequentially. Otherwise, Steps 2-5 are performed.

2. The sequences are swapped if necessary so that the first sequence, \([begin1, end1]\) (the notation \([\) indicates that the first value of the interval is included but not the last one), must be at least as long as the second sequence \([begin2, end2]\).

3. \(m1\) is set to the middle point in the first sequence. The item at that location is called \(key\).

4. \(m2\) is set to the point where \(key\) would fall in the second sequence.

5. Subsequences \([begin1, m1]\) and \([begin2, m2]\) are merged to create the first part of the merged sequence and subsequences \([m1, end1]\) and \([m2, end2]\) are merged to create the second part. Both operations take place in parallel with each other.

The TBB implementation of this algorithm is based on a parallel for. The subdivision of the sequences is implemented using an object of the ad-hoc range class ParallelMergeRange whose definition is shown in Figure 3.1. The predicate is divisible performs the test in step 1. The ParallelMergeRange class has two constructors. The first one, shown in lines 8-22, contains the dummy variable split. This argument is used by the TBB library to flag a Range constructor that is used to split an input Range in two. The constructor builds
template<typename Iterator>
struct ParallelMergeRange {
    ...
    bool empty() const { return (end1 - begin1) + (end2 - begin2) == 0; }
    bool is_divisible() const {
        return std::min( end1 - begin1, end2 - begin2 ) > grainsize;
    }
    ParallelMergeRange( ParallelMergeRange& r, split ) {
        if( (r.end1 - r.begin1) < (r.end2 - r.begin2) ) {
            std::swap(r.begin1, r.begin2);
            std::swap(r.end1, r.end2);
        }
        Iterator ml = r.begin1 + (r.end1 - r.begin1) / 2;
        Iterator m2 = std::lower_bound( r.begin2, r.end2, *ml );
        begin1 = ml;
        begin2 = m2;
        end1 = r.end1;
        end2 = r.end2;
        out = r.out + (ml - r.begin1) + (m2 - r.begin2);
        r.end1 = ml;
        r.end2 = m2;
    }
    ParallelMergeRange( Iterator begin1__, Iterator end1__,
                        Iterator begin2__, Iterator end2__,
                        Iterator out__ ) :
        begin1(begin1__), end1(end1__),
        begin2(begin2__), end2(end2__), out(out__)
    {}
};

struct ParallelMergeBody {
    void operator() ( ParallelMergeRange<Iterator>& r ) const {
        std::merge( r.begin1, r.end1, r.begin2, r.end2, r.out );
    }
};

Figure 3.1: TBB Parallel Merge
typedef TiledArray<float,1> TA_1;
struct Merging {

void operator () (TA_1 output_, TA_1 input1_, TA_1 input2_)
{
    ... 
    if (input1_.size > GRAINSIZE) {
        size1 = input1_.shape().size()[0];
        size2 = input2_.shape().size()[0];
        if( size1 < size2 ) {
            h2=input1_; h1=input2_; 
            std::swap(size1, size2);
        } else {
            h1=input1_; h2=input2_; 
        }
        int pos = h2.lower_bound(h1[(size1 - 1) / 2]);
        h1.addPartition((size1 - 1) / 2);
        h2.addPartition(pos);
        output_.addPartition(pos + (size1 - 1) / 2);
        output_.map(Merging(), h1, h2);
    } else {
        ... 
        std::merge(...);
    }
}

Figure 3.2: Ravenna Parallel Merge
a new range that stores one of the halves of the original `Range` and modifies the original `Range`, received as first parameter, to hold the other half. This constructor performs the steps described in steps 2-5 of the algorithm. The other constructor is a conventional constructor that assigns the iterators used to define the range. The overloaded () operator of the struct `ParallelMergeBody` passed to `parallel_for` simply performs the merge sequentially by means of a `std::merge`.

The Ravenna version is based on `map`. In the function applied by `map`, if the sequences are larger than a given threshold, steps 2-5 proceed. This part of the algorithm, shown in Figure 3.2, is implemented using Ravenna’s dynamic tiling. Lines 20-22 add new partitions to the two inputs and to the output at the locations described in step 3 of the algorithm. This is performed using the `addPartition` method, which accepts the position at which a new partition line is to be added, creating new tiles. The position where `key` would fall in the second sequence, mentioned in step 4 of the algorithm, is calculated in line 13 using the function `lower_bound`, which returns the index of the first element of the tiled array that is equal or larger than its argument.

Line 24 calls `map` recursively. Since the tile of each array was partitioned into two tiles, the recursive call is executed on each tile in parallel. The recursion finishes when the sequences to merge are smaller than a given threshold, then step 1 is performed.

### 3.3 The Game of Life

The ”Game of Life” is a problem which opened the mathematical research field of *cellular automata*. The game is played in a two-dimensional orthogonal grid of square cells, each of which is in one of two possible states: *live* or *dead*. Every cell interacts with its eight neighbors, which are the cells that touch the cell horizontally, vertically or diagonally. In every step of this evolution, each cell lives, dies, stays empty or is born based on a simple
... class tbb_parallel_task {
  ...
  static void set_values(Matrix* source, char* dest) {
    ...
    m_source = source; m_dest = dest;
    ...
  }
  ...
  void operator()( const blocked_range<size_t>& r ) const {
    ...
    begin=(int)r.begin();
    end=(int)r.end();
    Cell cell;
    for (int i=begin; i<=end; i++)
      *(m_dest+i) = cell.CalculateState(
        m_source->data, m_source->width,
        m_source->height, i);
    }
  }
  ...
  for (int counter=1; counter<NSTAGES; counter++)
    parallel_for (blocked_range<size_t>(begin, end, grainSize),
    tbb_parallel_task());
  ...
}

Figure 3.3: TBB Game of Life

decision depending on the surrounding population (number of neighbors). The rules that determine the evolution of life are:

1. Life persists in any cell where it is also present in two or three of their eight neighboring cells and otherwise disappears (from loneliness or overcrowding).

2. Life is born in any empty cell for which there is life in exactly three of the eight neighboring cells.

The decisions about each generation are taken based on the state of the cells in the previous generation, so the problem is fully parallel.

The parallel version decomposes the two-dimensional space of cells in a number of regions, and the decisions for the next generation are taken in parallel in the different regions. This is
struct EvolutionOp {
    void operator() (TiledArray data_source, TiledArray data_dest) {
        ... 
        CellHTA cell;
        size=data_dest.shape().size();

        for(int i=0; i<size[0]; i++) {
            for(int j=0; j<size[1]; j++) {
                data_dest[i][j]=cell.CalculateState(data_source, (i, j));
            }
        }

        Overlap<2>* ol=new Overlap<2>(Tuple<2>(1,1), Tuple<2>(1,1), PERIODIC);
        data= TiledArray::create(1, ((SIZEX/NTILESX,SIZEY/NTILESY),
            (NTILESX,NTILESY)),
            ol);

        for(int counter=1; counter<NSTAGES; counter++)
            data.map(EvolutionOp(), data, 0);
    }
}

Figure 3.4: Ravenna Game of Life

CalculateState(...) {
    ... 
    total += GetAdjacentCellState(source, x, y, cellNumber, upperLeft);
    total += GetAdjacentCellState(source, x, y, cellNumber, upper);
    total += GetAdjacentCellState(source, x, y, cellNumber, upperRight);
    total += GetAdjacentCellState(source, x, y, cellNumber, right);
    total += GetAdjacentCellState(source, x, y, cellNumber, lowerRight);
    total += GetAdjacentCellState(source, x, y, cellNumber, lower);
    total += GetAdjacentCellState(source, x, y, cellNumber, lowerLeft);
    total += GetAdjacentCellState(source, x, y, cellNumber, left);
    ... 
}

Figure 3.5: TBB Calculate State
GetAdjacentCellState(...) {
    char cellState = 0;     // return value
    bool onTopRow = onBottomRow = onLeftColumn = onRightColumn = false;

    switch (cp) {
        case upperLeft:
            if (onTopRow && onLeftColumn)
                return *(source +((x∗y)−x));
            if (onTopRow && ! onLeftColumn)
                return *(source +(((x∗y)−x)+(cellNumber−1)));
            if (onLeftColumn && ! onTopRow)
                return *(source +((cellNumber−(x∗2))−1));
            return *(source +((cellNumber+(x−1))));
            break;
        case upper:
            if (onTopRow)
                return *(source +(((x∗y)−x)+cellNumber));
            return *(source+(cellNumber+1));
            break;
        case upperRight:
            if (onTopRow && onRightColumn)
                return *(source +((x∗y)−x));
            if (onTopRow && ! onRightColumn)
                return *(source +(((x∗y)−x)+(cellNumber+1)));
            if (onRightColumn && ! onTopRow)
                return *(source+(cellNumber−(x−1)));
            return *(source +((cellNumber+(x∗2))−1));
            break;
        case right:
            if (onRightColumn)
                return *(source +((cellNumber−(x−1))));
            return *(source+(cellNumber+1));
            break;
        case lowerRight:
            if (onBottomRow && onRightColumn)
                return *(source +((cellNumber−(x∗2))));
            if (onBottomRow && ! onRightColumn)
                return *(source+(cellNumber+(x∗2)+1));
            if (onRightColumn && ! onBottomRow)
                return *(source+(cellNumber−(x−1)));
            return *(source +(((cellNumber+(x∗2)))+1));
            break;
        case lower:
            if (onBottomRow)
                return *(source+(cellNumber−((x∗y)−x)));
            return *(source+(cellNumber+x));
            break;
        case lowerLeft:
            if (onBottomRow && onLeftColumn)
                return *(source +((x−1)));
            if (onBottomRow && ! onLeftColumn)
                return *(source +((x∗y)−x−1));
            if (onLeftColumn && ! onBottomRow)
                return *(source +((cellNumber+(x∗2))−1));
            return *(source+(cellNumber+(x−1)));
            break;
        case left:
            if (onLeftColumn)
                return *(source+(cellNumber+(x−1)));
            return *(source+(cellNumber−1));
            break; }
    return cellState;
}

Figure 3.6: TBB GetAdjacentCellState
typedef TiledArray<int,2> TA_2;

CalculateState(TA_2 data, Tuple<2> cellCoordinates) {
    total += data[cellCoordinates[0]-1, cellCoordinates[1]-1];
    total += data[cellCoordinates[0], cellCoordinates[1]-1];
    total += data[cellCoordinates[0]+1, cellCoordinates[1]-1];
    total += data[cellCoordinates[0]+1, cellCoordinates[1]+1];
    total += data[cellCoordinates[0]-1, cellCoordinates[1]+1];
    total += data[cellCoordinates[0]-1, cellCoordinates[1]];}

Figure 3.7: Ravenna Calculate State

implemented in the TBB and Ravenna versions using a `parallel_for` and a `map` respectively. Both implementations can be seen in Figures 3.3 and 3.4. Tiling stencil codes traditionally requires the use of shadow regions. These regions represent elements found at the borders of neighboring tiles and are usually explicitly managed by the programmer. A feature of the HTA in Ravenna called Overlapped Tiling was introduced in [18]. Overlapped tiling handles the creation and management of shadow regions for the programmer. When creating a tiled array, the programmer can specify an Overlap object that represents the amount of overlap between tiles in each dimension as well as specify the behavior at the boundaries. The shared-memory implementation of Overlapped Tiling does not require extra memory between tiles to implement the overlap. Since the program has a single address space, the data in other tiles is directly read. Note that this places restrictions on its use for reasons of correctness, namely, that the same array cannot be both read and written in an operation. Line 12 of Figure 3.4 shows the creation of an Overlap object of size one in both positive and negative directions of each dimension of the board. The last argument of the constructor of the overlap region in line 12, PERIODIC, determines which values will contain the shadow cells in the boundary regions of the board. PERIODIC means that they contain the value located in the other side of the matrix. For example, the upper cell of position (0, 0) would
be \((N - 1, 0)\) where \(N - 1\) is the size of the first dimension.

