Optimizing Data Accesses for
Scaling Data-intensive Scientific Applications

Jae-Seung Yeom

Dissertation submitted to the Faculty of the
Virginia Polytechnic Institute and State University
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy
in
Computer Science and Applications

Madhav V. Marathe, Chair
Dimitrios S. Nikolopoulos, Co-chair
Keith R. Bisset
Calvin J. Ribbens
Martin Schulz

April 24, 2014
Blacksburg, Virginia

Keywords: Parallel systems, Software-managed memories, Distributed memories, Data locality, Scalability, Parallel discrete event simulation, Social networks, Contagion
Copyright 2014, Jae-Seung Yeom
Optimizing Data Accesses for Scaling Data-intensive Scientific Applications

Jae-Seung Yeom

(ABSTRACT)

Data-intensive scientific applications often process an enormous amount of data. The scalability of such applications depends critically on how to manage the locality of data. Our study explores two common types of applications that are vastly different in terms of memory access pattern and workload variation. One includes those with multi-stride accesses in regular nested parallel loops. The other is for processing large-scale irregular social network graphs. In the former case, the memory location or the data item accessed in a loop is predictable and the load on processing a unit work (an array element) is relatively uniform with no significant variation. On the other hand, in the latter case, the data access per unit work (a vertex) is highly irregular in terms of the number of accesses and the locations being accessed. This property is further tied to the load and presents significant challenges in the scalability of the application performance.

Designing platforms to support extreme performance scaling requires understanding of how application specific information can be used to control the locality and improve the performance. Such insights are necessary to determine which control and which abstraction to provide for interfacing an underlying system and an application as well as for designing a new system. Our goal is to expose common requirements of data-intensive scientific applications for scalability.

For the former type of applications, those with regular accesses and uniform workload, we contribute new methods to improve the temporal locality of software-managed local memories, and optimize the critical path of scheduling data transfers for multi-dimensional arrays in nested loops. In particular, we provide a runtime framework allowing transparent optimization by source-to-source compilers or automatic fine tuning by programmers. Finally, we demonstrate the effectiveness of the approach by comparing against a state-of-the-art language-based framework. For the latter type, those with irregular accesses and non-uniform workload, we analyze how the heavy-tailed property of input graphs limits the scalability of the application. Then, we introduce an application-specific workload model as well as a decomposition method that allows us to optimize locality with the custom load balancing constraints of the application. Finally, we demonstrate unprecedented strong scaling of a contagion simulation on two state-of-the-art high performance computing platforms.
This work was partially performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344 (LLNL-CONF-648533). This work has been partially supported by NSF Grants PetaApps OCI-0904844 and NetSE CNS-1011769, DTRA Grant HDTRA1-11-1-0016, DTRA CNIMS Contract HDTRA1-11-D-0016-0001, and NIH MIDAS Grant 2U01GM070694-09.

This research is part of the Blue Waters sustained-petascale computing project, which is supported by NSF award number OCI 07-25070 and the state of Illinois. Blue Waters is a joint effort of the University of Illinois and NCSA. This work is also part of the Contagion PRAC allocation support by NSF award number OCI-0832603.

This work has partially been supported by grants from NSF (CCF-0715051, CNS-0720750, CNS-0720673, OCI-0904844). We are grateful to our colleagues at the Barcelona Supercomputing Center (BSC) for providing us access to the IBM QS20 Cell blades.
Dedication

I dedicate this dissertation to my family. I cannot thank my father and mother enough for their unconditional love, endless support and encouragement. I could not have had better examples, role models and parents, who instilled in me the guiding principles and virtues I live by today. Above all, I want to express my deepest appreciation to my wife, for standing by my side through even the most difficult of times. Watching my two sons grow has been the greatest joy of my life. I am greatly indebted to my sister for all her help given to me while I am studying abroad.
Acknowledgments

I could not have successfully completed my Ph.D. study without the help of many people. Most importantly, I would like to express my greatest appreciation to the members of my dissertation committee, my advisors and my mentors. I am extremely grateful to Dr. Madhav V. Marathe, the chair of the committee and my advisor, who introduced me to the various cutting-edge research environments. I also thank him for the vision he shared, the persistent support and motivation. I am intellectually indebted to Dr. Dimitrios S. Nikolopoulos, the co-chair of the committee and my advisor, who has patiently persevered with me through my entire Ph.D. study looking over me with a warm heart and guided me to identify important problems. I owe a large debt of gratitude to Dr. Keith R. Bisset, a committee member and my mentor, for exposing me to the real-world problems in research and guiding me towards the right direction, which allowed me to make a seamless transition from my research to my career path. My sincere thanks to Dr. Martin Schulz, a committee member and my mentor while I was visiting LLNL, for the most insightful advice on the problems I had been having and for arranging the opportunities for me to move on to the next stage of my life. My heartfelt thanks to Dr. Calvin J. Ribbens, a committee member, for the suggestion, the encouragement and the compassion through the tough times. I thank Dr. Abhinav Bhatlele, my other mentor at LLNL, for the excellent guides in writing papers, the recommendation on Charm++ and the help in bridging my research and career opportunities. I thank the faculty members at VT, Drs. Eli Tilevich, Wu-chun Feng, Yang Cao, Naren Ramakrishnan, Adrian Sandu, Ali Butt, Kirk Cameron and many others for their good will and inspiration. I am deeply grateful to Dr. Laxmikant V. Kale for his generosity and consideration, and to many people from the UIUC PPL group, Eric Bohm, Abhishek Gupta, Lukasz Wesolowski, Nikhil Jain, Jonathan Lifflander, Chao Mei, Phil Miller and Harshitha Menon, for all their assistance with Charm++ and troubleshooting, in addition to the unwavering patience. I thank Robert Brunner and Celso Mendes for their consultation with NCSA Blue Waters. I am grateful to Rui Liu for his excellent support with perfsuite. I thank Ji Young Park for his aid with the Sequoia programming model. I have greatly benefited from working with Xavier Martorell as well as the resource at BSC for my Cell B.E. based research. I would like to express my gratitude to the Pearl group members for their cooperation, discussions and friendship: Filip Blagojevic, Matthew Curtis-Maury, Aleksandr Khasymski, Dong
Li, Ben Rose and Scott Schneider. I thank John Linford for his aid with Fixedgrid. I appreciate
the insights on graph partitioning given by Olga and Roger Pearce. I thank Peter D. Barnes for
the valuable information he shared on the performance of BG/Q. I thank Barry Rountree for his
comment on the random number stream. I thank Greg Bronevetsky for the exciting discussion
of future research. Special thanks to Bill Marmagas and Michael Snow for their assistance with
Shadowfax and the computing resources at VBI. I thank Chris Kuhlman for his help with Ph.D.
milestones. I thank Jiangzhuo Chen, Sandeep Gupta, Katherine Wendelsdorf, Bryan Lewis, Maleq
Khan and Samarth Swarup for their discussions regarding EpiFast, the graph problems, Enisi,
the epidemic processes and the synthetic population data. I thank Nidhi Parikh and Gaurav Tuli
for their input with the database schema. I appreciate the intellectual stimulations from talks by
Stephen Eubank, Henning Mortveit and Anil Vullikanti. I thank Maureen Lawrence-Kuether for
thoroughly proofreading my dissertation. I am indebted to the NDSSL staffs for their support:
Sandi, Sharon, Kayla, Dawen, Hannah, Edwina, and many others. I would like to express special
thanks to the NDSSL coffee club folks for the aroma we shared and for helping me get through
each day: Ashwin Aji, Balaji Sunapanasubbiah, Gaurav Tuli, Jose Cadena, Sudarshan Aji and
past members. I thank the near-field mates who used to sit next to me in the office: Dr. Jimenez,
Ming, Sichao, and Ashwin. I thank the rest of NDSSL people for the network dynamics: Hemanth,
Junwhan, Guarav, Huadong, Maksudul, Rajesh, Sherif, Shridhar, Siddharth, Spencer, Sushrut, Tariq,
Yifei, Zhao and countless others. I feel very fortunate to have interacted with Korean fellows in VT
community: Heayong, Hyungil, Joon-Suk, MyungKyu, Seung In, Seungwon, Sunshin, Sung Hee,
Tae-Hyuk, Young-Woo, and those who graduated long before me: Dong Kwan, Ji-Sun, Phl Sung,
Seonho and Yongju. I thank Kwangsoo Cho, Rohit Negi, Babu Pillali, Artur Dubrawski and Raji
Rajkumar for the kindness they extended to me. My sincere gratitude to Hosun, Jaeho, Jangwoo,
Myung and Sining for lasting friendship and for lending me their hands and ears.
Contents

1 Introduction
  1.1 Problem Description and Research Objectives ....................................... 3
      1.1.1 Handling Applications of Regular Accesses and Uniform Workload ....... 3
      1.1.2 Handling Applications of Irregular Accesses and Non-uniform Workload .. 4
  1.2 Research Accomplishments ................................................................. 5
      1.2.1 Handling Applications of Regular Accesses and Uniform Workload ....... 5
      1.2.2 Handling Applications of Irregular Accesses and Non-uniform Workload .. 5
  1.3 Dissertation Organization ................................................................. 6

2 Optimizing Strided Data Accesses for Nested Parallel Loops .................. 7
  2.1 Overview of Strider ............................................................................. 9
  2.2 Design of Strider ................................................................................ 11
      2.2.1 Multi-stride Data Access Optimization ......................................... 11
      2.2.2 Cell-Specific Optimizations ......................................................... 17
  2.3 Experimental Analysis ....................................................................... 20
      2.3.1 Experimental Platform ................................................................. 20
      2.3.2 Benchmarks ............................................................................... 20
      2.3.3 Experimental Results – Kernels ................................................... 22
      2.3.4 Experimental Results – Coordination between PPE and SPE ............ 26
      2.3.5 Experimental Results – Applications ........................................... 31
  2.4 Summary ............................................................................................. 38

3 Programming Abstraction for Strided Data Accesses ............................. 39
  3.1 A Working Example ............................................................................. 40
  3.2 Programming Complexity ..................................................................... 45
  3.3 Related Work ....................................................................................... 46
  3.4 Future Work ......................................................................................... 47
  3.5 Summary ............................................................................................. 49
4 Identifying Scalability Bottleneck from the Data with Heavy-tailed Properties

4.1 EpiSimdemics ................................................................. 51
   4.1.1 The EPISEMDEMICS Core Algorithm ......................... 53
4.2 Scalability Challenges with Social Network Data .................... 53
   4.2.1 A Simple Example ............................................... 54
   4.2.2 Limited Scalability with Power-law Data ...................... 56
4.3 Handling the Social Contact Data ...................................... 58
   4.3.1 Models to Estimate Workload ................................... 58
   4.3.2 Distributing the Population Data .............................. 61
   4.3.3 Scalability Bottleneck of the Population Data ................. 67
   4.3.4 Split Heavily-loaded Nodes .................................... 70
   4.3.5 Trade-off between Locality and Load Balancing in Graph Partitioning . . . 72
4.4 Related Work .............................................................. 74
4.5 Summary ................................................................. 76

5 Pushing the Limits of the Scalability of Social-network-based Simulations

5.1 Communication Optimizations ......................................... 77
   5.1.1 Message-driven Design ........................................... 77
   5.1.2 CHARM++ SMP Mode .............................................. 78
   5.1.3 Synchronization with Irregular Communication ................. 80
   5.1.4 Aggregating Fine-grained Messages ............................. 81
5.2 Performance Results ..................................................... 83
   5.2.1 Test Platforms ..................................................... 83
   5.2.2 Scaling Analysis .................................................. 86
5.3 Future Work ............................................................. 96
5.4 Summary ............................................................... 98

6 Conclusion ................................................................. 99

Bibliography ................................................................. 103

A Programming for Cell B.E. Processors .................................. 118
   A.0.1 Explicitly Managed Memory (EMM) Hierarchies ............... 118
   A.0.2 Cell B.E. Architecture Overview ................................ 119
   A.0.3 Off-chip Memory and Local Storage Accesses ................ 120
   A.0.4 Cell B.E. Programming Challenges ............................. 122
   A.0.5 Challenges to Achieve Sustained DMA Performance ........... 124

B Application Implementations on Cell B.E. ............................. 126
List of Figures

2.1 STRIDER runtime and external modules for Cell BE. The external modules are to be filled by programmers. .......................................................... 9

2.2 Variable depth buffering for the example of Table 2.1. P, S, E, BD, and $r_{max}$ stand for prologue, steady state, epilogue, buffering depth, and maximum reference depth respectively. A rectangular block represents a buffering step for an array. Figure 2.4 shows this step in more detail. ......................................................... 14

2.3 Buffering rotation types for an array: double-buffering (left), triple-buffering (center), and $d$-depth buffering (right). ......................................................... 14

2.4 A buffering step in detail. ................................................................. 15

2.5 Copy and transpose 2D arrays of three different sizes 1,000 times using 16 SPEs and eight different block shapes while relying on the two decomposition methods supported by STRIDER and one by SEQUOIA........................................ 23

2.6 Sustained bandwidth of the copy and transpose kernels for array copy and transpose implemented in STRIDER and SEQUOIA. This shows the impact of NUMA-aware page allocation on the bandwidth achieved by strided memory access. The kernels handle 2-D double type arrays of three sizes, 600×600, 1200×1200 and 1800×1800 using 30×40 blocks. ................................................................. 24

2.7 Scalability of the add-transpose kernel with the number of array pairs. The kernel adds multiple array pairs and transposes the output using 16 SPEs. This shows the effectiveness of STRIDER’s access pattern grouping and the management of simultaneous multiple accesses. STRIDER benefits from NUMA-aware page allocation. We use arrays of size 2048×2048, and blocks of size 16×16. Thus, SEQUOIA has no disadvantage from decomposition. RC stands for the case where the sum of rows from multiple arrays is transposed into a column. CR stands for the opposite case. . 24
2.8 Comparison of kernel load distribution. STRIDER achieves better balance than SEQUOIA does by the multi-level loop decomposition.