Overlapped Tiling greatly eases the implementation of another part of the code with respect to the TBB version. The class `Cell` is used to model the behavior of an isolated cell of the board. Method `CalculateState` of this class has to compute the new state for each cell. In the TBB version, most of the time, the state of cell \((i, j)\) depends on the state of its neighbors located in positions: \((i - 1, j - 1), (i - 1, j), (i - 1, j + 1), (i, j - 1), (i, j + 1), (i + 1, j - 1), (i + 1, j)\) and \((i + 1, j + 1)\). But in the boundary regions, the neighbor values must be read from the other side of the matrix. This complicates the implementation of `CalculateState` in TBB, shown in Figure 3.5. The TBB implementation of `CalculateState` uses a helper method called `GetAdjacentCellState`, shown in Figure 3.6, to calculate the appropriate neighbor of a cell in a given direction. In the case of the Ravenna version, shown in Figure 3.7, the Overlap object uses PERIODIC boundary conditions to specify this behavior, and the neighbors can be computed using standard Ravenna indexing and relative addressing.

### 3.4 Average

This algorithm calculates, for each element in a vector, the average of the previous element, the next element and itself, and the result is stored in an output vector. The TBB code implements this algorithm using the `parallel_for` construct and is shown in Figure 3.8. In this code, the first and the last elements of the array are special cases, since they don’t have previous and next elements, respectively. This is solved by adding elements at the beginning and the end of the array which are filled with zeros as shown in lines 15-18 of the code. In line 20, the task scheduler object is created and initialized with 4 threads. The task scheduler is the engine in charge of mapping tasks to physical threads and of the thread scheduling. It must be initialized before executing any TBB parallel constructs.

The first argument of the `parallel_for` in line 23 is a range which includes the whole
class Average {
public:
    float *input, output;

    void operator()( const blocked_range<int>& range ) const {
        for ( int i = range.begin(); i != range.end(); ++i )
            output[i] = (input[i-1] + input[i] + input[i+1]) * (1/3.0f);
    }

const int N = 100000;
static int nThreads = 4;

int main( int argc, char* argv[] ) {
    float raw_input[N+2], output[N];
    raw_input[0] = 0;
    raw_input[N+1] = 0;
    float* padded_input = raw_input + 1;
    ... /* Initialization not shown */
    task_scheduler_init init(nThreads);

    Average avg(padded_input, output);
    parallel_for( blocked_range<int>( 0, N, 1000 ), avg );
    return 0;
}

Figure 3.8: TBB Average

vector. A grain size of 1000 is advised in this case. The second argument is an object of the class Average which encapsulates the operation to be executed by the parallel_for. This class is defined in lines 1-9. The overloaded () in this class contains the code that computes the average for each element in the subrange on which it executes. The bounds of the indexes for each subrange are directly extracted from the range object using the begin() and end() methods.

The Ravenna implementation of this algorithm is shown in Figure 3.9. The data structures are created in lines 14-17. Line 15 defines an object that describes the overlapping of tiles in input. The first two arguments of the constructor specify that shadow regions have size one in both the positive and negative direction. This constructor allows a third optional argument to specify whether the boundary region built around the array is filled with zeros, which is default behavior when nothing is specified, or it is periodic, i.e., it replicates the values of the array on the opposite side. In line 16 this overlapping specification is used to
create a tiled array with $N$ values distributed across `nTiles`. Line 17 allocates the tiled array where the result will be stored, which has the same topology as the one used as input but with no overlapped regions.

The `map` operator is invoked in line 23. Its first argument is the operation to perform on each tile of the objects. This operation, `Average`, is defined as a `struct` in lines 3-8.

The `for` loop of line 5 iterates on the indexes of the elements in each tile.

In this example, there is little difference between the TBB and Ravenna implementations. The main difference is that Ravenna implements the computation with tiles that the programmer defines. TBB implements this example using a `blocked_range` object that represents the indices of the array. This range is partitioned dynamically by the runtime. However, this example does not especially benefit from dynamic scheduling since static scheduling can evenly divide the work among the processors. Tiling, in this case, makes this fact explicit to the programmer. It also allows this program to be portable to distributed-memory systems, unlike the TBB version.
3.5 Substring Finder

In this code, for each position in a string, the program finds the length and location of the largest matching substring elsewhere in the string. For instance, take the string `flowersflows`. Starting the scan at the first character at position 0, the largest match is `flow` at position 7 with a length of 4 characters. The position and length of those matches are stored for each position of the string.

The parallelization strategy consists of searching the largest matching string for each position of the scanned string in parallel. The TBB version uses a `parallel_for`, while the Ravenna version uses a `map`.

The codes, shown in Figures 3.10 and 3.11 are very similar. The operation performed in parallel is the same in both cases, the only difference is the indexing of the data structures, as it happened in previous codes. In the Ravenna version, the `max` and `pos` arrays, where the result will be stored, are divided in tiles, and the `map` operation is applied separately.
typedef TiledArray<int,1> TA;

struct SubStringFinderOp {

    void operator() ( TA max, TA pos ) {
        ...
        int pos=0;
        for (size_t i = init; i != end; ++i) {
            int max_size = 0, max_pos = 0;
            for (size_t j = 0; j < str.size(); j++) {
                if (j != i) {
                    int limit = str.size() - (i > j ? i : j);
                    for (int k = 0; k < limit; ++k) {
                        if (str[i + k] != str[j + k]) break;
                        if (k > max_size) {
                            max_size = k; max_pos = j;
                        }
                    }
                }
            }
            max[pos] = max_size;
            pos[pos] = max_pos;
        }
        max.map(SubStringFinderOp(), pos);
        ...
    }
}

Figure 3.11: Ravenna version

on each tile, so the indexing will be relative to the start position of the current tile. The
range of the search in this computation is the range of subscripts that index the characters
of the string. The TBB implementation uses a range object to implement this. The Ravenna
implementation reads the location of the tile in the flattened array from the arrays that store
the output and uses this information to determine the starting positions in the string that
a particular tile searches. The Ravenna implementation is thus slightly more complex than
the TBB implementation, but it is also more portable on account of tiling.

3.6 Evaluation

Having implemented these simple examples using both TBB and Ravenna, the Ravenna codes
were compared against the TBB versions for both performance and productivity. Table 3.1
lists the lines of code for both the Ravenna and TBB versions of each code. For Parallel
Parallel Merge, the Ravenna code is just smaller than the TBB code. Both codes work in roughly the same fashion, but the Ravenna code is more clear what is happening since the programmer is explicitly creating nested parallelism with dynamic tiling rather than relying on the library to do it as is the case for TBB. However, the Ravenna version of the Game of Life sees significant reduction in the amount of code. This is on account of Overlapped Tiling since it greatly simplifies accesses the neighbors of a cell. The Ravenna code uses normal indexing to find the neighbors. The TBB code has a large helper routine that must handle all the possible border cases. Both Average and Substring Finder have similarly sized implementations in both Ravenna and TBB. However, the Ravenna implementations are based on higher level data parallel operators that can be portable across platforms.

Performance results for the Ravenna and TBB versions of the examples are found in Table 3.2 and Table 3.3. Table 3.2 shows performance for an 8-core Intel Xeon system with 2 sockets and 4 cores per socket. Table 3.3 shows performance for a 16-core Intel Itanium 2 system with 8 sockets and 2 cores per socket. The performance of the Ravenna and TBB codes is fairly comparable on both systems. TBB tends to outperform the Ravenna code on Parallel Merge. This is largely due to the work-stealing scheduler of TBB. This scheduler is able to dynamically split ranges in order to balance the load. The Ravenna code does not exhibit this behavior by default, but it can be approximated by overdecomposing the tiled objects to have more tiles than processors. This allows the TBB scheduler upon which the Ravenna library depends to perform work-stealing when appropriate. The Ravenna

<table>
<thead>
<tr>
<th>Code</th>
<th>Lines (Ravenna)</th>
<th>Lines (TBB)</th>
<th>Savings</th>
</tr>
</thead>
<tbody>
<tr>
<td>Parallel Merge</td>
<td>70</td>
<td>74</td>
<td>+5.4%</td>
</tr>
<tr>
<td>Game of Life</td>
<td>97</td>
<td>309</td>
<td>+69.0%</td>
</tr>
<tr>
<td>Average</td>
<td>23</td>
<td>23</td>
<td>0.0%</td>
</tr>
<tr>
<td>Substring Finder</td>
<td>49</td>
<td>49</td>
<td>0.0%</td>
</tr>
</tbody>
</table>

Table 3.1: Number of Lines of Code
<table>
<thead>
<tr>
<th>Code</th>
<th>Execution Time (ms)</th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Merge Ravenna</td>
<td>68.6</td>
<td>36.4</td>
<td>34.0</td>
<td>22.2</td>
<td>21.3</td>
</tr>
<tr>
<td>Merge TBB</td>
<td>73.0</td>
<td>36.1</td>
<td>26.0</td>
<td>20.7</td>
<td>19.5</td>
</tr>
<tr>
<td>Game of Life Ravenna</td>
<td>4957.0</td>
<td>2465.0</td>
<td>2577.4</td>
<td>1745.7</td>
<td>1088.1</td>
</tr>
<tr>
<td>Game of Life TBB</td>
<td>4473.9</td>
<td>2745.5</td>
<td>2130.2</td>
<td>1813.3</td>
<td>1381.3</td>
</tr>
<tr>
<td>Average Ravenna</td>
<td>5.2</td>
<td>2.5</td>
<td>2.1</td>
<td>2.2</td>
<td>1.5</td>
</tr>
<tr>
<td>Average TBB</td>
<td>3.1</td>
<td>2.1</td>
<td>2.2</td>
<td>2.4</td>
<td>2.5</td>
</tr>
<tr>
<td>Substring Ravenna</td>
<td>5885.9</td>
<td>2992.0</td>
<td>2003.7</td>
<td>1541.6</td>
<td>768.9</td>
</tr>
<tr>
<td>Substring TBB</td>
<td>6380.2</td>
<td>3203.8</td>
<td>2132.1</td>
<td>1610</td>
<td>820.3</td>
</tr>
</tbody>
</table>

Table 3.2: Performance on Intel Xeon using 1 to 8 cores and 1 tile per core

code for Merge also modifies the tiling structure of the tiled objects as it recurses, incurring overhead. Conversely, the Ravenna version of the Game of Life tends to outperform the TBB implementation for two reasons. First, the information stored by the tiled object about its tiling simplifies indexing the neighboring cells. Secondly, the TBB scheduler does not take locality into account. Tiling this stencil computation enhances locality for the Ravenna version. This is further enhances by overdecomposing the tiled object to tune the tile size to fit the cache of the machine. Average is a simpler stencil computation than the Game of Life. For this example, the amount of work that must be performed per element is the same, making static scheduling efficient. Consequently, the Ravenna implementation outperforms the TBB implementation, which splits the range into more tasks than necessary. This limits spatial locality since tasks contain fewer contiguous elements and the TBB scheduler prefers to spread ranges out when assigning tasks to threads in order to avoid cache contention. A similar situation occurs for Substring Finder. The Ravenna and TBB codes have similar performance, with Ravenna edging out TBB.

As was mentioned above, overdecomposition can often help performance for Ravenna programs. Speedups for the Ravenna codes are shown in Figures 3.12 and 3.13 for Parallel Merge and Figures 3.14 and 3.15 for the Game of Life. The black bar represents the
<table>
<thead>
<tr>
<th>Code</th>
<th>Execution Time (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
</tr>
</tbody>
</table>
| Merge      | 199.2  
Ravenna | 202.4  
TBB    | 19396.7  
Ravenna | 16483.5  
TBB | 25.1  
Ravenna | 23.8  
TBB    | 9510.4  
Ravenna | 10689.4  
TBB |
| Merge      | 128.3  
Ravenna | 116.7  
TBB    | 9486.7  
Ravenna | 9623.1  
TBB | 11.2  
Ravenna | 13.1  
TBB    | 4895.6  
Ravenna | 5361.9  
TBB |
| Game of Life | 79.6  
Ravenna | 66.9  
TBB    | 6953.0  
Ravenna | 6147.6  
TBB | 7.2  
Ravenna | 11.4  
TBB    | 2455.3  
Ravenna | 2692.9  
TBB |
| Game of Life | 52.1  
Ravenna | 44.3  
TBB    | 3478.9  
Ravenna | 4386  
TBB | 4.5  
Ravenna | 11.2  
TBB    | 1256.8  
Ravenna | 1366.2  
TBB |
| Average    | 44.8  
Ravenna | 38.1  
TBB    | 2109.4  
Ravenna | 3654.7  
TBB | 3.8  
Ravenna | 11.7  
TBB    | 791.5  
Ravenna | 924.4  
TBB |
| Average    | 44.5  
Ravenna | 35.0  
TBB    | 1690.8  
Ravenna | 3409.9  
TBB | 3.5  
Ravenna | 11.3  
TBB    | 689.9  
Ravenna | 717.0  
TBB |

Table 3.3: Performance on Intel Itanium 2 using 1 to 16 cores and 1 tile per core

speedup obtained when using one tile per core. The white bar represents the best speedup obtained by overdecomposing the tiled objects into more tiles than cores. For Parallel Merge, overdecomposition helps alleviate load imbalance that result from the un-even partitioning of the second input. This brings the performance of the Ravenna implementation of Parallel Merge closer to that of the TBB implementation. For the Game of Life, overdecomposition enhances locality by using tiles that fit into cache. This could also be accomplished with a hierarchical tiling.

### 3.7 Conclusion

This evaluation compared the implementations of several short computations written both in Ravenna and in TBB, a library for shared-memory programming upon which Ravenna is built. The Ravenna implementations were compared against the TBB implementations
Figure 3.12: Parallel Merge Performance, 8 core Xeon

Figure 3.13: Parallel Merge Performance, 16 core Itanium 2
Figure 3.14: Game of Life Performance, 8 core Xeon

Figure 3.15: Game of Life Performance, 16 core Itanium 2
for both productivity and performance. The Ravenna implementations are shorter or on par with those written in TBB. Particular benefit existed for computations with complex indexing where the library simplified programmer effort. Performance of the Ravenna codes was on par with that of the TBB codes. However, the programmer had more control over the performance of the Ravenna codes through tiling. Programmers could start with partitioning the codes into as many tiles as processors. In the case of Average or Substring Finder, this was a good strategy. Overdecomposition improves the performance of Parallel Merge by creating a more balanced execution. It also improved the performance of the Game of Life both through load balance and choosing a tile size that fit into cache.

The Ravenna library seems a more natural way to express data parallel computations, which arise frequently in real programs, than the inherently task-based TBB library. TBB offers more flexibility at the cost of greater complexity and can be used to solve computations where Ravenna may not be suitable. However, the higher level data parallel operators of Ravenna have a higher level of abstraction that is portable to other platforms. TBB codes can only run in shared-memory environments.
Chapter 4

Tiled SPIKE

Linear solvers are an important class of numerical computation. Many important problems are sparse. It is well known that the desired data structure to represent sparse systems influences the performance of solvers for this type of linear system. These computations do not use dense arrays but rather only store the elements of a matrix that may be non-zero. Such storage mechanisms reduce not only the memory footprint but can also reduce the amount of computation needed by only performing computation on relevant elements. The SPIKE family of algorithms [38] is one such parallel solver for banded linear systems of equations.

Consider a linear system of the form $Ax = f$, where $A$ is a banded matrix of order $n$ with bandwidth much less than $n$. One can partition the system into $p$ diagonal blocks. Figure 4.1 shows the partitioned system for $p = 4$ where each $A_i$ is a banded matrix of order $n/p$. The matrices $B_i$ and $C_i$ are of order $m$ where the bandwidth of the original matrix $A$ is $2m + 1$. Only the $A$, $B$, and $C$ blocks need to be stored for this type of sparse matrix.

Let the block diagonal matrix $D = \text{diag}(A_1, ..., A_4)$. If one were to left-multiply each side of the above by $D^{-1}$, one would obtain a system of the form shown in Figure 4.2.

However, instead of computing $D^{-1}$, one can compute, as seen below, the blocks of $V$
and \( W \), or, the \textit{spikes} by solving a system of equations. The spikes have the same width, \( m \), as the \( B \) and \( C \) tiles in the original system.

\[
A_i \begin{bmatrix} V_i, W_i \end{bmatrix} = \begin{bmatrix} 0 & C_i \\ 0 & 0 \\ B_i & 0 \end{bmatrix} \tag{4.1}
\]

Solving the original system \( Ax = f \) now consists of three steps.

1. Compute the spikes by solving (4.1)
2. Solve \( Dg = f \)
3. Solve \( Sx = g \)
The solution of the system \( Dg = f \) yields the modified RHS for the system in the third step. Notice that each blocks of \( D \) are independent and thus can be computed in parallel. Solving the third system can be further reduced by solving the system \( \hat{S}\hat{x} = \hat{g} \), which consists of the \( m \) rows of \( S \) directly above and below the boundaries between the \( I \) tiles. The spikes, \( f \), and \( g \) can also be partitioned as follows.

\[
\begin{align*}
V_j &= \begin{bmatrix} V_j^{(t)} \\ V_j' \\ V_j^{(b)} \end{bmatrix} \quad W_j &= \begin{bmatrix} W_j^{(t)} \\ W_j' \\ W_j^{(b)} \end{bmatrix} \quad x_j &= \begin{bmatrix} x_j^{(t)} \\ x_j' \\ x_j^{(b)} \end{bmatrix} \quad g_j &= \begin{bmatrix} g_j^{(t)} \\ g_j' \\ g_j^{(b)} \end{bmatrix}
\end{align*}
\]

(4.2)

The reduced system thus takes the following form:

\[
\begin{bmatrix}
I_m & 0 & V_1^{(t)} \\
0 & I_m & V_1^{(b)} & 0 \\
0 & W_2^{(t)} & I_m & 0 & V_2^{(t)} \\
W_2^{(b)} & 0 & I_m & V_2^{(b)} & 0 \\
0 & W_{p-1}^{(t)} & I_m & 0 & V_{p-1}^{(t)} \\
W_{p-1}^{(b)} & 0 & I_m & V_{p-1}^{(b)} & 0 \\
0 & W_p^{(t)} & I_m & 0 & V_p^{(t)} \\
W_p^{(b)} & 0 & I_m & 0 & V_p^{(b)}
\end{bmatrix}
\begin{bmatrix}
x_1^{(t)} \\
x_1^{(b)} \\
x_2^{(t)} \\
x_2^{(b)} \\
x_{p-1}^{(t)} \\
x_{p-1}^{(b)} \\
x_p^{(t)} \\
x_p^{(b)}
\end{bmatrix} =
\begin{bmatrix}
g_1^{(t)} \\
g_1^{(b)} \\
g_2^{(t)} \\
g_2^{(b)} \\
g_{p-1}^{(t)} \\
g_{p-1}^{(b)} \\
g_p^{(t)} \\
g_p^{(b)}
\end{bmatrix}
\]

Finally, once the solution to the reduced system has been directly computed sequentially, one will have the values of the \( x^{(b)} \)s and \( x^{(t)} \)s. The rest of \( x \) can then be computed as follows:

\[
\begin{align*}
x_1' &= g_1' - V_1'x_2^{(t)}, \\
x_j' &= g_j' - V_j'x_{j+1}^{(t)} - W_j'x_{j-1}^{(b)}, \quad j = 2, \ldots, p - 1, \\
x_p' &= g_p' - W_p'x_{p-1}^{(b)}.
\end{align*}
\]

(4.3)

Thus the SPIKE algorithm can be broken down into the following steps:
1. Factorize the diagonal blocks of $A$. This is much more efficient than computing $A^{-1}$ directly.

2. Compute the spikes using the factorization obtained in the previous step and compute the right hand side.

3. Form and solve the reduced system.

4. Compute the rest of $x$.

### 4.1 SPIKE Variants

The original SPIKE algorithm explained above has many variants. Some variants target systems of equations with certain properties in order to reduce the amount of computation performed. Some also increase the amount of parallelism available during different stages of the algorithm. In [5], the focus is placed on two variants that use a truncated scheme to solve the reduced system. The truncated scheme is used for systems that are diagonally dominant. In diagonally dominant systems, the values in the spikes far from the diagonal are likely to be very close to zero and therefore contribute little to the solution. Consequently, the truncated scheme treats these values as zero and only computes the $m \times m$ portion of the spikes close to the diagonal, specifically, $V^{(b)}$ and $W^{(t)}$. This is accomplished by either using the LU or UL factorization computed for the blocks of the diagonal.

The two variants presented are called $\text{TU}$ and $\text{TA}$, and both implement the truncated scheme. LU factorization of $A_i$ is used to solve the bottom tips, $V^{(b)}_i$, of the spikes and the UL factorization of $A_i$ is used to solve for the top tips, $W^{(t)}_i$, of the spikes. The difference between $\text{TU}$ and $\text{TA}$ lays in the decomposition of the work. In the $\text{TU}$ scheme, the original matrix is partitioned into as many blocks as there are processors. Figure 4.3 shows this partitioning for the case with 4 processors. In this figure $\hat{B}$ and $\hat{C}$ are $B$ and $C$ extended.
with zeros as in equation 4.1.

\[
A = \begin{bmatrix}
A_1 & & & \\
& A_2 & & \\
& & A_3 & & \\
& & & A_4 \\
\end{bmatrix}
\]

\[
f = \begin{bmatrix}
P_1 & LU & A_1^{-1}B_1 & \hat{A}_1^{-1}f_1 \\
P_2 & LU & A_2^{-1}B_2, A_3^{-1}\hat{C}_2 & \hat{A}_2^{-1}f_1 \\
P_3 & LU & A_3^{-1}\hat{C}_3, A_4^{-1}B_3 & \hat{A}_3^{-1}f_1 \\
P_4 & UL & A_4^{-1}\hat{C}_4 & \hat{A}_4^{-1}f_1 \\
\end{bmatrix}
\]

Figure 4.3: Spike TU Partitioning

The TA scheme arises from the fact that the factorization step dominates execution time. TA is similar to TU with the exception that it partitions the matrix in a different fashion. Instead of each processor computing both LU and UL for a block since some blocks must compute two spikes, each processor now computes either LU or UL for a block but not both in order to compute a single spike. Note that this scheme partitions the matrix into fewer blocks than the TU scheme does, but results in better load balance for the computation of the spikes. Figure 4.4 shows this partitioning for 4 processors using \(\hat{B}\) and \(\hat{C}\) as above.

\[
A = \begin{bmatrix}
A_1 & & & \\
& A_2 & & \\
& & A_3 & & \\
& & & A_4 \\
\end{bmatrix}
\]

\[
f = \begin{bmatrix}
P_1 & LU & A_1^{-1}B_1 & \hat{A}_1^{-1}f_1 \\
P_2 & LU & A_2^{-1}B_2 & \hat{A}_2^{-1}f_1 \\
P_3 & LU & A_3^{-1}\hat{C}_3 & \hat{A}_3^{-1}f_1 \\
P_4 & UL & A_4^{-1}\hat{C}_4 & \hat{A}_4^{-1}f_1 \\
\end{bmatrix}
\]

Figure 4.4: Spike TA Partitioning

Both versions of the algorithm compute the \(W^{(t)}\), \(V^{(b)}\), and \(g\) tips that are needed for the truncated reduced system, shown in Figure 4.5. This system will be block diagonal and has one less block than the original system. Thus when solving with \(p\) processors TU will have \(p - 1\) blocks in the reduced system and TA will have \((p + 2)/2 - 1\) blocks in the reduced system. Thus the TU version will have more parallelism than the TA version in this stage of
the computation. Unlike the original SPIKE algorithm, the reduced system for truncated schemes can be solved in parallel via a direct scheme where each block has the following form:

\[
\begin{bmatrix}
I_m & V_j^{(b)} \\
W_j^{(t)} & I_m
\end{bmatrix}
\begin{bmatrix}
x_j^{(b)} \\
x_j^{(t)}
\end{bmatrix}
= \begin{bmatrix}
g_j^{(b)} \\
g_j^{(t)}
\end{bmatrix}
\] (4.4)

Finally the solution to the original system is recovered by solving:

\[
A_j x_j = f_j - \begin{bmatrix}
0 & \cdots & 0 \\
0 & \cdots & 0 \\
B_j & 0 & \cdots & 0
\end{bmatrix}
\begin{bmatrix}
x_j^{(t)} \\
x_j^{(t)} \\
C_j
\end{bmatrix}
\] (4.5)
This can be done in parallel with either the LU or UL factorization of $A_j$. Here again the TU version has more parallelism than the TA version.