2.9 Maximum memory access bandwidth achieved by STRIDER (top and left) and SEQUOIA (right) while copying 2048×2048 array between the local stores of SPEs and the off-chip main memory. We use the shape of blocks (2×N) that are DMA-friendly accessing mostly contiguous chunks of memory but with one stride, where N is the width of the block. STRIDER without NUMA-aware page allocation (left) achieves overall 27% higher bandwidth than that of SEQUOIA (right). When STRIDER further takes advantage of NUMA, it achieves 75% of the theoretical peak (51.2 GB/s).

2.10 Performance of synchronization and parameter passing between a PPE and SPEs. This shows the execution time for 20,000 iterations of offloading an empty task with varying numbers of parameters.

2.11 The execution time of each application implemented in three different methods.

2.12 Application speedup versus execution time with one SPE.

2.13 Timing breakdown of applications on 16 SPEs. Kernel is the time spent in computational kernels. DMAwait is the DMA data transfer time that is not overlapped with computation. DMAprep is the time to prepare DMA lists. In Jacobi, Others accounts for signaling overhead between PPE and SPE and the cost of reduction operations. In PBPI, Others accounts for the cost of executing part of the workload between offloaded regions on the PPE and the cost of reduction operations. In Fixedgrid, Others accounts for the cost of executing part of the workload between offloaded regions on the PPE.
2.14 Two different implementations of Jacobi by using SEQUOIA (left) and STRIDER (right). To access 2-D data array on the off-chip main memory, each implementation manages data buffers of sub-blocks on local stores. The buffer rotates between a prefetch and a compute stage, and additionally a write-back stage if the access is for the read-write type operation. In SEQUOIA, the buffering depth, the number of stages in the rotation, is fixed. In STRIDER, the reference depth of access pattern determines the buffering depth as shown in Table 2.1. With SEQUOIA, we experimentally choose a rectangular block that minimizes the cost of redundant data transfers and the overhead of handling non-contiguous data at the same time. With STRIDER, the variable depth buffering removes redundant transfers along the innermost loop iteration. In this example, we choose the innermost loop to iterate over the non-contiguous dimension (column) of the 2-D array to remove redundant transfers along the horizontal boundaries (rows) of a block. Thus, we use linear blocks, i.e., segments of rows that do not span columns. STRIDER automatically reuse these segments from buffers in local stores whenever needed. For the same amount of data to transfer, we reduce the number of strided accesses. The temporal reuse not only avoids redundant data transfers, but also minimizes the DMA overhead of transferring non-contiguous data.

2.15 PBPI accesses a contiguous chunk of memory from each one-dimensional array in each iteration starting from a non-deterministic address.

2.16 PBPI with various optimizations. We use page interleaving between two NUMA nodes in pg-intlv and noSIMD and even splitting of pages using first-touch in pg-split. We use an optimized unrolled SIMD kernel in pg-intlv, pg-split, and noNUMA, and the scalar kernel with the system’s default page allocation scheme in noOpt.

3.1 An example of a nested loop.

3.2 A naïve SPE offloader for the loop shown in Figure 3.1.

3.3 The user kernel for the loop shown in Figure 3.1.

3.4 The PPE offloader for the loop shown in Figure 3.1.

3.5 An optimized alternative to the one shown in Figure 3.2 using blocking.

4.1 EPI SIMDE MICS core algorithm for each time step. We label sub-steps 1-2 as person phase, and sub-steps 3-5 as location phase.

4.2 The optimal 5-way partitioning result for minimizing (a) load imbalance and (b) edge cut. Node 1 has weight 8, Nodes 2-5 have weight 2, and Nodes 6-13 have weight 1 each.
4.3 The bound on the scalability ($S_{ub}/D$) is shown in terms of the size of data ($D$) and
the exponent of the power-law degree distribution ($\beta$), given the assumption on the
load as a linear function of degree. The scalability is defined as the upper bound on
speedup scaled by the size of data given the unlimited number of processors. This
numerically shows Equation 4.2. .......................................................... 56

4.4 The model to estimate the static workload of each location object on (a) Blue Wa-
ters and (b) Vulcan and (c) Sierra. These models approximate the mapping from
application’s variables to the relative processing time of a location object. These
models are platform-dependent, and built using piecewise linear regression and
cross-validation. The model for Vulcan relies on the four-dimensional feature space
while displaying here using two-dimensional features and the target. On the other
hand, the other models rely on one-dimensional feature space. By approximating the
model form as a linear function, we choose the feature space based on the five-fold
cross-validation. .......................................................... 58

4.5 The distribution of location vertex indegree in the seven population data sets. The
left plots show the distributions with the original graph. The right plots show those
with the graph modified as we discuss in Section 4.3.4. A data point with bin $10^1$, for
example, shows the number of vertices of degree $d$, where $10^0 \leq d < 10^1$, for a
particular data set of the seven selected states. ........................................ 59

4.6 The distribution of load per location vertex estimated by the model shown in Fig-
ure 4.4 with the property of data shown in Figure 4.5. The left plots show the
distributions with the original graph. The right plots show those with the graph
modified as we discuss in Section 4.3.4. .......................................................... 62

4.7 Two strategies to distribute location data approximating optimal locality using the
zipcode. One ($Z$) is based on the zipcode only and the other ($ZC$) is on the zipcode
and the sum of costs. In both cases, we sort locations by the zipcode, and then by
the location id within a same zipcode. Then, we divide location data evenly between
processors in terms of the number of locations ($Z$) or the sum of weights ($ZC$). With
the former, one processor takes L1 and L2, and the other takes the rest. With the
latter, one processor takes L1, L2, L3 and the other takes L4. ......................... 63

4.8 Each of the distribution strategies based on zipcode, $Z$ and $ZC$ shown in Figure 4.7,
categorizes locations into two types: home and non-home. It handles locations of
each type separately. Then, it assigns each person object to the processor on which
the home location is. .......................................................... 63

4.9 Distribution of computational load on home locations. The line connects successive
locations in the sequence ordered by the zipcode and the id. The size of a dot is
proportional to the load on the location. .......................................................... 65
4.10 Distribution of computational load on **non-home** locations. The distribution spreads over a larger area than that of home shown in Figure 4.9 as people near state borders may visit locations across the borders. ........................................... 66

4.11 The estimated speedup \( \frac{L_{tot}}{L_{max}} \) for location computation based on the load distribution across data partitions as a result of graph partitioning. This presents a tighter bound on speedup than \( S_{ub} = \frac{L_{tot}}{L_{max}} \) by incorporating the load balancing achieved by graph partitioning. It is evaluated for each of seven states over various number of partitions between 12 and 196,608. The left plots show the the results with the original graph. The right plots show those with the graph modified as we discuss in Section 4.3.4. ........................................... 68

4.12 The maximum of the estimated speedup shown in Figure 4.11. The left plots show the results with the original graph. The right plots show those with the graph modified as we discuss in Section 4.3.4. Each dot represents one of 48 contiguous US states and DC. ........................................... 69

4.13 Two methods to split a heavily loaded node, exploiting extra parallelism available. A heavy node is split into a pair, nodes 1 and 14, dividing the load. In addition, the split pair may (a) or may not (b) split the set of edges connected to the original node, depending on how the computation depends on the data. ............................... 70

4.14 Impact of decomposition threshold on the estimated speedup with Texas, New York, and Michigan data sets. We rely on a platform-independent load approximation for the decomposition and on the load model for Blue Waters (see Section 4.3.1) for the speedup estimation. **Original** represents the case without decomposition. **Absolute** represents the case with threshold fixed to \( m \), the largest weight of a sublocation. Otherwise, we set the threshold as \( t = \max \frac{L_{tot}}{P/k, m} \) where \( L \) is the sum of sublocation costs in the data, \( P \) is the maximum number of partitions set to 192K. \( S_{ub} \) represents the upper bound on speedup. \( S_{max} \) represents the maximum of the estimated speedup as a result of graph partitioning for various numbers of partitions up to 192K. ........................................... 72

4.15 The load balancing with the NY data decomposed as the case **Absolute** in Figure 4.14, improves as we relax the connectivity in the graph for partitioning. **MXn** represents the case where any vertex that has more than \( n \) edges drops randomly chosen edges and leaves \( n \) edges. **DAn** represents the case where any vertex that has more than \( n \) edges drops all the edges. **Full** represents the case where no edge is dropped. ........ 73
5.1 EPISIMDEMICS implemented in CHARM++. This conforms to the algorithm for a simulation iteration which consists of two computation phases, person phase and location phase, followed by a global state reduction, described in Figure 4.1. We create a chare array of each type, LocationManager (LM) and PersonManager (PM). We distribute location objects among the LM elements and person objects among the PM elements. The CHARM++ runtime then assigns a set of chares to each physical processor.

5.2 Two AMD Opteron Interlagos processors of a XE6 node in Blue Waters. In CHARM++ SMP mode, we choose to assign a CHARM++ runtime process on core 0 of each core module (CM) and a worker thread to each of cores 2, 4, and 6 on each CM as we discuss in Section 5.2.2.

5.3 Two aggregation schemes for fine-grained messages. (a) In manual buffering, each source (blue) maintains a set of buffers, one per destination. When a buffer is full, the entire buffer is packed and sent out as an aggregate message. (b) TRAM aggregates messages per peers in a virtual mesh topology. Users can specify the dimension of the virtual mesh at runtime. It maintains a buffer per destination peer on each processor.

5.4 Each Cray XE6 blade includes four dual-socket compute nodes, and two Gemini interconnects. The heat sinks of the two middle sockets are removed for the display. Each Gemini has two NICs, which is shown on the right towards the back of the blade.

5.5 NCSA Blue Waters, pictured on October 22, 2013. There are total 5,688 XE6 blades, shown in Figure 5.4, as well as 1,056 XK7 blades in the entire Blue Waters system.

5.6 Uncovering scalability bottlenecks. (a) unoptimized represents the initial implementation in CHARM++. The code optimizations discussed in Section 5.2.2 results in the one represented as optimized. (b) Ratio shows the ratio of the HW counter stats of optimized to that of unoptimized.

5.7 The average hardware counter values per thread, collected during processing the US data on Blue Waters, show the characteristics of EPISIMDEMICS code.

5.8 Identifying the best CHARM++ SMP setup. npXmw denotes the case of n smp processes per compute node and m worker threads per process.

5.9 Timeline plots of an iteration of EPISIMDEMICS using MI data and 4K cores on Vulcan (sampled to show 332 PEs). The blue and orange components show the person and location computation, respectively. The red represents the message handling of TRAM (see Section 5.1.4 and Figure 5.10). The time taken for the iteration is shown above each plot.
5.10 Each of 3,072 bars in a subplot shows the amount of data in MBytes received per PE over three iterations using MI data on Vulcan while relying on TRAM (see Section 5.1.4 for further details). The darker portion shows the amount receiving as the final destination PE and the lighter portion as an intermediate one. \textit{*-splitLoc} allows better load balance. \textit{GP*} and \textit{Z*} reduces total volume of communication. \textit{GP-splitLoc} and \textit{ZC-splitLoc} improve load balance. There exist large peaks when load balance is ignored (\textit{Z-splitLoc}) or suboptimal (without \textit{*-splitLoc}).

5.11 The maximum per-partition edge cuts (GP-splitLoc). Each data point represents the maximum per-partition edge cuts as a result of graph partitioning for Vulcan. The line of matching orders represents a hypothetical reference case where all edges are cut but they are evenly distributed across partitions, which is essentially calculated as the number of total edges divided by the number of partitions.

5.12 Strong scaling performance of EPI SIMDEMICS in speedup for selected state data on Vulcan (BG/Q).

5.13 Strong scaling performance of EPI SIMDEMICS in execution time for selected state data on Vulcan (BG/Q).

5.14 Timeline plots of two iterations of EPI SIMDEMICS using MI data and 4K core modules on Blue Waters (sampled to show 384 compute PEs). Each bar represents the activity of a processor over time progressing from left to right.

5.15 Amount of data received per PE in MB over six iterations using MI data and 4K core modules on Blue Waters while relying on manual buffering (see Section 5.1.4 for further details). \textit{GP} has the highest maximum communication volume while \textit{GP-splitLoc} has the lowest. Both have smaller average communication volume than \textit{RR} and \textit{RR-splitLoc}.