4.2 Tiled Spike

The initial implementations of the SPIKE family of algorithms are available as the Intel Adaptive Spike-based Solver[44], or SpikePACK. It targets distributed-memory using MPI and is written in Fortran. However, the block structure of the SPIKE algorithms maps very well to tiled arrays in the Ravenna and HTA libraries. Several SPIKE algorithms were thus implemented using tiled arrays in these libraries for two reasons. First, writing SPIKE using the libraries would allow programmers to write one portable code that can be run on both shared-memory and distributed-memory target platforms. Second, the libraries’ notation allows for an elegant, clean implementation of the algorithms. A tiled SPIKE would more closely resemble the high-level mathematical expression of the algorithms than Fortran+MPI. Communication takes the form of simple array assignments between tiles.

The TU and TA variants of SPIKE were implemented with tiled arrays in the HTA and Ravenna libraries. The tiles of the arrays map to the blocks of the banded linear system. The bands of the system are stored inside the tiles using the banded storage format used by LAPACK. Since the code makes extensive use of LAPACK routines DGBTRF and DGBTRS to factorize and solve banded systems, the libraries were modified to support column-major data layout due to the Fortran origins of these routines. Both the HTA and Ravenna libraries are written in C++ and originally only supported row-major layout.

The blocks of the bands, spikes, and reduced system are all represented as tiled arrays. The storage for the diagonal blocks of the system is overwritten to store the LU or UL factorizations of each block. The storage for the $B$ and $C$ blocks is likewise overwritten to contain the tips of the spikes. The number of partitions used by the algorithm for a
given number of processors directly determines the tiling of the objects. The algorithm is represented as a sequence of data parallel operations. The semantics state that each data parallel operation is followed by an implicit barrier. This allows the programmer to reason about the algorithm sequentially as the parallelism is thus encapsulated inside of the data parallel operators. The data parallel operations often are represented as map operations. This is the mechanism through which one applies LAPACK kernels in parallel across all the tiles of an array. The implementations also heavily use the array operations provided by the library. When coupled with HTA’s and Ravenna’s first class tile objects, array operations enable programmers to write simple, compact statements that can communicate a range of data from one set of tiles to another. This contrasts with a Fortran+MPI approach where it is difficult to separate the algorithm from the implementation.

Porting the programs from one platform to another is accomplished by simply changing the header file for the library. In order to target MPI, one includes htalib_mpi.h. In order to target TBB, one includes ravenna_shmem.h.

4.2.1 TU

Figure 4.6 presents the core of the implementation. A simplified notation is used to represent Triplet objects. Recall that TU partitions the matrix into as many blocks as processors. The tiled arrays LUA and ULA initially are identical and contain the diagonal blocks of the system. The LU and UL factorizations of these blocks are performed in-place and in parallel by the map operators used in lines 3-4. The off-diagonal blocks, B and C, that will contain the spike tips are stored in the tiled array BC. Each tile of this array contains space for both the “left” (W(t)) and “right” (V(b)) spikes associated with each block. The spike tips are computed in line 7 using the LU and UL factorizations computed previously. The whole right-hand side (RHS) for the system is then updated in line 10 using the LU factorization of the diagonal blocks.
The reduced system, shown in Figure 4.5, can be formed now that the spikes and updated RHS have been computed. Lines 13-16 make use of HTA and Ravenna array assignments to construct the reduced system by copying the spike tips into the appropriate sections of each block of the reduced system. The tiled arrays \texttt{REDUCED} and \texttt{BC} are indexed using ( ) and [ ] operators and triplet notation. The first ( ) selects every tile of the tiled array \texttt{REDUCED}. For the tiled array \texttt{BC}, different ranges of tiles are selected for each statement. The [ ] operator is used to index a range of elements inside of a tile. The RHS of the reduced system is formed similarly in lines 19-20. Note that the array assignments used to form the reduced system imply communication. Once the reduced system has been formed, it may be solved in parallel as its blocks are independent. This is accomplished by calls to the \texttt{map} operator on lines 23 and 25.

Having solved the reduced system, the RHS of the original system is updated in lines 28-33. This is accomplished by array assignments and another call to \texttt{map} that performs matrix-vector multiplications in parallel. Once the RHS has been updated with the values computed from the reduced system, the rest of the solution is obtained in line 36.

The implementation of the \texttt{TU} scheme slightly deviates from the SpikePACK implantation of the algorithm in two ways. First, the first and last partitions need only compute LU or UL, respectively. The inner partitions must compute both LU and UL in order to compute the tips of the left and right spikes. The first and last partitions only have either a right or a left spike and do not need to compute both. However, the implementation chose to have the first and last partitions compute a fake spike in order to avoid special cases when computing the spikes. It computes both LU and UL for all partitions where as the SpikePACK only computes the LU for the first and the UL for the last as needed by the algorithm. Secondly the SpikePACK implementation uses a nonuniform distribution with larger partitions for the first and last partitions to balance the load since they are only computing one factorization. Since two factorizations are computed for every partition, the
// factorize blocks of A
LUA.map(factorize_lua());
ULA.map(factorize ula());

// calculate the spike tips W(t) and V(b) from Bs and Cs
BC.map(solve_bc(), LUA, ULA);

// update right hand side
g.map(solve lua(), LUA);

// form the reduced system
REDUCED()[0:m-1,m:2*m-1] =
BC(0:num_blocks-2)[0:m-1,0:m-1];
REDUCED()[m:2*m-1,0:m-1] =
BC(1:num_blocks-1)[0:m-1,m:2*m-1];

// form the reduced system RHS
greduced()[0:m-1] = g(0:num_blocks-2)[blocksize-m:blocksize-1];
greduced()[m:2*m-1] = g(1:num_blocks-1)[0:m-1];

// factorize the reduced system
REDUCED.map(factorize());

// solve the reduced system
greduced.map(solve(), REDUCED);

// Update RHS with the values from the spikes as r = r - Bz - Cz
fv = r(0:num_blocks-2); fr_half = greduced()[0:m-1];
B.map(dgemv(), fv, fr_half);
fr(0:num_blocks-2) = fv;
fw = r(1:num_blocks-1); fr_half = greduced()[m:2*m-1];
C.map(dgemv(), fw, fr_half);
fr(1:num_blocks-1) = fw;

// Solve the updated system
r.map(solve lua(), LUA);

...
spikes. TA has each processor compute only one spike. Consequently TA partitions the matrix into fewer blocks for a given number of processors than TU as shown in Figure 4.4. Whereas TU stored the diagonal blocks in the tiled arrays LUA and ULA, TA stores the appropriate blocks in the tiled array DIAGS. Note that DIAGS can contain two copies of the same block of $A$ since the same block is needed to compute two different spikes for the inner blocks. An additional tiled array, DIAG_MAP, is used to set flags that indicate whether each tile needs to perform the LU or the UL factorization for its block. This can be seen in line 3 for the factorization and line 7 for the computation of the spike tips. The tiled array TOSOLVERHS is used to refer to part of DIAGS as that array can contain multiple factorizations for each block. TOSOLVERHS, seen on line 4, contains only one factorization for each block of the matrix and is used to update the right hand side on lines 9 and 35. This is also matched with a map that indicates the type of factorization contained in the tile. Forming and solving the reduced system proceeds almost identically to the implementation of TU. Note that there is less parallelism available in this phase of TA than in TU due to partitioning the system into fewer blocks.

4.3 Evaluation

In order to evaluate the performance of the tiled implementations of the two spike variants, several experiments were conducted that compare the performance of the tiled implementations to both the SPIKE implementations in the Intel® Adaptive Spike-Based Solver version 1.0 and the sequential banded solvers found in the Intel® Math Kernel Library version 10.2 Update 5. The numbers reported are speedups over the execution time of the sequential MKL routines. All code was compiled with the Intel® compilers icc and ifort version 11.1 Update 6, and all MPI programs were run using mpich2. The Ravenna library runs on TBB version 2.2 Update 3.
// factorize the A blocks
DIAGS.map(factorize_diag(), DIAG_MAP);
TOSOLVERHS = DIAGS(0:num_blocks-1);

// compute the spike tips from Bs and Cs
BC.map(solve_bc(), DIAG_MAP, DIAGS);

// generate modified right hand side
g.map(solve rhs(), TOSOLVERHS_MAP, TOSOLVERHS);

// form the reduced system
REDUCED() [0:m−1,m:2∗m−1] =
BC(0:num_blocks−2)[0:m−1,0:m−1];
REDUCED()[m:2∗m−1,0:m−1] =
BC(num_blocks−1:2∗num_blocks−3)[0:m−1,0:m−1];

// form the reduced system right hand side
greduced()[0:m−1] = g(0:num_blocks−2)[blocks−m:blocksize−1];
greduced()[m:2∗m−1] = g(1:num_blocks−1)[0:m−1];

// factorize the reduced system
REDUCED.map(factorize());

// solve the reduced system
greduced.map(solve(), REDUCED);

// Update RHS with the values from the spikes as r = r − Bz − Cz
fv = r(0:num_blocks−2); fr_half = greduced()[0:m−1];
B.map(dgemv(), fv, fr_half);
r(0:num_blocks−2) = fv;
fw = r(1:num_blocks−1); fr_half = greduced()[m:2∗m−1];
C.map(dgemv(), fw, fr_half);
r(1:num_blocks−1) = fw;

// Solve the updated system using the LU and UL as needed
r.map(solve rhs(), TOSOLVERHS_MAP, TOSOLVERHS);

Figure 4.7: Tiled SPIKE TA
In all cases several different systems of equations were tested and the results were similar. One case is presented for each algorithm. Tests were run on two platforms. The first is a four socket 32-core system using Intel Xeon L7555 processors running at 1.86 GHz. The system has 64 gigabytes of memory installed. The second system is a cluster built out of Intel Xeon E5504 processors running at 2.0 GHz. The cluster contains eight compute nodes, each containing eight cores. All tests were run eight times and the minimum execution time is reported.

4.3.1 TU

Figures 4.8 and 4.9 present results for a matrix of order 1048576 with 256 superdiagonals and 256 subdiagonals. This size was chosen in order to partition the matrix into blocks of equal size for the number of processors used. The execution times of the sequential MKL solver routines are used to compute speedups for TU running on Ravenna, HTAs for distributed-memory, and the Intel SpikePACK.

Tiled SPIKE’s performance advantage comes from implementation differences. SpikePACK uses larger blocks for the first and last partitions to attempt to minimize any load imbalance when computing factorizations and the spikes. However, this creates imbalance when retrieving the solution to the whole system after the reduced system has been solved since the retrieval for the outer blocks will require more time than the retrieval for inner blocks. As the number of processors increases, the retrieval becomes a larger portion of the total execution, and this imbalance is magnified. The tiled versions use evenly sized partitions. Algorithmically, this is imbalanced as the first and last partition only need to compute one spike, but the tiled implementations ignore this in order to simplify programming. SpikePACK also does extra data copying in order to enable using SPIKE as a preconditioner.

It is also important to note that the performance of the tiled codes on both shared-memory and message-passing is almost identical when running on the L7555 machine. While
at first this result was surprising, it is indeed to be expected. The amount of computation is large, so the overheads of each runtime system are minimal. The ideal tiling structure may differ from one platform to the next, but a given tiling ought to perform similarly on the same system regardless of the backend. One can also note from the figures that the tiled implementation of \textit{TU} obtains between performance on the cluster than the large multicore when using more than eight processors. SPIKE is a very memory intensive algorithm, and consequently, increasing the amount of parallelism used in shared memory can require more memory bandwidth than the system can provide. This is shown by the greater speedups obtained on the cluster when using larger numbers of processors since each core is able to use a greater percentage of available memory bandwidth on a node.