5.16 Strong scaling performance of EPI SIMDEMICS in execution time for selected state data on Blue Waters (XE6).

5.17 The \textbf{weak scaling} (left) and \textbf{strong scaling} (right) performance of EPI SIMDEMICS. The optimizations discussed in Section 5.2.2 cause the difference between red (circle) and green (triangle) points for each number of cores used in subfigure (a), and between red (circle) and blue (diamond) points in subfigures (c) and (e). The difference between blue (diamond) and green (triangle) points is attributed to the domain decomposition discussed in Section 4.3.4. \textit{TRAM} and \textit{mbuf} represent two different message aggregation methods described in Section 5.1.4. For strong scaling, we use as much resources as allowed on each platform and the whole continental US population data. Subfigure (b) shows the largest strong scaling up to 352K cores.

A.1 The overview of the Cell Broadband Engine architecture.
A.2 The memory interface controller (MIC) and the off-chip shared memory subsystem 121

B.1 The serial implementation of Jacobi. ........................................ 127
B.2 One of three loops of PBPI in the original serial implementation. .... 127
B.3 The SPE offloader for a loop of PBPI implemented in STRIDER. ...... 128
B.4 Two leaf task definitions of PBPI in SEQUOIA. We do not use the definition, Leaf
in our study. Instead, we use the external leaf task, Leaf_ext, the body of which is
not shown here. It utilizes SIMD operations which are not handled by the SEQUOIA
compiler. Sequoia allows to use such a task via the task mapping as shown in
Figure B.7. ................................................................. 129
B.5 Inner task definition of PBPI implemented in SEQUOIA. .................. 130
B.6 Task offloading via the entrypoint call of PBPI implemented in SEQUOIA. 130
B.7 Mapping specification of PBPI implemented in SEQUOIA. .............. 131
B.8 The SPE offloader for the column discretization task of Fixedgrid implemented in
STRIDER. ................................................................ 132
B.9 The PPE offloader of the Fixedgrid implementation in STRIDER. .... 133
B.10 The external leaf task of Fixedgrid in SEQUOIA. The bodies utilizing SIMD operations
are not shown here. ............................................................ 134
B.11 Task offloading via the entrypoint call of Fixedgrid implemented in SEQUOIA. .. 134
B.12 Inner task definitions of Fixedgrid implemented in SEQUOIA. ........ 135
B.13 Mapping specification for the column discretization task of Fixedgrid implemented
in SEQUOIA. ................................................................. 136
B.14 The SPE offloader of the Jacobi implementation in STRIDER: Part 1. The rest is shown
in Figure B.15. ................................................................. 137
B.15 The SPE offloader of the Jacobi implementation in STRIDER: Part 2. The rest is shown
in Figure B.14. ................................................................. 138
B.16 The PPE offloader of the Jacobi implementation in STRIDER. .......... 138
B.17 The leaf task of the Jacobi running on SPES implemented in SEQUOIA. .... 139
B.18 The inner task of the Jacobi running on SPES implemented in SEQUOIA. .... 140
B.19 The entrypoint of the Jacobi running on PPE implemented in SEQUOIA. .... 141
B.20 The task mapping file written for the Jacobi implemented in SEQUOIA. .... 142
List of Tables

1.1 Types of applications in terms of data access pattern and workload variation, and the chosen types for the study. .................................................. 3

2.1 An example of the reference depth and the variable depth buffering with $c[i] = f(a[i - 2], a[i - 1], a[i], a[i + 1], a[i + 2], b[i - 1], b[i], b[i + 1])$. ........................................... 13

2.2 The number of parameters and the size of parameters passed in the experiment shown in Figure 2.10. ........................................................................... 31

3.1 Loop parameters for the loop shown in Figure 3.1. ................................. 40
3.2 The descriptor of array double A[nx][ny][nz][5]. ........................................ 40
3.3 The loop parameters in the code shown in Figure 3.5. ............................... 42
3.4 The descriptor of array transformed for the code shown in Figure 3.5 from the original form in Table 3.2. ................................................................. 42
3.5 Lines of code added for each combination of programming method and application. offload shows the number of lines in the serial version corresponds to the offloaded computation. The numbers inside of parenthesis show the line counts of the task mapping file of SEQUOIA written in XML format by programmers. ............... 46

4.1 Population data of various sizes based on a 2009 American Community Survey and other data [1, 2]. US includes 48 contiguous states and DC. ......................... 52
4.2 The details of the example shown in Figure 4.2. ........................................ 55
4.3 List of symbols used. ................................................................................. 57
4.4 Data distribution strategies. RR stands for round-robin, and mimics random distribution. GP stands for graph partitioning. Z stands for the distributions based on the zipcode only. ZC stands for the one based on the zipcode and the cost. Both Z, ZC, and GP considers locality, and RR ignores it. Both ZC, and GP considers load balance. GP optimizes locality subject to load balance constraints. It exercises load balance by a heuristic. ZC optimizes load balance subject to locality constraints. Both Z and ZC approximate the optimal locality by the zipcode and the object id. Although RR does not apply explicit load balancing, it makes an indirect attempt by randomizing the load assignment.

4.5 The total load $L^{tot}$ and the maximum load per location before ($l^{max}$) and after ($\ell^{max}$) graph modification.

5.1 Platforms used. Blue Waters and Vulcan are petascale systems. (©www.top500.org, Nov 2013.)
Chapter 1

Introduction

The optimal method of managing data locality in data-intensive scientific applications varies by the application and the platform used, and has a profound impact on the scalability of such applications. Optimizing data accesses includes improving data locality and optimizing non-local accesses. In addition, to further improve performance, it may be necessary to trade-off data locality with other qualities such as load balance. In this study, we discuss how explicit handling of data locality affects the performance of data-intensive applications. Understanding how application specific information can be used to control the locality and improve the performance is one of the key factors to consider in designing future platforms supporting extreme performance scaling [3–5].

Our discussion is based on the condition where all data exist on Random Access Memory (RAM). It is not uncommon for a data-intensive application to handle a large amount of data exceeding the memory space visible to a single processor. The visible memory space may consist of multiple physical memories allowing a processor to access them by means of mechanisms transparent to applications. To access data existing in a memory space not visible to a processor, it is necessary to bring the data to a locally visible space by means of a method implemented in software. Within a memory, data are staged in a hierarchy, where a slow and low-priced component provides larger capacity at the stage farther from processors, and a fast and high-priced component serves a smaller capacity, but is located at a closer stage to processors.

Improving the locality of data aims to allow a processor to access data with the least delay or inefficiency when needed. The decision on the placement of data in space and time impacts the performance under various circumstances. For example, it can save the cost of accessing data on the memory not visible to a processor in a distributed memory (DM) system. Similarly, it can save the cost in shared memory (SM) systems with non-uniform memory access (NUMA), where the access cost is not even among the memories visible to a processor. It is also important for
efficiently accessing data through the memory hierarchy. Such a decision is often made in part by
heuristics implemented in hardware especially for managing data placement in a memory hierarchy.
In addition, there exist hardware platforms exposing the full control over the management of
data locality to programmers. The rest is up to application programmers or runtime software.
Application programmers may rely on abstract interfaces offered by high-level programming
frameworks based on languages or libraries. This work shows how a programmer can take
advantage of the explicit control of data placement to improve the scalability of data-intensive
applications and when it makes a significant difference using two case studies. Table 1.1 shows two
vastly different types of applications in terms of the data access pattern and the workload variation.

In the first case study, we discuss the techniques to optimize applications with regular accesses
and uniform workload such as those with regular nested parallel loops. In this case, we use a
platform with software-managed memories, specifically Cell B.E., which allows us to explicitly
manage data locality and apply software-prefetching to hide access latency while streaming data
through a limited local memory [6–9]. Cell B.E. offers the full control of locality management and
enables us to study every aspect of application-specific locality optimization. However, due to
the complexity of programming using the low-level interface, a majority of existing works has
focused on improving the programmability as we discuss in Section 3.3 rather than on taking full
advantage of the opportunity to optimize performance. Other platforms exist that automate certain
aspects of locality management by hardware in exchange for less locality control capability offered
to programmers for application specific optimization. For example, one of the state-of-the-art
platforms, Blue Gene/Q, supports L1 cache prefetching. It allows users to provide the access list
while the hardware takes care of scheduling the accesses [10]. A subset of the lessons learned from
Cell B.E. will be applicable to such a platform depending on which aspect of locality management
is offered to the programmer.

In the second case study, we use a contagion simulation over large irregular graph data as an
example of applications with irregular accesses and non-uniform workload. For this study, we
use large-scale distributed memory platforms including two state-of-the-art petascale platforms.
We investigate how the property of data affects the process of data placement, and eventually the
scaling performance of the application. While localizing data is critical for minimizing the cost
associated with accessing data over distributed memories, balancing load is important as well. The
challenge is that balancing load may require sacrificing locality or vice versa. The extreme variation
of load makes the locality optimization difficult or ineffective. The other two types not discussed
in this study can be treated as the most general case with irregular accesses and non-uniform
workload. In addition, there exist related studies which fall into the case with regular accesses and
non-uniform computation [11, 12], and the case with irregular accesses and uniform load [13]. The
latter provides an example of the integer sort problem with random memory accesses.
Table 1.1: Types of applications in terms of data access pattern and workload variation, and the chosen types for the study.

### 1.1 Problem Description and Research Objectives

#### 1.1.1 Handling Applications of Regular Accesses and Uniform Workload

Multi-core processors with explicitly-managed local memories enable application-specific optimization of data caching and prefetching in software [14–18]. Unfortunately, these capabilities are neither easily accessible to programmers, nor exploited to their greatest potential by current language, compiler, or runtime frameworks. Nevertheless, such a system is an attractive platform for optimizing data-intensive applications, particularly with regular memory accesses and uniform workload. Software-managed memories present a unique opportunity for optimizing such applications by enabling application-specific prefetching and caching to hide access latency, to reuse data and to manipulate data layout.

High-level parallel programming models exist for multi-core processors with explicitly-managed memories. While these solutions significantly improve the programmability, they still fall short in terms of performance, even in common cases as we discuss in Chapter 2. Software prefetching and caching of data accessed with unit and non-unit strides raises unique challenges, including the overhead for generating the transfers—which adds substantial software latency on the critical path. Furthermore, exploiting temporal reuse across partially overlapping blocks of data, while handled effortlessly with hardware-managed caches, is a difficult task with software-managed local memories.

**Research Objective**

The goal is to design and construct a concise and abstract programming interface, and implement the runtime to support it. This programming environment possesses the following traits:

i) Performance: minimizing the overhead associated with software prefetching and caching on software-managed memories.

ii) Fine-tuning: supports the tuning of performance-sensitive parameters such as block
sizes and shapes.

iii) Versatility: not only applies as a plug-in to provide runtime module supporting source-to-source compilers for high-level programming models, but also usable as a stand-alone programming framework for application fine-tuning.

iv) Productivity: providing a concise API that results in high productivity without sacrificing performance.

1.1.2 Handling Applications of Irregular Accesses and Non-uniform Workload

In this case study, we use a contagion simulation, EPISIMDEMICS, as a representative example of data-intensive applications with irregular accesses and non-uniform workload. EPISIMDEMICS simulates epidemic diffusion in extremely large and realistic social contact networks. It captures the dynamics among co-evolving entities. With rising input sizes and strict deadlines for simulation results, e.g., for real-time planning during the outbreak of an epidemic, we must push the boundaries of scalability for this application area. Our study relies on large-scale distributed memory platforms. Such an application typically involves large-scale, irregular graph processing, and is known for the difficulty of scaling.

Research Objective

The goal is to identify which aspect of the application impacts the scalability and then to devise effective solutions.

i) Identify the scalability bottlenecks in the type of applications.

ii) Determine if an identified bottleneck is application-specific, and can be modeled.

iii) Develop application-specific solutions if necessary to avoid performance bottlenecks.

iv) Utilize any existing solution available in the underlying runtime system if applicable to improve the scalability.
1.2 Research Accomplishments

We summarize the main contributions of this dissertation as follows.

1.2.1 Handling Applications of Regular Accesses and Uniform Workload

Research contributions

The contribution of this study, as we further discuss in Chapter 2 and 3, is that we provide a parallel programming framework for applications with regular nested parallel loops to support application-specific optimization of data prefetching and caching on multi-core processors with software-managed memories, which:

i) incurs lower overhead on the critical path in generating transfers than the existing state-of-the-art framework does, by blocking and strip-mining [19].

ii) enables to take advantage of temporal locality by variable-depth buffering [20] in accessing partially overlapping data blocks, as found in common stencil codes.

iii) provides comparable programmability to high-level programming models [21].

iv) is based on a pure runtime approach where the determination of performance-sensitive parameters, such as the dimension of a data block, can be deferred until execution unlike language-based frameworks [22].

v) can serve as an optimization module to parallelizing source-to-source compilers [23].

vi) can be used as a stand-alone programming framework for fine-tuning performance [20].

vii) supports multi-level loop decomposition to improve load balancing in nested loops by multi-level tiling followed by multi-dimensional loop partitioning [24].