4.3.2 TA

Figures 4.10 and 4.11 present results for a matrix of order 1093950 with 256 superdiagonals and 256 subdiagonals. This size was again chosen to partition the matrix into blocks of uniform size. Recall that the \textit{TA} scheme partitions the matrix into fewer blocks than the \textit{TU} scheme for a given number of processors. \textit{TU} assigns one block of the matrix per processor while \textit{TA} assigns one spike calculation per processor. The results of these tests are presented in Figure 4.10 which again shows speedup over sequential MKL for the three implementations. Each version tends to outperform \textit{TU} and scales reasonably with increasing processors. However, SpikePACK begins to outperform the tiled implementations after 16 processors.

The performance difference seen in this case is due to the differences in the communication patterns between the tiled versions and the SpikePACK version. In the SpikePACK version of the algorithm, care is taken so that only one of the tips needs to be communicated to build the reduced system. This produces an irregular distribution of data. In cases where the number of partitions is small, distribution does not have a large impact but as the number of partitions grow the impact becomes more significant.
We believe that this behavior could be implemented in the tiled versions of TA in two ways. First, the version of the library built on top of MPI provides support for user-defined distributions. These distributions could map the tiles of the spikes, RHS, and reduced system in such a way that minimizes communication between processors. Ravenna currently has no analog. This limitation is inherent in many libraries for shared-memory programming as they do not expose mechanisms to bind a thread to a particular core. The second way through which we could mimic SpikePACK’s performance is through changing our implementation of the algorithm. By storing the blocks of the reduced system in a different order, we could more closely align the respective tiles of the spikes and RHS with the appropriate tiles of the reduced system. However, this complicates the implementation as the programmer becomes responsible for maintaining the mapping of the blocks of the reduced system to their locations in the array’s tiling structure. We chose to initially focus on implementing a simple, elegant solution that closely maps to the algorithm.

4.4 Conclusion

This chapter presented Tiled SPIKE, an implementation in Ravenna of two variants from the SPIKE family of algorithms for solving banded systems of linear equations. The higher level data parallel operators in Ravenna facilitate portable parallel programming and increase productivity. Tiles facilitate the mapping of block algorithms to code and result in programs that can run without modification on both shared-memory and distributed-memory machines.

Experiments show that the performance of the same Tiled SPIKE code running on both shared-memory and distributed-memory achieve similar performance and are competitive to the reference Intel Fortran+MPI SPIKE implementations. In addition, Tiled SPIKE shows that the features provided by Ravenna result in programs that are both clean and compact.
Figure 4.8: TU Speedups
Figure 4.9: Cluster TU Speedups

Figure 4.10: TA Speedups
and more closely resemble the algorithmic description of the problem.
Figure 4.11: Cluster TA Speedups
Chapter 5

Computations on Non-Array Datatypes

In this chapter, three examples of computations implemented using Ravenna are presented. (1) *Barnes-Hut* implements a n-body simulation in three dimensions. (2) *Clustering* implements a simple k-means algorithm. (3) *Discrete Optimization* performs breadth-first and A* graph searches in parallel. A simple problem, the 15-puzzle sliding tile problem, is used to measure parallel performance [17] for A*. Experiments were performed on three systems. The first is a Xeon E7450 system at 2.4 GHz with four sockets and 24 cores. Each processor has 32KB of private L1 cache, every pair share 3MB of L2, and each socket of 6 processors share 12 MB L3. The second system is a Xeon E5405 system at 2.0 GHz with two sockets and 8 cores. Each processor has 32KB of private L1 cache and each socket of 4 processors has 12MB of shared L2. The third system is a Xeon L7555 system at 1.86 GHz with four sockets and 32 cores. Each processor has 32KB of private L1 cache, 256KB of private L2 cache, and 24MB of shared L3. Barnes-Hut and Search use one level of tiling, while Clustering uses two levels in order to block for L1 cache.
5.1 Barnes-Hut

Barnes-Hut is an algorithm to carry out an n-body simulation. In our example, the algorithm computes one force between all pairs of particles distributed in space. Barnes-Hut is an improvement over the naive $O(n^2)$ algorithm. Improvement comes from using spatial decomposition to reduce the number of force computations performed. This spatial decomposition recursively divides a cube containing one or more bodies into eight evenly sized cubes until no cube contains more than a single particle. This decomposition is represented with a spatial data structure called an Octree. Bodies are contained in the leaves of the Octree. Intermediate nodes represent the centers of mass of their children in the tree and their locations. When a body is computing the forces acting upon itself, it may compute the force using these centers of mass if they are sufficiently distant from the body, reducing the number of computations performed. Figure 5.1 illustrates the spatial decomposition performed by the Octree.

Each iteration performs several tasks. First, the octree is constructed to represent the spatial decomposition. Next, the centers of mass are computed for the tree. The forces are computed for each body in space, and finally the bodies are updated to their new positions. The analysis focuses its explanation on the force calculation step of the algorithm as it dominates execution time. As the amount of parallelism increases, this dominance decreases, but even using 16 processors it still accounts for 90 percent of execution time.

The implementation makes use of three features of Ravenna: a tiled set, functional tiling, and \texttt{map}. An excerpt of the code is shown in Figure 5.2. The bodies are stored in a tiled set. \texttt{build\_tree} constructs the Octree for the set of bodies. The centers of mass are then computed via a traversal of the Octree. \texttt{tile} then reorganizes the bodies so that each body is found in the tile to which the tiling function \texttt{strategy} maps it. The force computation is then performed in parallel across all the tiles of the set. This is accomplished by using
TiledSet bodies(num_tiles);
for each timestep:
    root = build_tree(bodies);
    compute_centers_of_mass(root);
    bodies.tile(strategy());
    bodies.map(ForceOp(root));
    bodies.map(AdvanceOp());
    ...

Set bodies;
for each timestep:
    root = build_tree(bodies);
    compute_centers_of_mass(root);
    for each body in bodies:
        compute_force(body, root);
        for each body in bodies:
            advance(body);
    ...

Figure 5.1: Spatial Decomposition (a) and Octree (b)

Figure 5.2: Barnes-Hut Code

Figure 5.3: Sequential Barnes-Hut
map and a programmer-supplied function \texttt{ForceOp} that performs the force computation in parallel on all the tiles. The bodies can be then updated in parallel by \texttt{AdvanceOp}, and finally this computation iterates until completion.

Having implemented the algorithm, its parallel performance was then tuned through trying several different tiling strategies. Specifically, three different tiling strategies were implemented:

- **Naive** - This strategy evenly distributes the bodies into tiles. Bodies are assigned to tiles in the order in which they are originally initialized. This is roughly equivalent to parallelizing a \texttt{for} loop that iterates over the bodies in a sequential implementation. This is the baseline performance that does not use the \texttt{tile} operator.

- **Locality** - This strategy seeks to exploit locality inherent to the computation. Recall that the force computation performs a treewalk of the Octree. Siblings in the tree are reasonably close to each other in space and are likely to have similar treewalks. This strategy “tiles the leaves of the Octree” by chunking bodies in the order in which they appear in a walk of the tree into tiles, exploiting temporal locality. This is probably the natural way to do it. We chose to not exploit spatial locality since bodies do not share cache lines in our implementation.

- **Load Balance** - The previous strategy exploits locality in the computation, but it does not necessarily provide execution that is load balanced. Tiles might perform different amounts of work. In the case of Barnes-Hut, work equates to force computations performed by the bodies. This strategy seeks to exploit locality in a fashion that also balances the load. It does this through tiling the tree like the \texttt{Locality} strategy and bounding the amount of work that a tile can perform. In the implementation this is done with Cost Zones [43]. The number of force computations a body performs in the previous iteration is used as an estimate for the amount of work in this iteration. This
is a reasonable estimate as bodies are not expected to quickly vary in space. The total amount of work is computed by summing the estimates of each body and then this determines a bound for the amount of work per tile.

Experiments were run using 100000 bodies that were initialized according to the empirical Plummer model. Figure 5.4 and Figure 5.5 show the performance of the three different tiling strategies on both systems. Speedup is computed against a sequential reference implementation in C++ adapted from the Lonestar Benchmarks [31]. One tile is used per processor in each case. The points on the plots represent the average of several runs whose execution times varied by approximately one percent. Naive scales well as the number of processors increases, obtaining a whole-program speedup almost equal to the number of processors used. The superlinear speedups shown in Figure 5.4 and Figure 5.5 by the Locality and Load Balance tiling strategies for the Barnes-Hut example result from exploiting locality. These two strategies partition the data so that siblings in the tree will be in the same tile. Since siblings in the tree are close in the space, they are likely to access the same bodies. Thus, this results in the re-use of the bodies accessed during the force computation of the previous body. Naive performs a static partitioning that does not take into account spatial proximity; this can result in bodies accessing completely different bodies during the force computation, with no re-use. Figure 5.6 displays the number of L2 line misses for the force computation on the Xeon E5405 using eight processors. An order of magnitude drop in the number of misses is observed when comparing the performance of Naive to that of the Locality and Load Balance strategies, illustrating the exploitation of locality by these strategies. One can also observe in Figure 5.7 the standard deviation in the average number of L2 misses per tile when using eight processors. Since Load Balance seeks to balance the computation by performing roughly the same amount of work per tile, defined as bodies accessed during the force computation, one expects that each tile would incur roughly the same amount of misses. Figures 5.6 and 5.7 show that while Load Balance has a slightly higher average
Figure 5.4: Performance of Tiling Strategies for Barnes-Hut (Xeon E7450)

Figure 5.5: Performance of Tiling Strategies for Barnes-Hut (Xeon E5405)
Figure 5.6: Cache Performance (NP=8)

Figure 5.7: Standard Deviation of Avg # of Misses Per Tile (NP=8)
number of misses per tile than *Locality*, it exhibits more balanced behavior.

Expressing Barnes-Hut using Ravenna provides many advantages. In each case, the expression of the algorithm did not change, only the function provided to tile. Indeed, if one removes the call to the `tile` operator in Figure 5.2, the remaining pseudocode expresses the original unoptimized algorithm, seen in Figure 5.3. Requiring the programmer to manually implement tiling would result in a program that is less clean. Note that expert programmer might have to compute additional information needed by the tiling function, but such extra information is separate from the core algorithm and only a tuning concern.
5.2 Clustering

In this example, a clustering algorithm illustrates the use of tiled sets for locality. The algorithm chosen is k-means [13], which is a common clustering algorithm used in pattern classification. The algorithm works on a set of $N$ points and randomly picks $k$ points as the centers. Next, it iterates over the set of $N$ points by computing the distance from each point to the center and assigning each point to the nearest center. At the end of this step new centers are computed as the centroids of each cluster. The algorithm iterates until convergence, that is, no point changes the center to which it was assigned.

Two major data structures implement this algorithm with tiled sets. The first is a set of points, each containing the location of the point, the center to which it is currently associated, the shortest distance observed this iteration, and closest center seen so far. This set is tiled for parallelism. The other structure is the set of centers. This structure contains the location of the center, an $x$ accumulator, a $y$ accumulator, and a points count. This set is tiled for locality and then replicated (through the `spread` operator) for each processor, resulting in a tiled object with multiple levels of tiling. The number of centers in the tile was selected first from the size of the centers and then verified empirically by testing several sizes.

The tiled algorithm proceeds as Figure 5.8 shows. First, the centers are initialized to the first $k$ points of the input set and then it loops until convergence. In each iteration of the loop, the `SELECT_CENTER` function shown in Figure 5.9 computes the nearest center for each point. In this function, the tiling of centers introduces some complexity, because the nearest center cannot be computed until all the points have gone through all the centers. Thus, a final pass is needed that assigns each point to the nearest center. Notice that the function `SELECT_CENTER` is invoked through a `map` operator, so it will be applied in parallel across the tiles of the points set. After `SELECT_CENTER`, the new centers are computed through the `UpdateCenters` function, which runs sequentially because it represents a very small fraction
of the total time. Expressing UpdateCenters with data parallel operators can be done using
the mapReduce operator described in Section 2.2.