1.2.2 Handling Applications of Irregular Accesses and Non-uniform Workload

Research contributions

The contribution of this study, as we further discuss in Chapter 4 and 5, as follows:

i) We analyze the challenges in scaling a state-of-the-art contagion simulation code, Episimdemics, and connect them to the heavy-tailed properties of the input graph.

ii) We introduce a workload model that allows state-of-the-art graph partitioners to use custom, application-specific load balancing constraints as well as the proposed application-
specific data partitioning.

iii) We implement and evaluate a data distribution method based on application-specific information of data locality, i.e., zipcode in population data.

iv) We propose a technique to preprocess the input graph to split heavy nodes, which enables graph partitioners to produce a more balanced workload distribution.

v) We implement and evaluate a series of communication optimizations for irregular graph-processing applications, including message aggregation.

vi) We demonstrate unprecedented strong scaling of a contagion simulation on over 352K cores of Blue Waters (Cray XE6) at National Center for Supercomputing Applications (NCSA), one of the largest HPC systems in the world, as well as 128K cores of Vulcan (IBM Blue Gene/Q) at Lawrence Livermore National Laboratory (LLNL).

1.3 Dissertation Organization

We discuss how we improve the performance and scalability of two different types of data-intensive applications by optimizing data accesses. In the next two chapters, we investigate the former case on a platform with software-managed memories, which offers a great control of data locality as well as the great complexity of programming. Then, in the following two chapters we discuss the other case. Chapter 2 introduces a programming framework, STRIDER, for software-managed memories to take advantage of additional control exposed to programmers with a reduced effort. We describe the design and implementation of STRIDER on the Cell Broadband Engine processor. Then, we demonstrate how the performance of applications with regular access pattern and uniform workload benefits from the proposed framework. Chapter 3 further discusses the programmability of the framework by presenting an example of coding in STRIDER API and by comparing against one of the state-of-the-art language-based framework which allows the expression of strided references via the language abstraction. In Chapter 4, we perform the analysis of scalability bottleneck from data. Then, we present application-specific load models as well as the methods to decompose and distribute data. In Chapter 5, we further investigate remaining bottlenecks and apply solutions available for a broader range of applications than a particular application offered in an existing parallel programming framework, CHARM++. Finally, we present the scaling performance of the application. Chapter 6 presents the conclusion of this dissertation.
Chapter 2

Optimizing Strided Data Accesses for Nested Parallel Loops

Many multi-core processors for high-performance and embedded systems use software-managed on-chip memories, also referred to as scratchpad memories or local stores [14–18]. Software-managed memories enable application-specific optimization of data caching and prefetching by programmers, compilers, runtime systems, or some combination of the three. Automatic management of software-managed memories in a parallel programming framework improves programmability and simplifies code maintenance. However, the added value of automatic management of software-managed memories depends critically on the extent to which the compiler and the runtime system can optimize performance.

While prior work has contributed to improving programmability of multi-core processors with software-managed memories, current solutions still fall short in terms of performance, even in common cases. Software prefetching and caching of data accessed with unit and non-unit strides raises challenges. For example, generating the transfers adds substantial software latency on the critical path while it is nontrivial to distribute and schedule data transfers between cores to maximize parallelism and minimize non-overlapped memory latency. Furthermore, exploiting temporal reuse across partially overlapping blocks of data, while natural for hardware-managed caches, is a difficult task for software-managed local memories.

In this chapter, we present STRIDER, a runtime framework for transparent optimization of bundles of strided and non-strided memory accesses on multi-core processors with explicitly managed memories. The framework targets loop-dominated programs with regular nested parallel loops, where loop bounds are affine functions of loop indices. STRIDER auto-optimizes the generation and scheduling of explicit data transfers in such loops. In particular, STRIDER contributes to
(a) improving temporal locality  (b) optimizing the critical path of scheduling data transfers for array accesses with multiple, unit and non-unit strides in nested parallel loops, and (c) effectively distributing data accesses between cores. We make the following contributions to optimize for software-managed memories through STRIDER:

- **Variable-depth buffering**, a new method for grouping data transfers and exploiting temporal locality between partially overlapping data blocks. Variable-depth buffering benefits common computational patterns such as stencils and enables further optimization of data grouping so that overlapping data blocks can be fetched with predominantly contiguous data transfers instead of strided data transfers.

- A new method for characterizing and grouping strided array accesses based on the concepts of monochromatics and wavelengths, two terms that we borrow from physics. STRIDER uses monochromatics as the basis for optimizing the critical path of grouping, preparing, and scheduling explicit strided accesses of arrays. The method is based on the idea of grouping accesses with the same stride across arrays.

- A multi-level decomposition method of the iteration space of perfectly and non-perfectly nested parallel loops that does not rely on nested parallelism. The method supports loop collapsing as proposed by OpenMP [25] and enables aggregation of the working sets of multiple tasks scheduled on the same core, to minimize the latency of the associated data transfers.

We present an implementation of STRIDER on Cell. In addition to the aforementioned broader contributions, STRIDER implements the architecture-specific optimizations on Cell, such as NUMA-aware data placement for even distribution of data accesses between processor nodes with localized DRAM and block shaping for optimizing direct memory access (DMA) transfers.

Our experimental analysis shows that STRIDER performs competitively to painstakingly hand-optimized code and markedly better (up to $3.6\times$) than contemporary language frameworks for the Cell processor in non-trivial parallel applications and important application kernels.

The rest of the chapter is organized as follows. Section 2.1 provides an overview of STRIDER. Section 2.2 discusses the design and implementation of STRIDER. We present our experimental analysis of STRIDER in Section 2.3. Section 2.4 summarizes the chapter. Chapter 3 further discusses the programming abstraction provided by STRIDER. In addition, Appendix A provides the background information about the Cell processor, and the programming challenges related to the locality management in such an architecture.
2.1 Overview of Strider

STRIDER consists of a runtime system, an API and external code modules provided by the compiler or the programmer. We illustrate the components of STRIDER in Figure 2.1. White rectangles correspond to runtime components of STRIDER. Shaded rectangles represent external code modules conforming to STRIDER specifications and using the Strider API. Arrows connecting modules represent the flow of module dependencies.

External modules define parallel tasks and the computation kernels encapsulated in tasks via the provided API. The API provides capabilities to offload tasks from control-efficient to compute-efficient cores on heterogeneous multi-core architectures. The STRIDER prototype on the Cell processor, to which we will refer from this point on, offloads kernel tasks from the PowerPC core (PPE) of the Cell to the Synergistic Processing Elements (SPEs), which provide the bulk of the computational power of the processor [14]. SPEs have software-managed local stores and STRIDER
automates their management.

The STRIDER runtime system automatically generates explicit data transfers in the form of DMA lists [14]. The block access list generator is the component that integrates STRIDER’s novel optimizations. It generates DMA lists based on an explicit description of the iteration space of a nested loop and the loop’s data access pattern. SPE offloaders supply this information via the STRIDER API. The DMA handler provides abstract interfaces to the DMA primitives of the IBM Cell SDK [26]. The handler issues DMA calls based on metadata generated by the block access list generator as well as metadata provided by the buffer manager, including the address of buffers in local stores and DMA tags for data in the buffers.

From the perspective of a compiler or a programmer, calling the buffer manager in SPE offloaders with a description of the iteration space and the layout of data arrays is sufficient to run the code with automated data caching and prefetching performed by STRIDER. First, a programmer or a compiler identifies parallel loops to offload and data arrays accessed in the loops, and creates data structures to represent loops and arrays using the STRIDER API and static or dynamic application parameters. The PPE offloader handles signaling and parameter passing to SPEs by directly writing in or reading from the effective addresses of local store variables shared between the PPE and SPEs. The PPE relies on efficient on-chip core-to-core communication to perform these operations. In PPE and SPE offloaders, the programmer or the compiler describes whether to deliver each parameter value from PPE to SPEs before a loop starts, let SPEs calculate parameter values using information available at runtime, or use a hard-coded value. The programmer or compiler relies on the STRIDER’s automatic decomposition API for parallel execution of loops. Computational kernels must be modified to interface with the parameter passing method of STRIDER’s runtime in advance and then plugged into the framework via the user kernel interface at runtime. The buffer manager allocates data buffers in local stores, schedules DMA transfers, and invokes the externally provided kernels.

STRIDER implements new optimizations for minimizing the cost of generating DMA lists, grouping data in order to execute more efficient DMA transfers of contiguous data from memory, and reusing data across tasks from shared buffers in local memories to improve temporal locality. The framework includes custom debugging and profiling tools used in the code development and production phases.
2.2 Design of Strider

We present the design of STRIDER and the optimizations that STRIDER uses to improve caching and prefetching for strided data access patterns on processors with explicitly managed memories. Section 2.2.1 introduces the architecture-independent design components of STRIDER, while Section 2.2.2 discusses architecture-specific optimizations on the Cell processor.

2.2.1 Multi-stride Data Access Optimization

We present STRIDER’s essential data structures and follow up with a discussion of monochromatics, decomposition of data accesses between cores and variable-depth buffering.

Strider Data Structures

The input to STRIDER is a high-level description of the iteration space of a loop nest and the data accesses to arrays included in this nest. From this description, STRIDER identifies data access patterns and schedules explicit data accesses using DMA lists and direct buffering of data in local stores.

STRIDER uses iterators to map loop iterations to offsets in memory for all array elements accessed in the loop. STRIDER handles both perfectly and non-perfectly nested loops where the loop bounds are affine functions of the loop indices. The runtime system determines a loop iteration space \( I \) from the loop bounds and the loop step using linear inequalities [27]. Formally, STRIDER represents iteration spaces with a data structure \( \text{IterSpace} \), \( I = \vec{i} \), where \( \vec{i} \) is a vector of index variables, i.e. \( \vec{i} = [i_1, i_2, \ldots, i_n]^T \), and each index variable \( i_k \) takes values between a lower bound \( l_k \) and an upper bound \( u_k \). The STRIDER iterator maps each access to an array element in a loop nest to a unique offset in memory. For each access to an array element \( A[i_n][\ldots[i_2][i_1] \) of an array (in the C array naming convention) with bounds \( d_n \times \ldots d_2 \times d_1 \), STRIDER calculates the offset as \( ((\ldots((i_n \times d_{n-1} + i_{n-1}) \times d_{n-2} + i_{n-2}) \times \ldots + i_3) \times d_2 + i_2) \times d_1 + i_1 \) [19]. STRIDER represents arrays through array descriptors specifying the array rank, dimensions, and base address. The runtime system constructs descriptors for arrays upon allocation, or a programmer may manually create them. STRIDER handles both statically allocated and dynamically allocated arrays. Compiler frameworks typically estimate the code size and statically defined data structure sizes to check if a local store has enough space for both. Programmers are to statically allocate each data structure with the maximum size required during execution for compilers’ provision to work. With STRIDER, on the other hand, it is the runtime system’s responsibility to guarantee that the local store is not overflowed.
Calculating offsets of array elements is expensive on processor cores with pipelines that are not optimized for scalar instruction execution. On the Cell SPE, an integer multiplication costs 7 cycles, an integer addition 2 cycles, and a load 6 cycles. STRIDER reduces the cost of calculating offsets by performing blocking and strip-mining [19]. The runtime system fetches data to local stores in blocks of contiguous or non-contiguous array elements, after labeling each block with the strides between adjacent elements in each of the block’s dimensions. The runtime system then computes the offsets of elements in each block relatively to the block’s base address by iteratively adding strides, thus avoiding expensive multiplication, division, and modulo operations. The runtime system further optimizes offset calculation by vectorizing it across array elements.

**Monochromatics**

On processors with explicitly managed memory hierarchies, data transfers are most efficient if they transfer large blocks of contiguous data. Data transfers of blocks of non-contiguous data suffer from overhead for the preparation of transfer lists, which is dominated by the calculation of offsets of non-contiguous array elements in memory. STRIDER introduces monochromatics and waves, to perform grouping and aggregation of data with the same access pattern and maximize the efficiency of data transfers.

A monochromatic is a data structure that encapsulates one or more distinct arrays with identical access patterns, rank, and dimensions. STRIDER handles these arrays in a bundle. The access pattern of a monochromatic is formulated as a periodic function, a wave, of the innermost loop index. The monochromatic’s period is a constant positive stride between adjacent array elements along the array dimension accessed by the innermost loop index. A positive stride \( s \) and a negative stride \(-s\) are both represented by the same period \(|s|\). Waves with the same period accessing arrays of the same rank and dimension form the monochromatic. A monochromatic further combines the accesses on the arrays of different element sizes together as well as the accesses with a constant displacement added to the loop induction variable. The key optimization enabled by a monochromatic is to calculate offsets of array elements once, for all arrays in the monochromatic, and reuse the same DMA list. This optimization substantially reduces the critical path of data transfer preparation and scheduling.