Two versions were implemented using data parallel operators, one with tiling and one
without. The performance of k-means was tested with a randomly generated data set with
$2^{18}$ points searching for 1024 centers. Figures 5.10 and 5.11 show the speedups of both
versions compared to a baseline untiled sequential version for both systems. One can see
from the figure that both versions scale almost perfectly with increasing number of processors
compared to the best sequential version with tiling. This is reasonable as the problem is
very parallel. However, the tiled version obtains roughly thirty percent more speedup over
the version without tiling. This super-linear speedup comes from the increased locality in
the tiled version. Figure 5.12 shows the L1 cache misses that each version incurs on the
Xeon E5405 system. For all the numbers of processors the difference is about two orders of
magnitude. This clearly shows the locality advantages in the tiled version.
1 TiledSet points, centers;

3 Kmeans(TiledSet points, int number_centers, int NP)
4 {
5   centers = SELECT(number_centers, points);
6   centers.spread(0, number_processors);
7   while (not_converged){
8     points.map(SELECT_CENTER(), centers);
9     centers = UpdateCenters(centers);
10   }
11   return centers;
12 }

Figure 5.8: Parallel k-means

1 SELECT.CENTER(TiledSet.tile points, TiledSet centers)
2 {
3   foreach tile in centers {
4     init();
5     foreach point in points {
6       foreach center in tile
7         if (nearer(center, point.best)) update(point.best);
8       }
9     }
10   foreach point in points{
11     add(bestcenter(point), point);
12     if (center(point) changed)
13       not_converged = true;
14   }
15 }

Figure 5.9: SELECT.CENTER()
Figure 5.10: Performance of K-means (Xeon E7450)

Figure 5.11: Performance of K-Means (Xeon E5405)
5.3 Breadth-First Search

The first graph algorithm examined was Breadth-First Search, or BFS. It is a simple yet important graph algorithm used for many applications such as finding connected components, shortest paths, or maximum flow. BFS examines all the vertices in a graph that are the same distance from the root before examining those at the next level. BFS can be naturally expressed as a sequence of data parallel operators applied to sets. Figure 5.13 shows a data parallel version of BFS. This version has two data parallel operations, FIND_NEIGHBORS and MARK_NEIGHBORS, that are applied to two sets of vertices, work_list and neighbors. The first, FIND_NEIGHBORS finds all the neighbors of every vertex in the set work_list and adds them to the set neighbors. The second, MARK_NEIGHBORS, marks the level of the vertices in neighbors from the root, if necessary, and adds newly marked vertices to work_list to be examined in the next iteration.

The data parallel implementation of BFS described above maps directly to a sequential implementation. Such an implementation would iterate over all the elements of the set and compute the appropriate result. However, executing this algorithm in parallel introduces a few difficulties. In FIND_NEIGHBORS, identifying all the neighbors of the vertices in work_list can easily occur in parallel, but modifying neighbors to insert the found vertices requires that it be implemented with either a concurrent data structure or that its access is protected by locks. A similar situation occurs in MARK_NEIGHBORS. An alternative implementation would tile the sets work_list and neighbors. Each processor would only process the elements in the tiles that it owns, removing the need for locks or concurrent data structures. This style of implementation would also be more amenable to execution on a distributed-memory platform. However, tiling creates a different issue to address. Tiling BFS maps the vertices of the graph identified by FIND_NEIGHBORS in a one-to-one fashion to a tile. If a tile finds that the neighbor of a vertex that it owns maps to a different tile,
Figure 5.12: Cache Performance of K-means (Xeon E5405)

```
1 Set work_list;
2 Array levels;
3 Set neighbors;
5 BreadthFirstSearch(State start)
6   work_list.insert(start);
8 while(work_list not empty) {
9   work_list.map(FIND_NEIGHBORS(), neighbors);
11   neighbors.map(MARK_NEIGHBORS(levels), work_list);
12 }
```

Figure 5.13: Data Parallel Breadth-First Search
it must somehow communicate this vertex to the appropriate tile. Finding a natural way to express this communication was what first brought about Ravenna’s tile operator.

The tiled implementation did not investigate different tiling strategies to determine their effects on performance since the initial exploration of a data parallel graph search only sought to examine the feasibility of expressing graph algorithms using tiled data parallel operators. The implementation of BFS works on graphs that are loaded into memory from a file. Graphs are represented in memory as an adjacency list. The graphs are statically partitioned into tiles by evenly distributing blocks of vertices to tiles. This strategy was used in order to match the behavior of the computation when it is expressed with parallel loops or in SPMD fashion. Since Ravenna is built on top of TBB, it is simple to overdecompose the problem to create more tiles than processors. This allows the work-stealing runtime to dynamically schedule work and hopefully reduce load imbalance.

Figure 5.15 show the performance of tiled data parallel BFS on a 32-core Xeon L7555 system. The algorithm was run on three randomly generated graphs of sizes four million, eight million, and sixteen million vertices. Each vertex has between 10 and 20 edges. Experiments were run using varying numbers of tiles, from fewer tiles than processors to more tiles than processors. Since the vertices are statically partitioned into tiles, the resulting computation can be imbalanced since the number of edges that each tile might examine in each iteration can vary. For each of the three graphs shown in Figure 5.15, overdecomposing the computation provided better performance, offering up to 3x speedup over using one tile per processor. An alternative to overdecomposition for BFS is to identify a tiling strategy that better partitions the amount of work performed by the search. For example, one could examine the structure of the graph to determine the partitioning or one could partition the edges instead of the vertices.
1 TiledSet work_list;
2 Array levels;
3 TiledSet neighbors;

5 BreadthFirstSearch(State start)
6     work_list.insert(start);
8     while (work_list not empty) {
9         work_list.map(FIND_NEIGHBORS(), neighbors);
10        neighbors.tile(strategy());
11        neighbors.map(MARK_NEIGHBORS(levels), work_list);
12     }

Figure 5.14: Data Parallel Breadth-First Search with Tiling

Figure 5.15: Tiled Breadth-First Search
5.4 Best-First Search

Search algorithms are an important class of algorithm that can be used to solve discrete optimization problems. Examples of problems that can be formulated as such include planning and scheduling, VLSI layout, and motion planning. This example implements a form of Best-First Search called A* [19] described in [17]. Best-First search uses a heuristic to guide which states to examine, preferring to examine states that are more likely to quickly lead to the solution over those that are less likely to do so. For this problem, a solution is the shortest path from some initial state to a goal. A good example of this type of search problem is the Fifteen Puzzle. The Fifteen Puzzle is a simple game played on a four by four grid of tiles with one tile missing. One solves the puzzle by sliding tiles, effectively swapping the positions of a single tile and the hole, until the goal configuration is reached. If one considers the different configurations of the puzzle as states in a graph and the moves one makes as edges, then solving the puzzle is reduced to performing a graph search. Figure 5.16 illustrates a small section of the graph from a particular starting state. This example performs its search on a graph that is not fully expanded in memory before the search begins as the search space contains approximately 16! states. Consequently, the search graph is expanded as the search progresses.

The heuristic, as previously mentioned, guides the order in which the search graph is expanded. It is possible to reach the same state via different paths so the search is a graph, not a tree, and checking for duplicates is required to avoid searching in cycles. The heuristic assigns every state a value. Lower values represent states that are more likely to quickly lead to the solution. The heuristic consists of two components. The first is the number of steps taken on the search to reach the current state. The second component is a lower bound on the number of transitions required to reach the goal from the current state. In the case of the puzzle, an accumulated Manhattan distance provides this lower bound. The Manhattan
Figure 5.16: 15 Puzzle Search Graph
distance measures the number of horizontal and vertical puzzle tiles each piece is from its position in the goal state.

The tiled implementation of this algorithm is shown in Figure 5.17. It makes use of Ravenna’s tiled objects, functional tiling, and map operator. For the puzzle, map applies three programmer-specified functions in parallel across the tiles of the sets:

- **SELECT_BEST** - This function, when applied to a tile of work_list, selects several of the best states. In this case, best means those states that have the most favorable heuristic values. Several states are selected each iteration to amortize the overhead of parallelism. This is a necessary deviation from the sequential algorithm in order to obtain good performance. The selected states are then removed from the work_list set and added to the best_states set.

- **EXPAND** - This function takes as input the states selected by SELECT_BEST. The selected states are then added to the set seen. This set keeps track of all the states that have been previously expanded in the search in order to avoid searching in cycles. The operator then generates, or expands, all the states that can be reached from the selected set of states. These states are added to the set children.

- **UPDATE** - This function iterates over the expanded states. For each of the newly expanded states, it first tests if it already exists in either work_list or seen. If it is contained in neither, the state is added to work_list. If the state is already in work_list but the newer state has a lower cost, it replaces the one already in work_list. Similarly if the state is in seen but the new state has a lower cost, the state is removed from seen and added to work_list.

Note that the tiling strategy might not map the new states produced by EXPAND to the tiles whose elements created them. The tile primitive is then used to correctly communicate these new states to the tiles to which the strategy maps them.
5.4.1 Different Tiling Strategies

Since this computation makes use of Ravenna’s functional tiling, we can now tune its parallel performance through testing four different tiling strategies:

- **Fine Grain** - This strategy makes use of characteristics of the computation to cyclically distribute states among the tiles. It does this through using the Manhattan Distance component of the cost heuristic.

- **Coarse Grain** - This strategy is very similar to **Fine Grain** in design. However, instead of using the Manhattan Distance directly, it uses half the Manhattan Distance. This “coarsens” the distribution by mapping two consecutive Manhattan Distances to the same tile, reducing the amount of communication done by calls to `tile`.

- **Random** - This strategy randomly maps states to tiles.

- **Identity** - This strategy most closely resembles a traditional task parallel implementation of this computation. The search begins sequentially until enough states are in the `work_list` to provide each tile with initial work to execute in parallel. At this point, states remain in the tile whose elements created them, modeling the behavior of several threads concurrently performing sequential searches. This strategy can be viewed as the naive way to parallelize this problem. It does not use the `tile` operator as states are never communicated between tiles.

Before comparing their performance, one can comment on the underlying mechanisms of each strategy. Since the computation is performing a heuristic search, an optimal and balanced execution would evenly distribute *good* states among all the tiles. By *good*, one means states that have lower cost. The strategies **Fine Grain** and **Coarse Grain** use the Manhattan distance as an approximation of this. This is a reasonable approximation as decreasing Manhattan distances are usually accompanied by increasing steps searched so far,
the second component of the cost function. Consequently, these strategies do a reasonable job of spreading the best states, those with lowest cost, around. Coarse Grain does this less frequently than the first. Two consecutive Manhattan distances get mapped to the same tile as opposed to one in the case of Fine Grain. As one can see, this can affect the scalability of this strategy as it effectively limits the amount of available parallelism later in the search. It does this because as the search progresses, the Manhattan distances of new, good states are lower than those of the states that created them. If these new states with shorter distances cannot be evenly distributed, then imbalance will occur and processors may perform suboptimal work. The latter two strategies do not use any information specific to the computation. When better information is not available, random mapping is a reasonable strategy to try. The last strategy attempts to remove any communication between tiles and better exploit any locality there might be.

Figures 5.18 and 5.19 show the average performance of the four tiling strategies on several initial configurations. Performance is presented as speedup over a sequential implementation. Several runs were performed for each initial configuration, with the average of each run being used to determine speedup. Execution times for the runs varied by approximately one percent. Figure 5.20 shows the speedups for each initial configuration. Immediately one can observe that both the Random and Identity strategies exhibit both poor speedups and scalability. This shows that, for the puzzle problem, these computation agnostic strategies do not do a good job of balancing the computation, or spreading out good states to the different tiles. Figure 5.21 shows the cache performance for one of the initial configurations of the puzzle using several tiling strategies. Identity conceptually should best exploit locality since it never communicates states between tiles. However, the best states that each iteration expands are not necessarily the states that were created in the last iteration, and one can see that cache misses do not determine the performance of the tiling strategy for this algorithm. Indeed having a balanced distribution of the good states appears to be
the dominant factor in determining performance for the puzzle. Both the Fine Grain and Coarse Grain strategies scale fairly well up to eight tiles, both obtaining speedups of about four to five over the sequential base. Indeed Coarse Grain tends to outperform Fine Grain up to eight tiles. However, it does not scale when going to sixteen tiles. Recall that both Fine Grain and Coarse Grain use the Manhattan distance to map states to tiles. This distance is bounded for the puzzle problem of this size. As the search increases, there are more states to examine deeper into the search, with decreasing Manhattan distances. Using half the Manhattan distance for the second strategy is simply too coarse, as the mapping becomes unable to evenly distribute good states among the tiles using sixteen tiles. One would expect the first strategy will also face scalability issues as the number of parallel execution resources increases.