Besides optimization of offset calculation, monochromatics compress the iteration space representation for multiple arrays with identical access patterns, since only the base effective addresses of different arrays needs to be differentiated in the representation. This optimization saves space in local memories. In addition, when buffers are large enough to hold data blocks for multiple kernel invocations, automatic aggregation of data blocks of a wave reduces the overhead of DMA transfer preparation and completion notification. Furthermore, contiguous blocks are merged for more
Table 2.1: An example of the reference depth and the variable depth buffering with $c[i] = f(a[i - 2], a[i - 1], a[i], a[i + 1], a[i + 2], b[i - 1], b[i], b[i + 1])$.

<table>
<thead>
<tr>
<th>Array</th>
<th>Reference depth</th>
<th>Access type</th>
<th>Buffering depth</th>
<th>Buffering start</th>
</tr>
</thead>
<tbody>
<tr>
<td>$a[]$</td>
<td>4</td>
<td>in</td>
<td>6</td>
<td>0</td>
</tr>
<tr>
<td>$b[]$</td>
<td>2</td>
<td>in</td>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>$c[]$</td>
<td>0</td>
<td>out</td>
<td>2</td>
<td>4</td>
</tr>
</tbody>
</table>

efficient DMA except when data spans the boundaries of an array dimension.

A single monochromatic alone is considered as a fundamental monochromatic (or simply fundamental) in which accesses are associated with each iteration. A group of monochromatics that differ in their iteration spaces may need to be handled together when computation deals with array data from different monochromatics. For example, there can be streams of signals from multiple sensors, which are sampled at different frequencies and are fused for computation. The sampling rate of a signal is not likely to be a multiple of that of an unrelated signal. When a loop represents such a computation, it is not efficient as it relies on conditionals to determine whether to process each stream at each iteration. Thus, accessing arrays from different monochromatics requires special coordination. Harmonic monochromatics (or simply harmonics) is a group of monochromatics where the accesses of different monochromatics are coordinated over a virtual iteration space.

STRIDER analyzes monochromatics to efficiently generate DMA lists for transferring in (read-only), out (write-only) and inout (read-write) data to and from local memories. Monochromatics are also the basis for implementing dynamic (runtime) dependence analysis between explicit data accesses and dynamic scheduling of dependent accesses in STRIDER. However, dynamic dependence analysis in STRIDER is beyond the scope of this study.

Variable-Depth Buffering

Variable-depth buffering improves temporal locality by removing redundant data transfers from external memory to local memories and vice versa. Similarly to static multi-buffering (such as the double- or triple-buffering used commonly on Cell to prefetch data to local stores), variable-depth buffering allocates persistent buffers in local stores and fills some of these buffers with data blocks that will be accessed in future kernel invocations to mask memory latency. Buffers are allocated on a per-array basis, with simultaneous translation of array indices to offsets in local stores. The difference to static multi-buffering is that the number of persistent buffers is variable and defined by the references to each array in one iteration.
Figure 2.2: Variable depth buffering for the example of Table 2.1. P, S, E, BD, and $r_{\text{max}}$ stand for prologue, steady state, epilogue, buffering depth, and maximum reference depth respectively. A rectangular block represents a buffering step for an array. Figure 2.4 shows this step in more detail.

Figure 2.3: Buffering rotation types for an array: double-buffering (left), triple-buffering (center), and $d$-depth buffering (right).

Figure 2.2 illustrates variable-depth buffering. We define the reference depth $r$ of an array in a loop iteration as the number of distinct references to array elements with indices that vary by a constant in a chosen dimension, minus one. For example, the reference depth for $a[ ]$ is 2 in the following statements: $b[i] = f(a[i - 2], a[i - 1], a[i]), b[i] = f(a[i - 1], a[i], a[i + 1]),$ and $b[i] = f(a[i], a[i + 1], a[i + 2])$. Variable-depth buffering uses $(r + 2)$-depth buffering for prefetching both read-only and write-only arrays and $(r + 3)$-depth buffering for prefetching and writing back read-write arrays with reference depth $r$.

More specifically, variable-depth buffering uses a set of rotating buffers for each array accessed in a loop (Figure 2.3). The runtime system rotates buffers between data prefetching, computation and data write-back (Figure 2.4). Innermost loops are strip-mined and scheduled in three stages, a prologue, a steady state, and an epilogue. The prologue spans over the first $r_{\text{max}} + 1$ iterations of the innermost loop, where $r_{\text{max}}$ is the maximum reference depth of the arrays accessed in the loop. During the prologue, the runtime system preloads $r + 1$ buffers for each array with reference depth $r$ starting at every $(r_{\text{max}} - r)$-th iteration of the innermost loop, each time the runtime executes the
a. initiate loading next read-type buffer  
b. wait for loading current read-type buffer to complete  
c. wait until current write-type buffer becomes free  
d. invoke kernel  
e. initiate storing results in current write-type buffer

Figure 2.4: A buffering step in detail.

innermost loop.

In steady state, an externally provided kernel computes using data from the \( r + 1 \) preloaded buffers of each array and writes back data from write buffers to memory, while issuing DMA transfers to preload the \( (r + 2) \)-th buffer of each array, with the goal of overlapping computation with memory latency. STRIDER merges data blocks of multiple reference depths from the same array in one buffer, to avoid data redundancy in overlapping buffers and minimize buffer management overhead. The epilogue issues the remaining write-backs to memory.

Additionally, programmers may apply variable-depth buffering to improve overlapping of communication and computation even with a reference depth of 0. If the computation time is not long enough to completely hide the latency, variable-depth buffering can help initiate the transfers at an earlier iteration rather than just one ahead, costing extra buffer space.

Managing Multiple Accesses

DMA transfers rely on tags for allowing the runtime system to poll their status and detect their completion. Unfortunately, a Cell processor supports only a limited number of tags, 32 total, to simultaneously keep track of all outstanding DMAs in multiple accesses. Thus, we need a method to carefully manage the scarce resource in order to avoid a potential tag shortage. Tag management can be complex with variable-depth buffering as there are a larger number of accesses to keep track of across multiple iterations. STRIDER implements a new tag distribution and management strategy to increase the number of arrays that can be handled simultaneously. If the buffering depth of an array is \( d \), the runtime system requires \( d \) tags for DMAs from and to the array. For optimal performance, the runtime system first attempts to assign a unique tag to each buffer of each array. If this is not possible due to the limited number of available tags, the runtime system splits the available tags in proportion to the number of tags required by the three different types of data accesses, namely \( \text{in/out/inout} \). The runtime system also groups arrays with the same access type and the same buffering depth and reuses the set of tags assigned to an array for other arrays in the same group. The scheduling of DMA transfers for arrays in the same group is identical and
is also reused by the runtime system. In the case of SEQUOIA [21, 22, 28], a unique tag is assigned per access. Consequently, the number of arrays that can be accessed simultaneously is the most limited. On the other hand, in case of Cellgen [23], two tags are shared by all in arrays, and two tags are shared by all out arrays, and three tags are shared by all inout arrays, insensitive to the order of array being accessed. This limits the flexibility in scheduling as well as the opportunity of communication/computation overlapping.

**Iteration Space Partitioning and Aggregate Task Blocking**

STRIDER implements automatic decomposition of the iteration space of parallel loops using multi-level tiling, followed by multi-dimensional loop partitioning [24]. The API also provides an optional decomposition for explicit control of loop scheduling. The default loop partitioning method in STRIDER divides the entire iteration space of a nested loop across all permutable loop dimensions, leading to smaller tasks, better load balancing and better computation-to-memory latency ratio than decompositions of lower dimensionality [24]. This design choice further increases the opportunities of variable-depth buffering to improve temporal reuse of data in partially overlapping blocks that belong to the working sets of different chunks of loop iterations (tasks) scheduled on the same core.

In STRIDER, programmers divide nested loop levels into three hierarchical layers: offloading layer, decomposition layer, and streaming layer. The iteration space in an offloading layer iterates on PPE sequentially offloading a task of a subspace of the entire iteration space one at a time. This is useful if the reduction of intermediate results from the current subspace is needed to process the next subspace. Programmers decompose the offloaded subspace and distribute the space among SPEs as evenly as possible. Programmers can choose to decompose the entire subspace. Otherwise, they can further split the subspace into an outer layer and an inner layer, and only decompose at the outer layer. In the former case, the entire offloaded subspace is the decomposition layer. In the latter case, the outer layer is the decomposition layer. Finally, each SPE is assigned a subspace of the offloaded subspace and this layer is the streaming layer. STRIDER runtime processes the streaming layer on each SPE.

Variable-depth buffering may incur fragmentation of buffer space in local stores, due to the transfer of potentially many disjointed small blocks of array elements. Fragmentation becomes more pronounced if the number of innermost loop iterations scheduled on an SPE with each kernel invocation (i.e. the task size in conventional parallel programming nomenclature) is small. STRIDER provides an API for performing aggregation of data sets used in multiple tasks in blocks, which further reduces the latency of data transfers to and from local stores. Through this API, STRIDER reduces the rank of array blocks transferred with DMAs, effectively creating larger blocks of contiguous data and lower dimensionality. These are in turn transferable with fewer and more
efficient DMAs than the blocks generated originally by iteration space partitioning. The runtime system performs aggregation by fusing loop levels in the partition of the iteration space assigned to an SPE and re-blocking the fused loops, under the constraint that the aggregated working sets of tasks do not exceed the available space in local memory. This fusion step enables prefetching of working sets of multiple tasks each time the variable-depth buffering engine requests data from memory, provided that tasks are scheduled for execution on the same core.

The Parameter Stack of a User Kernel

A user kernel is written independently of the runtime framework and is dynamically plugged into it later at the runtime. A kernel exchanges parameters with the framework and the user provided modules via a parameter stack. The parameter stack is divided into two parts: the internal stack and the user stack. The runtime framework uses the internal stack to pass the internal low-level information to a kernel, such as the current iteration indices, padding size, alignment size, block size, the number of blocks in the buffer, and the number of waves. User parameters are passed to a kernel via \texttt{pop()}, \texttt{push()}, \texttt{peek()} and \texttt{poke()} operations. \texttt{peek()} reads a value from the user stack without removing any parameter, while \texttt{pop()} removes the read value out of the stack. \texttt{poke()} writes a value into the user stack without adding any parameter, while \texttt{push()} adds one.

2.2.2 Cell-Specific Optimizations

Architecture-specific optimizations of STRIDER on Cell B.E. include NUMA support and block shaping.

Allocating Memories for NUMA

STRIDER implements NUMA aware memory allocation to bind pages to specific nodes and avoids memory accesses crossing the interconnection network between Cell processors on blades. A Cell blade has two nodes with one processor each. When a program starts, the runtime system binds each thread to an SPE and assigns the thread a rank (ID) equals to the order of the physical ID of the thread-bound SPE. This ordering performs best with common data access and exchange patterns on SPEs [29].

The runtime system maintains a thread split ratio, which is the ratio of the number of threads with affinity to the SPEs of each node and a page split ratio, which is the ratio of the number of pages to be allocated from each node, out of the total number of pages in the system. The default behavior is to distribute pages between nodes according to the thread split ratio, however the
compiler/programmer has the option to modify the split ratio on-demand. The NUMA-aware page binding technique in STRIDER splits the address space available for program data in $N$ segments, where $N$ is the number of physical nodes. The runtime system logically binds segments to nodes, according to the order in which threads request allocations, the ranks of threads, and their mappings to nodes. This strategy is similar to the first-touch NUMA page allocation scheme [30]. STRIDER supports also an optional page interleaving pattern which assigns pages in a round-robin fashion between nodes. Page interleaving often benefits applications that exhibit phase changes in their DRAM access patterns, by spreading out memory traffic more evenly. As an additional optimization, the runtime system may align the memory space allocated to each node so that the boundary between data allocated by different nodes is aligned to the page size.

**Aligning Memory Accesses**

Multi-core processors with explicitly managed memories expose alignment constraints of memory access to programmers. This further complicates strided accesses. On a Cell processor, alignment is not only required for issuing DMA commands to the memory subsystem but also for applying the SIMD operations [26,31,32]. The Cell DMA engine requires 16-byte alignment to transfer data packets of size 16 bytes or larger. The programmer can use the STRIDER API for allocating a dynamic array to avoid potential violation of the DMA constraint at runtime. STRIDER aligns the innermost dimension of a multi-dimensional array to a 16-byte boundary by setting the start address of each array to a multiple of 16 bytes and, if necessary, pads extra bytes at the end of the dimension to ensure that the entire data size in the dimension is a multiple of 16 bytes. This ensures that the start of the innermost dimension always begins from an address that is a multiple of 16 bytes.