5.4.2 Centralized Data Structures

In the data parallel implementation described above, each processor expands states from its own tile of work list each iteration. One of the uses of tiling in that version is to partition structures like the work list so that each processor does not have to acquire locks in order to access them. An alternative data parallel implementation would have a globally shared work list instead of a local list per tile. At each iteration, a chunk of states would be removed from the front of the list and passed to processors to expand in parallel. Such an implementation could be advantageous because it ensures that the globally best states are expanded each iteration. Consequently, several data parallel implementations of this version of the search were also examined.

A natural way to implement this strategy is to protect access to shared structures, such as work list or seen, with locks. Since this was implemented as a data parallel algorithm rather than a task parallel algorithm, some locks were not necessary as barriers exist between data parallel operations. For example, one does not need to worry that extracting the best
1 TiledSet work_list;
2 TiledSet seen;
3 TiledSet best_states;
4 TiledSet children;

5 BestFirstSearch( State start )
6     work_list[ strategy(start) ].insert(start);
7
8     while (not done) {
9         work_list.map( SELECT_BEST(), best_states );
10
11             if ( found_solution(best_states) )
12                 done = true
13             else {
14                 best_states.map( EXPAND(), children, seen );
15                 children.tile( strategy() );
16                 children.map( UPDATE(), work_list, seen );
17             }
18     }
19 }

Figure 5.17: Data Parallel Best-First Search with Tiling

Figure 5.18: Performance of Puzzle Tiling Strategies (Xeon E7450)
Figure 5.19: Performance of Puzzle Tiling Strategies (Xeon E5405)

Figure 5.20: Performance of the Initial Configurations (Xeon E7450)
Figure 5.21: Puzzle Cache Behavior
states from the \texttt{work\_list} will happen at the same time as new states are being added.

Additionally, there are two main categories that could describe the implementation strategy: coarse and fine. These categories refer to the locking strategy used to protect access to shared data. Coarse locking uses a single lock to protect access to the entire structure. While this is the easier of the two to program, it is clear that the overhead of this approach can be excessive since two threads are forbidden to modify the structure concurrently, even if they are modifying disjoint sections. Consequently, the other approach was examined, using concurrent data structures that support fine-grain locking in order to allow concurrent modification. Examples of such data structures can be found in Java’s \texttt{java.util.concurrent} package or Intel’s Threading Building Blocks.

Intel’s TBB provides \texttt{concurrent\_vector} and \texttt{concurrent\_hash\_map} classes that can, in many cases, be used as drop-in replacements for their STL counterparts. \texttt{concurrent\_hash\_map} differs slightly in that it provides accessor objects that are used to lookup and insert items into the map. These objects provide the fine-grain locking that is desired. \texttt{concurrent\_hash\_map} is a good fit for the \texttt{seen} structure in the algorithm. It provides concurrent operations on the map as well as having reasonably fast lookup and insertion. Likewise \texttt{concurrent\_vector} is suitable for implementing the structures \texttt{best\_states} and \texttt{children}. These require no particular ordering of the elements inside them and only need to support concurrent insertion. However, neither is a suitable structure for \texttt{work\_list}. Recall that the tiled implementation uses a priority queue to implement the \texttt{work\_list}. A priority queue allows quick access to the best states and insertion of new ones. Consequently, it was decided to write a concurrent data structure that implemented a priority queue. However, on account of the data parallel nature of the implementation, the only operation that can happen concurrently is insertion. Also, a sorted list was chosen to represent the queue instead of a heap due to the reduced complexity for an implementation.

More specifically, a skiplist \cite{39} was chosen in order to more quickly find the correct place
to insert a new item into the list. A skiplist is a sorted list where a list node can possess multiple next pointers. The lowest level of pointers implements a regular sorted list. The upper levels skip over nodes or sections of nodes in order to enable logarithmic searching of the list. Since **UPDATE** is the only point where the priority queue is concurrently updated, the skiplist only needs to implement concurrent insertion and not every possible operation. Concurrent operation is implemented by locking accesses to the next pointers that would point to the newly inserted node. These locks were implemented using the x86 compare and swap intrinsic operation instead of mutex objects provided by a library.

The concurrent skiplist was integrated into the data parallel search with global structures. However, while performance of the resulting program improved over the version that used coarse locking, it still had little to no speedup over a sequential implementation. This is due to the nature of the puzzle computation. **EXPAND** and **UPDATE** do very little actual computation. Consequently, the overhead of even fine-grain locking dominates execution time. In the case of the puzzle, using tiling to create local structures for each task/tile appears to be a better strategy than using global structures.

### 5.4.3 Overheads

In order to measure the overhead of implementing the examples presented above using Ravenna, implementations of the example computations were also written using TBB. These implementations manually implemented functional tiling. In the cases of Barnes-Hut and k-means, the overheads of the library were negligible. However, the puzzle example illustrates the overheads of functional tiling in Ravenna. Figure 5.22 shows the performance of the **Fine Grain** tiling strategy for the puzzle problem for both the Ravenna and TBB implementations. The performance difference is similar for the other strategies. The TBB implementation shows minor gains in performance when using two or four processors, but much more when using eight. The TBB version of the puzzle implements an optimization
Figure 5.22: Library Overhead
that is currently unavailable in Ravenna. In the algorithm shown in Figure 5.17, there is a call to `map` followed by a call to `tile` followed by a call to `map`. The first `map` operates on the set `best_states` and writes its output to the set `children`. `tile` is then applied to `children`, which is then used as input to the second `map`. Recall the implementation the `tile` operator described in Section 2.4. The `tile` operator uses internal storage, or "buckets", to communicate the data between tiles. In the TBB implementation, the first `map` that applies `EXPAND` to `best_states` writes its results directly into these instead of into `children`. Likewise, the `map` that performs `UPDATE` uses these buckets as input. This optimization eliminates two barriers, those between each `map` operator and the call to `tile`. More importantly, it eliminates the copies that must occur in to and out of `children`, insteading directly writing to and reading from the buckets used internally by `tile`. In the case of the puzzle, using more processors potentially increases the amount of states generated by the search each iteration since each tile on a processor expands a set of states. Recall that the implementation uses one tile per processor. Consequently, calls to the `tile` operator must now copy more total data as more states exist globally. These extra copies introduced by the implementation negatively affect performance with increasing numbers of processors. The manually-optimized TBB implementation removes these copies and thus does not exhibit this penalty when increasing the number of processors. This result does not appear for Barnes-Hut when increasing the number of processors as the number of bodies is fixed.

5.5 Programmability

Each example in this chapter is implemented very naturally as a sequence of data parallel operators applied to tiled objects. The resulting codes are both simple and compact. The codes are data parallel, but they look and can be reasoned about like sequential codes. Parallelism is encapsulated inside of the operators, and the resulting codes are deterministic
since the functions applied by \texttt{map} are without side effects. Functional tiling allows expressing these computations based on non-array data types clearly using data parallelism. More importantly, it presents programmers with a simple yet powerful abstraction for optimizing the performance of their data parallel codes. The examples of Barnes-Hut and Best-First Search show how domain programmers can implement the core algorithm using data parallel operators and then the tuning expert can experiment with different tiling strategies in order to obtain the best performance.
Chapter 6

Related Work

Co-Array Fortran [36], an extension of Fortran, is a Partitioned Global Address Space, or PGAS, language that supports a SPMD programming model. Arrays can be declared with a co-array dimension. Doing so has each program image or thread allocate a local copy of the array. Threads can access remote data by indexing into the co-array dimension using the id of the thread that owns the desired data. CAF still requires effort on the part of the programmer to properly distribute data, but it greatly simplifies communication since it takes the form of indexing co-arrays. This also simplifies reasoning about the execution of the program as it is clear when communication occurs. The co-array dimension can be used to model a tiled array with one level of tiling. However, unlike Ravenna and several other approaches, it does not support nested parallelism or a global view of control.

UPC [6] is another PGAS language that extends C. Arrays declared as shared have their elements distributed to threads in either cyclic or block-cyclic fashion. UPC provides several collective communication operators as well as upc forall, which provides programmers a global view of control. upc forall partitions loop iterations based on affinity expressions. However, exploiting locality still requires manual restructuring and indexing by the programmer, and does not treat tiles as first class objects that programmers can directly
HPF [30, 20, 29] is an extension to Fortran 90 directed at parallel processing. It was an extension of earlier languages such as Fortran D [24], Vienna Fortran [10], and Connection Machine Fortran. HPF uses a data parallel programming model that provides programmers with a logical single thread of control. Parallel array computations are achieved through the use optional compiler directives for distribution of array data. HPF supported several common distributions such as block, cyclic, or block-cyclic. In addition to distribution directives, HPF provided alignment directives that specified an elementwise matching between different arrays. HPF also provided a runtime library that included several higher level data parallel operators. The HPF compiler would take the programmer’s code, annotated with these directives and operators, and produce a parallel program. However, different HPF compilers focused on different optimizations, and it was often difficult to reason about the performance of a given program based on its code. Part of this difficulty stemmed from the fact that communication was not explicit in the program. Also, HPF’s few options for data distribution and lack of support for non-array data types limited the expressivity of the language. HPF2 attempted to resolve some of the shortcomings of the initial design with optional supported extensions. One example of these is the addition of extra distribution patterns such as GEN_BLOCK and INDIRECT, which allowed more flexible mappings of array elements to processors.

ZPL [8, 14] is an array-based parallel language developed at the University of Washington. It provides programmers with a global view of data and computation, but its compiler generates SPMD code. All computations are written as array expressions that are distributed among the different threads or processors in an aligned fashion. When this cannot be done, higher level array operators are used, translations or replications, for example, giving programmers a ”WYSIWYG” model through which one can reason about performance of the program based on the operators used. ZPL supported two types of arrays, parallel arrays.
and indexed arrays. Parallel arrays are partitioned through the use of Regions that generalize index spaces. ZPL uses a set of rules based on the dimension of the data to determine how to partition the data using a block distribution among processors. Indexed arrays are replicated across each processor and are not used as a mechanism for parallelism. Programmers could create a single level of tiling in their computations by using arrays of indexed arrays, but this method could not support multiple levels of tiling. It also required programmers to explicitly modify the rest of the code to support such a layout. Lack of control over distribution as well as support for non-array data types also limited the expressiveness of the language.

Chapel [7, 9] recognizes the shortcomings of previous approaches such as HPF or ZPL. Chapel generalizes ZPL’s Regions to support other data types. It also adds user-defined Distributions as first class objects in the language. Chapel also offers a mechanism called Locales that model units of parallel architecture inside a machine. Chapel is among the most similar of current approaches to Ravenna, allowing programmers to use Distributions and Locales to obtain similar effects to first class tiles and functional tiling.

In Sequoia [15] tiles appear as the result of recursive task decomposition. In that sense, Sequoia is close to the dynamic partitioning operator proposed by the HTA [18] and implemented as a mechanism for parallelism in Ravenna, although Sequoia works on a hierarchy of tasks rather than on a hierarchy of data. Sequoia differs from Ravenna as it does not support the partitioning of data structures other than arrays and does not support functional tiling, as the data are not re-distributed as the computation advances. Notice that Sequoia is not a data parallel language.

The Galois system [33, 32] works on irregular Java programs that operates on data structures such as graphs, sets, and trees. The programming model is data centric in that it expressed computations through the use of special set iterators. These iterators take an element of a set and apply some operation to it, potentially in parallel with other elements. This parallel execution is done speculatively, potentially conflicting with the execution of
other elements. Conflicts are detected through clearly defined interfaces that must be implemented. These set iterators contain low-level interfaces for partitioning the data structures upon which the iterators work. However, similar to HPF, UPC, or ZPL the partition strategy is not used by the programmer. In Galois, the partition strategy is used by the runtime system to perform a partition-aware scheduling that maintains thread affinity or attempts to reduce conflicts of the parallel speculative execution. Thus, Galois can support the initial tiling of arbitrary data types, but it does not offer the expressiveness tile operator that re-partitions data structures as the computation advances. Notice that in Ravenna thread affinity is achieved because the library is built on top of TBB, which provides affinity-aware partitioners.