While STRIDER hides the details of padding and alignment for dynamically allocated arrays, it does not do so for statically defined arrays. STRIDER, however, provides an API for wrapping an individual scalar variable with necessary padding to make the size of a structure containing each scalar at a multiple of 16, which can be used to construct an array. This may be the simplest method to avoid the complexity associated with alignment requirements. However, it is not efficient in terms of memory space used and performance achieved, especially with the small native types such as char, int, and double. A better approach is the aggregation of blocks as we discuss shortly. Alternatively, a programmer can define statically allocated arrays using an alignment type attribute for the native compiler and necessary padding. For example, to statically define an array $A$ as in Table 3.2, a programmer can add the alignment attribute to the array definition as $\text{double } A[\text{nx}][\text{ny}][\text{nz}][5+\text{padding(double,5)}] \_\text{align}$. It is also worth noting that there exists techniques for aligning data in vector registers for SIMD operations [32].
**Shaping Data Blocks**

In STRIDER, the shape of a block is represented separately from the iteration space, thus allowing the programmer to choose among various shapes for application-specific optimizations including vectorization. The programmer decides the shape of the blocks first, and then determines how to represent the loop of the data arrays as the iteration space of blocks. The programmer can also write the code to choose between different shapes at runtime.

Software using STRIDER selects block sizes and shapes through a *block descriptor* of STRIDER. The block descriptor can represent $n$-dimensional hypercubes as well as amorphous block shapes. For the latter, the runtime system uses a DMA template list to describe the shape of amorphous blocks, and STRIDER adjusts the offsets during execution based on the current iterator. In our experimental evaluation of STRIDER (Section 2.3), we use an exhaustive search of optimal block shapes and sizes to isolate the impact of these two parameters on performance.

In code generation frameworks, the shape of blocks is often limited by design for the convenience of static analysis. For example, SEQUOIA [21] requires a block to have a fixed shape at compile time as a hypercube up to six-dimensional.

STRIDER supports coarse-grain aggregation and blocking for user-defined blocks. STRIDER allows programmers to specify up to how many blocks to buffer together for efficient DMA transfers. STRIDER finds the closest parameter for block aggregation preserving DMA alignment constraints. This can also be done automatically by the framework under certain conditions. If the shape of a block is one-dimensional and the size of a block is the same as the stride between the addresses of adjacent blocks, STRIDER treats the blocks in a buffer as a whole for efficient DMA. For example, with the array $A$ accessed in the loop shown in Figure 3.1, the innermost loop iterates with the unit stride. In this case, the runtime can group the successive accesses along dimension 1. STRIDER divides the contiguous data chunk into three parts – head, body, and tail. The body begins at an aligned address and ends at another aligned address. The head precedes the body, then the tail follows. The size of a head and tail is limited to 16 bytes. In this example, the head is empty and the tail is eight-byte long. Then, STRIDER inspects whether a single DMA list can transfer the three parts without altering the data layout in the buffer. If so, it generates a list entry for each part of the non-zero size. Otherwise, it repeatedly generates a DMA for the maximal portion that satisfies the constraints until covering all the remaining data.
2.3 Experimental Analysis

We first present our experimental setup, including our platform and applications. We proceed with the presentation of our experimental results and their analysis.

2.3.1 Experimental Platform

We present results from experiments on an IBM QS20 Cell blade running Linux Fedora Core 7. The kernel version of Linux operating system is 2.6.22-5 compiled for the 64-bit PowerPC architecture. The blade has two Cell processors with a NUMA organization. The processors reside on two nodes with 512 MB of off-chip XDRAM each. We use the time base registers of the PPU and SPU cores to collect breakdowns of execution times. Specifically, STRIDER uses the mftb() macro \(^1\) for PPE timing and the spu-timer library 3.0 for SPE timing. We implement STRIDER with the IBM Cell SDK 3.0 and compile it with ppu-gcc 4.1.1, and spu-gcc 4.1.1 [26].

2.3.2 Benchmarks

Our set of benchmarks includes two kernels, copy and transpose, and three applications, Jacobi, Fixedgrid and Parallel Bayesian Phylogenetic Inference (PBPI). In addition, we use microbenchmarks to evaluate the performance of the synchronization and parameter passing approaches taken by the two runtime systems. The copy and transpose kernels evaluate the performance of contiguous and non-contiguous data transfers between memory regions in two implementation alternatives, one using STRIDER as an OpenMP back-end and one using SEQUOIA, a state-of-the-art programming language with explicit management of the memory hierarchy by the programmer to control locality (see Section 3.3) [21, 22]. Both kernels are common in scientific applications. The kernels have the minimum operations-per-byte ratio and stress bandwidth utilization and DMA performance. They are sensitive to the overhead of preparation and scheduling of DMA transfers, to DMA block sizes and shapes, and to data placement in DRAM. Furthermore, the shape of the input and output data regions stresses the load balancing capabilities of the runtime system.

Jacobi implements a five-point stencil code [33] and stresses the capability of the runtime system to overlap computation with memory latency and exploit temporal locality.

PBPI uses Markov-chain Monte Carlo methods to compute alternative phylogenetic trees, using the maximum likelihood criterion [34]. PBPI stresses the performance of contiguous data transfers with phase changes in the DRAM locations accessed between NUMA nodes. It is an application

\(^1\)http://www.ibm.com/developerworks/power/library/pa-timebase/
with a fine granularity of parallelism, and its performance is sensitive to the size and frequency of
DMAs between the SPE and main memory. Especially, overlapping DMA latency and computation
is important for its performance. PBPI exposes chain-level parallelism and sub-sequence-level
parallelism. The original implementation of PBPI in MPI exploits both, and it is also ported to Cell
with further utilization of data parallelism present in subsequence-level computation by offloading
the computationally expensive likelihood estimation functions to SPEs and applying Cell-specific
optimizations on the offloaded code such as double buffering and vectorization [11, 34, 35]. In
our experimental setup, PBPI is configured to run with a single chain. There are three major data
parallel loops we offload to accelerators in PBPI. PBPI offloads these loops total 324,071 times to
compute the data set chosen for our study, 107 taxa with 19,989 nucleotides for a tree. Especially,
it offloads one of the loop 88% times of the total, loading 1.2 MB of data from the off-chip main
memory to local stores to compute and writing results of 0.6 MB back to the main memory each
time. Each of the other loops accounts for roughly the half of the remaining offloading. One loads
1.8 MB to local stores and writes results of 0.6 MB back to the main memory, and the other loads 0.6
MB to compute a result of 8 bytes.

Fixedgrid is the core of a chemical kinetics kernel that operates over a fixed domain grid. The kernel
models the evolution of chemical species and is an important component in atmospheric modeling
systems [36, 37]. Fixedgrid stresses strided data accesses by performing data discretization along
multiple array dimensions. The original implementation of Fixedgrid on Cell B.E. [36] offloads the
computation of mass flux and chemical concentration by a third order upwind-biased advection
discretization and second order diffusion discretization on two dimensional grid for each time step.
Fixedgrid relies on a double precision type array of size $w \times h$ to represent each of the latitudinal wind
field, longitudinal wind field, and horizontal diffusion tensor, where $w$ and $h$ is the width and the
height of domain respectively. In addition, it keeps a separate array of size $N \times w \times h$ for concentration
of $N$ chemical species. Fixedgrid allows selectively disabling a chemical or transport processe
to observe isolated effect on concentrations. In our experimental setup, Fixedgrid calculates the
concentration of ozone ($O_3$) on a 600$\times$600 domain, roughly loading 24.7 MB of data to local stores
and writing 8.24 MB to off-chip main memory at each step.

In the experiments with applications, we compare the performance of STRIDER against hand-
optimized implementations of the applications using the IBM Cell SDK and implementations in
SEQUOIA. We obtain the SDK implementations directly from application domain experts who are
not involved in the development of STRIDER and have already invested significant effort in paral-
lelization and optimization of their applications on Cell. These implementations are documented
elsewhere [11, 37, 38]. We introduce additional optimizations to the SDK implementations and
present them in detail wherever applicable. We also optimize the SEQUOIA implementations to the
extent of our understanding of the language, compiler, and runtime environment.
To achieve symmetry between different implementations, we use the same SIMD kernel in all implementations. Nevertheless, kernel performance may still differ due to differences in iteration space partitioning, parameter passing between the PPE and SPE kernels, and DMA data marshaling. In addition, we eliminate an additional copy that SEQUOIA uses to create a replica of the original application arrays on the PPE. SEQUOIA performs this copy because the data layout used by SEQUOIA runtime may differ from that in base applications. This capability is non-essential for our experiments but adds overhead non-existent in the implementations using the other programming frameworks as we unify the layout.

### 2.3.3 Experimental Results – Kernels

**Copy and Transpose by Various Blocks**

We evaluate the performance of STRIDER in the copy and transpose kernels. We use the optimized implementations in SEQUOIA as our baseline. In this targeted experiments, we evaluate the impact of block shaping and load balancing in the runtime systems. The kernels copy and transpose 2-D arrays of three sizes, 600×600, 1200×1200 and 1800×1800. We use eight block shapes of the same volume (1,200 elements): 600×2, 300×4, 100×12, 40×30, 30×40, 12×100, 4×300, and 2×600. The inflicted buffer space is 1,200 double elements.

With STRIDER, we use the decomposition strategies to divide the entire iteration space offloaded to SPEs (labeled as -all) as well as the one to divide the outer layer (labeled as -outer) as discussed in Section 2.2.1. The former decomposes the entire range of iterations across all loop levels, while the latter decomposes only at the outermost loop in this example. In SEQUOIA, the parallelization of one or more loop levels dictates the block shape and the data decomposition method. The programmer defines this decomposition statically at compile time. Therefore, we recompile the SEQUOIA implementation for each data point to statically set to the optimal iteration decomposition scheme. On the other hand, in STRIDER, the runtime system performs iteration space partitioning and parallelization and allows recalibration on the fly for dynamic optimization.

For a given block shape of size \(a \times b\), the larger the \(a\), the longer the DMA list and the longer the time it takes to prepare and handle the list. Therefore, taller (large \(a\)) and thinner (small \(b\)) blocks takes longer to transfer than shorter and thicker blocks in general. Figure 2.5(a) and Figure 2.5(b) verify this claim. In row copy, the outer loop iterates over the column direction. Thus, when the 600×2 block is used for a 600×600 array and only the outer loop is parallelized, a single SPE copies the entire array which is split in 300 blocks. On the other hand, when iterations across all loop levels are decomposed, each of 4 SPEs takes 18 blocks and each of the other 12 SPEs takes 19 blocks.
Figure 2.5: Copy and transpose 2D arrays of three different sizes 1,000 times using 16 SPEs and eight different block shapes while relying on the two decomposition methods supported by STRIDER and one by SEQUOIA.

The difference is shown with the the data points of 600-outer and 600-all for block size 600×2 in Figure 2.5(a). In column copy, the most unbalanced case occurs for block size 2×600, as shown in Figure 2.5(b). Among the eight block shapes, 30×40 is the closest to a square. Here, we provide an example of how the block shape affects the load distributed among SPEs, for copying 600×600 in rows. In the case labelled as -all, each of twelve SPEs takes 19 blocks and each of four SPEs takes 18 blocks. In the case labelled as -outer, each of twelve SPEs takes 15 blocks and each of four SPEs takes 30 blocks. In the case labelled as -sequoia, each of twelve SPEs takes 20 blocks, and each of four SPEs takes 15 blocks.

Copy and Transpose with Numa-aware Page Allocation

In both experiments shown in Figure 2.6 and Figure 2.7, we select a block size of 30×40 following an exhaustive search of the optimized block sizes and shapes for the transpose kernel as shown in Figure 2.5. With STRIDER, we use the first-touch NUMA-aware page placement scheme which optimizes performance in both cases. However, SEQUOIA does not provide any method to take
Figure 2.6: Sustained bandwidth of the copy and transpose kernels for array copy and transpose implemented in STRIDER and SEQUOIA. This shows the impact of NUMA-aware page allocation on the bandwidth achieved by strided memory access. The kernels handle 2-D double type arrays of three sizes, 600×600, 1200×1200 and 1800×1800 using 30×40 blocks.

Figure 2.7: Scalability of the add-transpose kernel with the number of array pairs. The kernel adds multiple array pairs and transposes the output using 16 SPEs. This shows the effectiveness of STRIDER’s access pattern grouping and the management of simultaneous multiple accesses. STRIDER benefits from NUMA-aware page allocation. We use arrays of size 2048×2048, and blocks of size 16×16. Thus, SEQUOIA has no disadvantage from decomposition. RC stands for the case where the sum of rows from multiple arrays is transposed into a column. CR stands for the opposite case.
advantage of NUMA configuration. In this targeted experiments, we evaluate the impact of NUMA-aware page allocation on the bandwidth achieved by strided memory access. Figure 2.6 shows that STRIDER achieves twice the bandwidth of SEQUOIA in copy and $1.7 \times$ the bandwidth of Sequoia in transpose. STRIDER performs better than SEQUOIA primarily because Strider benefits from NUMA-aware page allocation especially for non-transposed transfers, and also because SEQUOIA does not partition the entire loop iteration space across multiple levels.

Managing Multiple Transfers

Figure 2.7 presents an additional experiment, showing superior bandwidth scaling of STRIDER in a kernel where we add multiple pairs of input arrays of doubles and write the result into a transposed array. The array dimensions are $2048 \times 2048$. Both STRIDER and SEQUOIA access the arrays using the blocks of size $16 \times 16$ and 16 SPEs. Thus, in this experiment, SEQUOIA does not have the disadvantage from the decomposition. The chart plots the sustained bandwidth versus the number of input array pairs. This experiment shows the effectiveness of STRIDER’s access pattern grouping in monochromatics, which substantially reduces the overhead while synthesizing DMA lists, as discussed in Section 2.2.1. As discussed in Section 2.2.1, MFC supports 32 tags to track the status of issued DMA commands. SEQUOIA fails to complete the experiment for more than 14 pairs due to tag shortage. SEQUOIA assigns a tag to each buffer—28 for the 14 double buffers of input and 2 for the double buffer of output—and 2 tags to the runtime system. On the other hand, STRIDER reuses tags, and avoids the tag shortage.

Multi-level Decomposition

We further evaluate the impact of the iteration space partitioning strategy on load balancing in STRIDER and SEQUOIA, using a targeted experiment. We use a variant of the transpose kernel, error-transpose, which computes the mean square error between two 3-D matrices and transposes the first and the second dimension of the output matrix. We use two $10 \times 150 \times 200$ input arrays, a $10 \times 200 \times 150$ output array, and $2 \times 10 \times 20$ blocks. SEQUOIA splits each dimension in 2-, 4- and 2-ways respectively to distribute total 750 blocks to 16 cores as evenly as possible. As a result, 60, 45, 40 and 30 blocks are distributed to 6, 2, 6 and 2 cores respectively. STRIDER decomposes the iteration space across all dimensions and distributes 47 blocks to 14 cores and 46 blocks to 2 cores. Figure 2.8 shows the difference between the maximum time and the minimum time spent in the computational kernel of the error-transpose. The load imbalance in SEQUOIA is $36 \times$ higher than that in STRIDER as measured by the difference between the maximum and the minimum execution time among SPEs. STRIDER achieves better balance than SEQUOIA does by the multi-level loop
Figure 2.8: Comparison of kernel load distribution. STRIDER achieves better balance than SEQUOIA does by the multi-level loop decomposition.

decomposition as described in Section 2.2.1.

Memory Access Bandwidth

Figure 2.9 shows the maximum memory access bandwidth achieved by each framework using DMA-friendly block shapes, while copying 2048×2048 array between the local stores of SPEs and the off-chip main memory. We use blocks of size $2 \times N$ where $N$ is the width of the block ranging from 2 to 1024 array elements. The maximum bandwidth that STRIDER achieves is twice as much as the maximum bandwidth that SEQUOIA achieves and close to 40 Gbytes/s out of a theoretical peak of 51.2 Gbytes/s on the Cell blade. When we deactivate NUMA-aware page allocation in STRIDER, Strider still achieves 27% higher average bandwidth than SEQUOIA.

2.3.4 Experimental Results – Coordination between PPE and SPE

Here, we discuss further differences in the runtime systems that impact the performance, in addition to the aforementioned effectiveness in managing transfers, distributing load and supporting NUMA. Specifically, we evaluate the impact of the coordination mechanism between PPE and SPE taken by each runtime system on the performance. We use a simple task that only synchronizes with or without passing parameters between a main thread on PPE and threads on SPEs. The parameters include the addresses of arrays. However, there is no computation or access to the off-chip memory in the task.
Figure 2.9: Maximum memory access bandwidth achieved by STRIDER (top and left) and SEQUOIA (right) while copying 2048×2048 array between the local stores of SPEs and the off-chip main memory. We use the shape of blocks (2×N) that are DMA-friendly accessing mostly contiguous chunks of memory but with one stride, where \( N \) is the width of the block. STRIDER without NUMA-aware page allocation (left) achieves overall 27% higher bandwidth than that of SEQUOIA (right). When STRIDER further takes advantage of NUMA, it achieves 75% of the theoretical peak (51.2 GB/s).
Synchronization

For synchronizing the start and completion of an offloaded task between a PPE and SPEs, threads rely on signaling via shared variables. To start a task, a PPE thread writes into such a variable. An SPE thread begins a task as it detects the change in the variable by polling. *Polling* here refers to repeatedly checking the status of a variable, and it is sometimes used exchangeably with *busy-waiting*. Upon completion of the task, an SPE sets a variable, of which the PPE thread is waiting for the change. **STRIDER** and **SEQUOIA** runtime implements the method using different mechanisms. The Cell B.E. offers various mechanisms for communication between a PPE and an SPE [39, 40], which includes:

i) A PPE thread can access variables/registers on local stores, signal notification registers and mailboxes in SPEs by load and store with memory-mapped effective addresses.

ii) An SPE can use the channel access instructions to read to or write from mailboxes or signal notification registers in the local MFC.

iii) An SPE thread can issue DMA commands on its MFC to transfer data between its local store and the effective address space.

iv) A PPE thread as well as an SPE thread can issue proxy DMA commands on the MFC of an SPE to transfer data between the local store of the SPE and the effective address space, where the SPE of the local store is not necessarily the same as the one that issues proxy commands.

Methods i) and ii) are suited for transferring small data, while iii) and iv) are for large data.

**SEQUOIA** relies on the mailbox interface of the IBM Cell SDK. Mailboxes support the exchange of 32-bit messages between an SPE and other processing elements including the PPE and the other SPEs. Each SPE thread can access its inbound and outbound mailbox channels, each of which is connected to a mailbox register in its own MFC. A PPE thread writes in the inbound mailbox of each SPE to initiate tasks, and reads from the outbound mailbox of each SPE to detect completion of tasks. Before a PPE thread reads from the outbound mailbox of an SPE, it needs to poll the status register in MFC of the SPE to make sure that there is a new unread message in the mailbox. Such polling as well as reading and writing by the PPE generates EIB traffic and may degrade the application performance.

**STRIDER** relies on the following mechanism for synchronization between PPE and SPE, which is also discussed in the existing studies [11, 40]. In **STRIDER**, a PPE directly writes to the effective address of the variable on the local store of each SPE to initiate a task. An SPE detects the change in the variable on its local store by polling. Upon completion of a task, an SPE modifies the variable on its local store, of which the PPE is waiting for the change. A PPE detects the change in the signal
variable by directly reading from the effective address. The PPE's polling in this case does not generate as much EIB traffic as with mailboxes due to the snoop-based cache coherency mechanism. When a PPE tries to read from an effective address for the first time, a cache miss occurs and the read goes out on the EIB. After this, the PPE polls the variable on its L2 cache. When the SPE changes the value of the variable, it invalidates the L2 cache entry and the next PPE read on the variable goes out on the EIB to get the new value from the local store.

Both STRIDER and SEQUOIA rely on on-chip communication for synchronization, which are better suited for small-sized signal variables. The hand-optimized versions of both PBPI and Jacobi rely on the same synchronization method as the one used by STRIDER. In the hand-optimized version of Fixedgrid, the communication from a PPE to SPEs and SPEs to a PPE respectively relies on proxy DMA commands and DMA commands. A PPE and SPEs poll their local copies of variables to detect an event and use a DMA transfer to write the remote variable while the other processing unit is polling it locally. Therefore, EIB traffic is generated only once for a variable update.

When an SPE completes a task, by default, there is no guarantee that all the previous storage accesses of the SPE are completed. Instead, it simply means that commands to access storage are successfully issued. Both STRIDER, SEQUOIA, and hand-optimization rely on the heavyweight sync instruction to create a barrier between accesses on shared main-storage before and after the instruction. In addition, STRIDER inserts dsync at the end of each offloaded task to ensure that all prior loads, stores, and channel accesses have completed as the dsync call completes before the end of the task. On the other hand, SEQUOIA SPE threads synchronize themselves via signal notification registers at the end of a task.

**Parameter Passing**

In this experiment, we consider array related parameters only. In SEQUOIA, a PPE thread packs parameters into data structure objects in the off-chip main memory. Each of such an object includes a set of information for an array, the four-byte long array address, as well as the runtime specific information on the array. The latter includes the offset, the pitch and the number of elements in each dimension of the array. While some of this information is not actually utilized in the current implementation, the generated code still contains it. A PPE thread passes the address of the objects to SPEs thread immediately after creating it. An SPE thread retrieves the parameters using DMA when it begins a task. The structure is a multiple of 128 bytes and aligned to a cache line.

In STRIDER, programmers define a parameter block based on their need. They can decide to include only the addresses of arrays if the dimensions and the iteration space are known in advance. In this experiment, we include the address of each array accessed in a loop, and three additional integers
Figure 2.10: Performance of synchronization and parameter passing between a PPE and SPEs. This shows the execution time for 20,000 iterations of offloading an empty task with varying numbers of parameters.

such as the total number of iterations, and the beginning and the end of iterations. We also set the parameter block as a multiple of 128 bytes. STRIDER merges the initiation of tasks with parameter passing while reusing the signal variable for the additional purpose of passing a parameter block. A parameter block is reset for every iteration.

In the hand-optimized version of Fixedgrid, a PPE allocates a data block on the off-chip main memory, including all the parameters used during execution. The address of the block is passed as a thread parameter when the thread is created. The PPE sets the parameters in the block as needed. An SPE thread begins a task by loading the block into the local store using DMA.

Observations

The execution time for 20,000 times of task synchronization with or without parameter passing is averaged over twenty measurements using various numbers of SPEs and parameters as shown in Figure 2.10. The number of SPEs used are 1, 2, 4, 8, 12, and 16. The number of parameter sets used are 0 (none), 1, 4, 8, 16, 24, and 31. In the case of SEQUOIA, we remove a few lines of the generated codes to exclude DMA calls in the task. This also helps to avoid the runtime error due to the lack of available DMA tags discussed in Section 2.3.3. Table 2.2 shows the size of data structure used to pass parameters in each case in terms of the number of (array) parameters and the passing method.
Table 2.2: The number of parameters and the size of parameters passed in the experiment shown in Figure 2.10.

Without parameter passing, the difference between STRIDER and SEQUOIA solely results from the synchronization method used. Figure 2.10 shows the superior performance of the one based on cache coherency over the one on mailboxes. The hand-optimized version of Fixedgrid relies on a proxy DMA to initiate a task and a DMA to copy a parameter block from the off-chip main memory to local stores. This method is not as efficient as that of STRIDER for synchronization. However, as the number of parameters as well as the number of writes increases, the performance of cache coherency based-method degrades. In this method, a single PPE writes $n$ parameters into the local stores of $m$ SPEs. On the other hand, in the DMA-based method, a PPE packs $n$ parameters into a structure on the off-chip main memory once per task or execution, and lets each SPE load them simultaneously to its local store using DMA. It is worth noting that the performance of this method is significantly poorer if the problem state area of SPEs is not mapped to the effective address space. For both the data points labeled as ‘Hand-coded Fixedgrid’ and ‘Sequoia’ in Figure 2.10, such a mapping is enabled.

2.3.5 Experimental Results – Applications

Figure 2.11 shows the execution time of the applications, and Figure 2.12 shows the speedup. We calculate the speedup as $\frac{T_1}{T_p}$ where $T_1$ is execution time with 1 SPE and the PPE as a control processor and $T_p$ is execution time with $p$ SPEs and the PPE as a control processor. Figure 2.13 shows breakdowns of execution times of applications on 16 SPEs, to provide further insight into the performance of alternative implementations. In all cases, we use exhaustive search to find optimal block shapes and sizes for DMA transfers given the DMA constraints discussed in Appendix A.0.4 and the limited local store size.
Jacobi

We present results from executions of Jacobi with an input of 4000×4000 array of doubles. The benchmark computes the residual for 1,000 iterations. STRIDER performs close to the SDK implementation and markedly better (over a factor of $2 \times$) than SEQUOIA as shown in Figure 2.11(a) and Figure 2.12(a). On 16 SPEs, execution time with STRIDER is 16% higher than the execution time of SDK code. Compared to SEQUOIA, STRIDER reduces memory latency by 97% (shown by the DMAwait component in Figure 2.13(a)). This difference arises because of the data buffering strategy used in each implementation. Figure 2.14 shows the key differences between the implementations. The implementations with SDK and STRIDER avoid redundant DMA transfers for the innermost loop of the 5-point stencil and reuse previously buffered data in local stores. STRIDER implements this optimization via variable-depth buffering as discussed in Section 2.2.1. SEQUOIA does not provide a mechanism to reuse previously buffered data across tasks. Instead, the SEQUOIA implementation fetches larger data blocks into local stores compared to those of the STRIDER and SDK implementations. These blocks need to include additional boundary data from neighboring blocks computed on other SPEs. Furthermore, the SEQUOIA implementation performs redundant transfers of such data.

In the SEQUOIA implementation, we experimentally choose a block size and shape (10×256) that minimizes the cost of redundant data transfers and the overhead of handling non-contiguous data at the same time. In the SDK implementation and the STRIDER implementation, we use linear blocks of size $1 \times 512$, i.e. segments of rows that do not span columns. These segments are reused from buffers in local stores whenever needed. STRIDER provides this optimization automatically whereas the SDK implementation hard-wires it in the code. Temporal locality optimization not only avoids redundant data transfers, but also minimizes the DMA overhead of transferring non-contiguous data. As a result, the non-overlapped cost of data transfers is higher with SEQUOIA (Figure 2.13(a), DMAwait component). This is also partly because of the lack of NUMA-aware page allocation scheme with SEQUOIA. Furthermore, SEQUOIA implements the reduction of the residual with barrier operations, which are expensive, especially when threads running on different Cell processors exchange data. The other two implementations handle reductions more efficiently by allowing the PPE to directly access the reduction variables on the local stores of SPEs via remote loads and stores. Such a difference contributes to Others component.