Another relevant feature of the Ravenna programming model is that data parallel operators are semantically separated by barriers. This programming model is also that of Bulk Synchronous Parallel (BSP) [48, 21] where computation proceeds in steps, alternating between a step of independent computation involving no communication and a step of communication involving no computation. Each step is terminated with a global barrier.

The notion of data parallel operators is not a new one. There are many existing operators that come from a variety of sources. Many operators were not initially designed with parallelism in mind, they were designed as compact means to express operations on all the objects in a collection. Examples of operators designed for this purpose include the \texttt{map} [35] and \texttt{reduce} [46] functions of LISP, the operation on sets of SETL [42], and the array, set, and tree operators of APL [28].

Data parallel operators are also found in the vector instructions of the SIMD machines, and the early array and vector processors, IlliacIV [1], TI ASC [49], and CDC Star [23]. Extensions to the instruction set of conventional microprocessors (SSE [27] and Altivec [16]) or languages for GPU hardware accelerators [34] are examples of array operators in modern machines.
Another source of data parallel operators are those of high-level languages and the libraries developed to encapsulate parallel computations. The functional data parallel language NESL [4] made use of dynamic partitioning of collections and nested parallelism. Thinking Machines developed data parallel extensions of Lisp (*Lisp). Data parallel operations on sets were presented as an extension to SETL [26] and discussed in the context of the Connection Machine [22], but it seems there is not much more about the use of data parallel operations on sets in the literature.

The recent design of a MapReduce [12] data parallel operation combining the map and reduce operators can be seen as a low level abstraction that Ravenna could have used to implement functional tiling. However, MapReduce is a low level operator that does not support any concept of tiling on its own.

6.1 Summary

Table 6.1 presents a summary of the features of several higher level parallel languages. In particular, it focus on four characteristics: sequential view of execution, global view of data, first class tiles that can be directly accessed and operated upon, and support for non-array data types. Many of the languages discussed provide programmers with a sequential view of execution, simplifying reasoning about the program. However, only Ravenna and the HTA treat tiles as first class objects that programmers can directly access. Chapel and ZPL can approximate first class tiles with arrays of arrays, but these are subject to several limitations. Finally, only Chapel and Ravenna support non-array data types. However, only Ravenna supports operations on first class tiles of non-array data types, and its functional tiling provides a powerful abstraction for expressing several different types of computations.
<table>
<thead>
<tr>
<th></th>
<th>Seq. View</th>
<th>Global Data</th>
<th>1st class Tiles</th>
<th>Non-Array DT</th>
</tr>
</thead>
<tbody>
<tr>
<td>CAF</td>
<td></td>
<td>√</td>
<td></td>
<td></td>
</tr>
<tr>
<td>UPC</td>
<td></td>
<td>√</td>
<td></td>
<td></td>
</tr>
<tr>
<td>HPF</td>
<td>√</td>
<td>√</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ZPL</td>
<td>√</td>
<td>√</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Chapel</td>
<td>√</td>
<td>√</td>
<td>-</td>
<td>√</td>
</tr>
<tr>
<td>Sequoia</td>
<td>√</td>
<td>√</td>
<td></td>
<td></td>
</tr>
<tr>
<td>HTA</td>
<td>√</td>
<td>√</td>
<td>√</td>
<td></td>
</tr>
<tr>
<td>Ravenna</td>
<td>√</td>
<td>√</td>
<td>√</td>
<td>√</td>
</tr>
</tbody>
</table>

Table 6.1: Comparison of Features
Chapter 7

Conclusions

7.1 Conclusions

This thesis presents ideas on the use of higher level data parallel operators as a mechanism for programming parallel processors. It illustrates these through Ravenna, a library of data parallel operators for shared-memory multiprocessors to facilitate the writing and optimization of parallel programs. Programs in Ravenna are represented as a sequence of data parallel operators applied to tiled data types. The resulting programs can be portable across different classes of machine. Ravenna optimizes the performance of programs through tiling, and it provides programmers with several mechanisms to tile their computations. In particular, it provides a new abstraction called functional tiling that generalizes the specification of tiling for an object so as to apply to many classes of computations.

The key contributions of this thesis are as follows:

• A library of higher-level data parallel operators for shared-memory - The Ravenna library allows programmers to express their applications as a sequence of data parallel operators. This simplifies programming as it allows programmers to reason about their programs sequentially, since parallelism is encapsulated inside the operators. It pro-
vides primitives for programmers to write scalable parallel computations, since scalable computations are data parallel.

• Portability - The implementation of the SPIKE solver for banded linear systems shows that it is possible for programs written as a sequence of higher-level data parallel operators to be portable across multiple platforms. In the case of SPIKE, the same code can run on either the Ravenna library for shared-memory or the HTA library for distributed-memory.

• Support for non-array data types - Ravenna does not limit programmers to writing data parallel computations on dense arrays. It provides programmers abstractions to write data parallel computations on arbitrary data types like sets or graphs.

• Functional Tiling - Ravenna provides the programmer with several tiling primitives, and it introduces the concept of functional tiling, which generalizes previous tiling approaches. Functional tiling provide a separation of concerns between the expression of an algorithm and optimizing its performance. It treats the specification of tiling as a first-class object, and this allows programmers to easily test different tiling strategies in order to find the best performance for their programs.

• Evaluation of Higher Level Data Parallel Operators as a Programming Model - This thesis examines several applications implemented in Ravenna, including linear solvers for banded systems, n-body simulation, and graph search. In each case, the resulting code is simple and compact, and the resulting programs are able to use the operators implemented in the library to obtain parallel speedups for programs based on both arrays and other data types. This shows that a programming model based on higher level data parallel operators is a good model for many different types of computations, not just those based on dense arrays.
7.2 Future Work

Ravenna is currently implemented as a library for shared-memory systems. Currently, only programs based on statically tiled arrays are portable between Ravenna and the distributed-memory HTA library. However, many features of Ravenna, such as functional tiling, are amenable to a distributed-memory implementation. A distributed-memory implementation of Ravenna would enable the portability of many more classes of applications, as well as allow larger computations to be performed.

Functional tiling in Ravenna is a powerful primitive for optimizing the performance of applications. However, in the examples presented in this thesis, the best-performing tiling strategy was not immediately intuitive. A mechanism to identify good tiling strategies would be very beneficial to programmers when optimizing their codes. However, it is unclear if such strategies could be statically identified since, for the computations examined in this thesis, they required domain-specific information.

Many parallel array languages provide a set of default tiling strategies for programmers. These include strategies like block or cyclic distribution of array elements. It could be possible to examine several application domains, such as graph search, in order to discover a suite of default tiling strategies for non-array based programs. These strategies might not be ideal for the given computation, but they would provide programmers with a place from which to start if greater performance is desired.

7.3 Limitations of the Model

Ravenna’s programming model consists of expressing programs as a sequence of data parallel operators that have been extended with tiling. Operators are separated by barriers. The model prefers that computations on tiles are independent, but it does not enforce this, allowing programmers to use critical regions or modify shared state inside of an operation.
In this regard it gives programmers more flexibility over pure data parallel models at the cost of increased difficulty in correctness or reasoning about the program. While the model of expressing programs as a sequence of higher level data parallel operators applied to tiled objects has been shown to work well for the examples presented here, it is instructive to discuss where and when the model might fail to either express the computation or to obtain good performance.

Distributed-memory programs written using message passing rely on decomposing the computation into independent units of execution. Fundamentally, this is a form of tiling. Indeed, any parallel program at some point requires some sort of decomposition to create concurrent work. Since the Ravenna model supports data parallel operations on tiled objects, it is reasonable to expect that any program which can be represented in this manner can be expressed using Ravenna’s programming model.

However, being able to express a program in the model does not mean that the resulting program will be elegant. Unfortunately, the elegance of the resulting program cannot be determined until the program is written, so different computations must be examined on a case by case basis. This case by case examination was performed for each of the examples in this thesis, as well as others not discussed including data mining and mesh refinement algorithms. However, computations that are inherently data parallel seem likely to elegantly fit into the model. For the computations examined, little restructuring was required to express them in the model. In cases where such restructuring is less feasible or clear, computations can be expressed in the model by performing a map on an object that has as many tiles as desired tasks, and each tile could then execute a different task based on its location in the tiling.

Extending the model with additional operators could simplify or optimize the implementation of some applications. Programmers can compose existing operators to implement new operations, but in some cases writing a new operator can be more efficient. For the examples
presented in this thesis, the only new operator required was tile.

In regards to the performance of programs written using Ravenna’s programming model, one can expect that programs that fit into the model with little or no restructuring can obtain good performance when tiled appropriately. Likewise, one can expect that parallel programs that obtain good performance with distributed-memory implementations will obtain similarly good performance when expressed in the model. This is because tiling can be used to express the decomposition. Obtaining good performance will be more difficult for programs that require more restructuring to fit into the model. Similarly, some task parallel computations implemented in the model could be harder to optimize since the data parallel execution model of operators separated by barriers could place too many artificial constraints on execution. In these cases, one would expect a task parallel execution would allow a more natural implementation and simplify optimization.
Appendix A

Additional SPIKE Experiments

The following are additional experiments performed on the TU algorithmic variant of SPIKE using the four socket 32-core system with Intel Xeon L7555 processors running at 1.86 GHz. The experiments use several different sizes of matrix and, for each size, test several different bandwidths. The number of superdiagonals and subdiagonals used is denoted in the captions by $k$. The data points represent the average of several runs, with error bars showing the minimum and maximum speedup obtained for a given parameter set.
Figure A.1: TU, N = 131072, K = 16

Figure A.2: TU, N = 131072, K = 32

104
Figure A.3: TU, $N = 131072$, $K = 64$

Figure A.4: TU, $N = 131072$, $K = 128$
Figure A.5: TU, $N = 131072$, $K = 256$

Figure A.6: TU, $N = 262144$, $K = 16$
Figure A.7: TU, $N = 262144$, $K = 32$

Figure A.8: TU, $N = 262144$, $K = 64$
Figure A.9: TU, N = 262144, K = 128

Figure A.10: TU, N = 262144, K = 256
Figure A.11: TU, N = 524288, K = 16

Figure A.12: TU, N = 524288, K = 32
Figure A.13: TU, N = 524288, K = 64

Figure A.14: TU, N = 524288, K = 128
Figure A.15: TU, N = 524288, K = 256

Figure A.16: TU, N = 1048576, K = 16
Figure A.17: TU, N = 1048576, K = 32

Figure A.18: TU, N = 1048576, K = 64
Figure A.19: TU, $N = 1048576$, $K = 128$

Figure A.20: TU, $N = 1048576$, $K = 256$

113
Figure A.21: TU, $N = 2097152$, $K = 16$

Figure A.22: TU, $N = 2097152$, $K = 32$
Figure A.23: TU, $N = 2097152$, $K = 64$

Figure A.24: TU, $N = 2097152$, $K = 128$
Figure A.25: TU, N = 2097152, K = 256
Appendix B

Additional Barnes-Hut Experiments

The following are additional experiments performed for Barnes-Hut using the Xeon E5405 system at 2.0 GHz with two sockets and 8 cores. The experiments test the performance of the different tiling strategies on different problem sizes. The average execution time of several runs was used to compute speedup. Results are also shown where speedups are computed against an optimized sequential implementation that sorts the bodies in tree order. As is seen in the figures, this eliminates superlinear speedups that result from cache effects.
Figure B.1: 10k Bodies

Figure B.2: 50k Bodies
Figure B.3: 100k Bodies

Figure B.4: 10k Bodies, Optimized Sequential
Figure B.5: 50k Bodies, Optimized Sequential

Figure B.6: 100k Bodies, Optimized Sequential
Bibliography