In Figure 2.13(a) we show an additional data point corresponding to the original SDK implementation of Jacobi [38], in which the performance of the kernel is significantly lower than the optimized version that we use for comparisons. The original version uses multiple DMA transfer sizes and dynamic buffer resizing in local stores, in order to meet the alignment constraints for DMA transfers. This results in the use of expensive modulo operations for calculating the addresses...
Figure 2.11: The execution time of each application implemented in three different methods

Figure 2.12: Application speedup versus execution time with one SPE.

Figure 2.13: Timing breakdown of applications on 16 SPEs. Kernel is the time spent in computational kernels. DMAwait is the DMA data transfer time that is not overlapped with computation. DMAprep is the time to prepare DMA lists. In Jacobi, Others accounts for signaling overhead between PPE and SPE and the cost of reduction operations. In PBPI, Others accounts for the cost of executing part of the workload between offloaded regions on the PPE and the cost of reduction operations. In Fixedgrid, Others accounts for the cost of executing part of the workload between offloaded regions on the PPE.
Figure 2.14: Two different implementations of Jacobi by using SEQUOIA (left) and STRIDER (right). To access 2-D data array on the off-chip main memory, each implementation manages data buffers of sub-blocks on local stores. The buffer rotates between a prefetch and a compute stage, and additionally a write-back stage if the access is for the read-write type operation. In SEQUOIA, the buffering depth, the number of stages in the rotation, is fixed. In STRIDER, the reference depth of access pattern determines the buffering depth as shown in Table 2.1. With SEQUOIA, we experimentally choose a rectangular block that minimizes the cost of redundant data transfers and the overhead of handling non-contiguous data at the same time. With STRIDER, the variable depth buffering removes redundant transfers along the innermost loop iteration. In this example, we choose the innermost loop to iterate over the non-contiguous dimension (column) of the 2-D array to remove redundant transfers along the horizontal boundaries (rows) of a block. Thus, we use linear blocks, i.e., segments of rows that do not span columns. STRIDER automatically reuse these segments from buffers in local stores whenever needed. For the same amount of data to transfer, we reduce the number of strided accesses. The temporal reuse not only avoids redundant data transfers, but also minimizes the DMA overhead of transferring non-contiguous data.

of scalar array elements while packing them in vectors. STRIDER avoids this complexity by using fixed-size, variable-depth aligned buffers in local stores and a matching alignment of the application’s arrays in memory. We introduce this optimization in the SDK implementation, which ends up outperforming STRIDER with lower overhead in preparing DMA (Figure 2.13(a)).
PBPI

PBPI executes three dominant computational kernels in parallel loops. We use three 1-D block shapes of sizes 32, 32, and 16 for the three loops in the SDK implementation, those of sizes 112, 96, and 32 in the STRIDER implementation and those of sizes 512, 512, and 256 in the SEQUOIA implementation. In principle, our exhaustive search of block sizes yields larger blocks for the runtime systems that incur higher overheads. STRIDER runs no more than 13% longer compared to the SDK implementation and up to a factor of 3.6× better than SEQUOIA, which does not scale beyond 8 SPEs. STRIDER reduces DMA wait time by 53% compared to SEQUOIA (Figure 2.13(b), DMAwait component). Most of the performance differences between the three implementations of PBPI (Figure 2.12(b)) arise due to synchronization and page allocation.

The overhead of signaling and barrier synchronization during reduction is high in SEQUOIA (Figure 2.13(b)). SEQUOIA uses the mailbox mechanism for inter-core synchronization as discussed in Section 2.3.4. This generates traffic on the ring interconnect of the Cell and degrades performance significantly, especially when it has to reach SPEs across nodes of the blade and if the number of tasks offloaded on SPEs is large. There are 324,941 times of task offloading in a PBPI run, while there are 2,592 times in Fixedgrid and 1,000 times in Jacobi. In the other two implementations of PBPI, a PPE thread directly writes to and reads from the effective address of a flag variable in the local store of each SPE to initiate tasks and detect task completion.

Both vectorization and NUMA-aware page allocation improve the performance of PBPI by large margins. The application accesses a fixed-length segment of a one-dimensional array starting from

Figure 2.15: PBPI accesses a contiguous chunk of memory from each one-dimensional array in each iteration starting from a non-deterministic address.
a non-deterministic position in each iteration as shown in Figure 2.15. This access pattern creates memory hot spots on individual nodes on the Cell blade with evenly distributed pages between nodes using first-touch (Figure 2.16). PBPI performs better with round-robin page interleaving. This provides a more significant performance benefit in conjunction with vectorization (shown by the difference between pg-intlv and noNUMA versions), because the non-overlapped DMA latency is higher in the vectorized kernel without page interleaving than in the scalar kernel without page interleaving.

The SDK implementation relies on statically allocated buffers, while STRIDER and SEQUOIA rely on dynamically allocated buffers. When we plug the original kernel into STRIDER and SEQUOIA, we observe significant kernel performance degradation. We also notice the performance difference of the original hand-optimized kernel when we replace static buffers with dynamically allocated buffers (shown as SDKorg in Figure 2.13(b)). The difference stems from the obscured opportunity of compiler optimization for read only access via aliased pointers, additional local store accesses and an inefficient vectorization scheme. The original kernel prepares vector arrays by aliasing scalar arrays to point to buffers. The kernel uses multiple aliases to the buffers to improve code readability. We fix this problem by removing pointer aliasing. This modification improves the performance of the SDK kernel with static buffers by 8.5% and the performance of the STRIDER and SEQUOIA kernels with dynamically allocated buffers by 27%. Furthermore, it reduces the difference between the SDK kernel and the STRIDER/SEQUOIA kernels from 26% to 7%.
Fixedgrid

Fixedgrid calculates the concentrations for species of interest, such as ozone, in a \( M \times N \) domain of size 600\times600 from a two-component wind vector, a horizontal diffusion tensor, and the current concentration. The computation in each time step executes five phases of row discretization, parameter updates, column discretization, parameter updates, and row discretization. The implementation on Cell offloads row discretization and column discretization on SPEs using fixed block sizes of 2\times600 and 600\times2 respectively.

Fixedgrid has high compute density and its vectorized kernels dominate performance. DMA latency is almost entirely overlapped with computation (Figure 2.13(c), where the DMA\_wait component is almost invisible). Therefore, all three implementations perform similarly (Figure 2.12(c)). Execution time with STRIDER is 2.4\% higher than execution time with the SDK implementation, whereas execution time with SEQUOIA is 14\% higher than with the SDK implementation. This result arises from subtle differences in signaling mechanism, parameter passing, and decomposition in the runtime systems. With SDK and STRIDER, SPEs performs decomposition of iteration space and calculation of starting addresses for DMA transfers in parallel, where each SPE calculates the bounds of its own partition of the iteration space. With SEQUOIA, the PPE performs these, thereby creating a centralized bottleneck during loop scheduling. We illustrate this difference in Figure 2.13(c) by comparing the Others component among the three implementations. SEQUOIA also uses a different SPE-PPE signaling method and incurs higher DMA preparation costs on the PPE. STRIDER reduces the DMA preparation cost by 21\% compared to SEQUOIA (Figure 2.13(c), DMA\_prep component).

The data point denoted as SDK\_org in Figure 2.13(c) shows the performance of the original SDK implementation of Fixedgrid [37]. This implementation vectorizes only the column discretization kernel, which specifically discretizes two columns simultaneously on each SPE. The code uses strided DMA transfers to pack column elements for vectorization in local stores. Vectorizing row discretization requires shuffling of data in local stores. The original implementation interleaves scalar elements in vector arrays, leaving half of the vector execution units unutilized. We modify the row discretization kernel to fully utilize the vector execution units and use the modified kernel for comparison with STRIDER and SEQUOIA. This modification brings about a 52\% kernel performance improvement and a 55\% overall performance improvement.
2.4 Summary

We presented STRIDER, a runtime system for the optimization of strided data accesses in parallel loops, targeting multi-core processors with explicitly managed memory hierarchies. STRIDER leverages (a) variable-depth buffering to improve temporal locality in stencil codes, (b) monochromatics and aggregate task blocking to reduce the critical path for preparing, initiating, and detecting completion of DMA transfers, and (c) multi-level loop decomposition to improve load balancing in nested loops. The runtime system abstracts away details of data transfers and generates efficient code from high-level specifications of loop nests and array references. These specifications conform to common representations used in parallelizing and optimizing compilers, for which STRIDER serves as an optimization module. The prototype of STRIDER on Cell outperforms modern language frameworks by up to a factor of $3.6 \times$ in kernels and applications dominated by strided access patterns, while performing competitively to hand-optimized code.
Chapter 3

Programming Abstraction for Strided Data Accesses

Platforms with software-managed memories offer a great control of data locality as well as the great complexity of programming. In Chapter 2, we discuss how the programming framework we introduce, STRIDER, helps achieve a performance competitive to that of the hand-optimized alternative while outperforming the existing framework, SEQUOIA. In this chapter, we discuss whether the programming abstraction, which STRIDER offers, provides programmers with application-specific optimizations at a comparable level of productivity in terms of the lines of code.

Automating caching and prefetching in software-managed local memories has been studied extensively. Prior solutions can be categorized into solutions that delegate data caching to programmers through a simplified API that hides the details of prefetching [41, 42], solutions that semi-automate caching and prefetching by leveraging annotations provided by programmers [21, 28, 41, 43–46], and solutions that hide all details of caching and prefetching from programmers, presenting them instead with a high-level programming API based on the abstraction of a single shared address space, such as OpenMP [23, 31, 47, 48].

We deploy STRIDER as a plug-in to provide runtime support to source-to-source compilers that implement transparent software caching and prefetching for high-level programming models, specifically OpenMP [23, 47, 49, 50]. STRIDER is also usable as a stand-alone programming framework for application fine-tuning.

The rest of the chapter is organized as follows. Section 3.1 provides an example of code using STRIDER. Section 3.2 compares the programmability of the framework in terms of the line counts of the application implementations in the APIs of three different programming paradigms. Section 3.3 discusses related work and Section 3.4 summarizes the chapter with future directions.
```python
for (i = 0; i < nx; i += 3)
    for (j = ny-2; j >= 1; j-=2)
        for (k = 1; k < nz-1; k++)
            for (m = 0; m < 5; m++)
                C[i][j][k][m] = dt*A[i][j][k][m] + B[i][j][k][m];
```

Figure 3.1: An example of a nested loop.

<table>
<thead>
<tr>
<th>Induction variable</th>
<th>Array dimension</th>
<th>Start</th>
<th>End</th>
<th>Step size</th>
<th>Loop level</th>
<th>Step direction</th>
</tr>
</thead>
<tbody>
<tr>
<td>i</td>
<td>4</td>
<td>0</td>
<td>nx</td>
<td>3</td>
<td>4</td>
<td>forward</td>
</tr>
<tr>
<td>j</td>
<td>3</td>
<td>1</td>
<td>ny-1</td>
<td>2</td>
<td>3</td>
<td>backward</td>
</tr>
<tr>
<td>k</td>
<td>2</td>
<td>1</td>
<td>nz-1</td>
<td>1</td>
<td>2</td>
<td>forward</td>
</tr>
<tr>
<td>m</td>
<td>1</td>
<td>0</td>
<td>5</td>
<td>1</td>
<td>1</td>
<td>forward</td>
</tr>
</tbody>
</table>

Table 3.1: Loop parameters for the loop shown in Figure 3.1.

<table>
<thead>
<tr>
<th>Dim 4 size</th>
<th>Dim 3 size</th>
<th>Dim 2 size</th>
<th>Dim 1 size</th>
<th>Element (dim 0) size</th>
</tr>
</thead>
<tbody>
<tr>
<td>nx</td>
<td>ny</td>
<td>nz</td>
<td>6 (=5+pad)</td>
<td>8</td>
</tr>
</tbody>
</table>

Table 3.2: The descriptor of array double A[nx][ny][nz][5].

Appendix B presents further examples of coding in STRIDER API.

### 3.1 A Working Example

We use the parallel nested loop in Figure 3.1 as a working example to demonstrate the use of STRIDER. STRIDER composes the offloaded loop from two pieces of code, an SPE offloader shown in Figure 3.2 and an externally provided compute kernel, which is not shown. The SPE offloader defines the shape of blocks to be transferred, describes the iteration space of the offloaded loop, sets the user-provided kernel, and calls the buffer manager.

Lines 7–10 declare data structures for the iterator, blocks, and monochromatics. Line 12 defines the shape of a block to be transferred. Lines 13–17 describe the parallel loop using the parameters for the loop and the arrays, shown in Table 3.1 and Table 3.2 respectively. Line 18 pre-computes the strides using the element size of the reference array as input. The block access list generator uses the pre-computed strides in the iterator to generate memory offsets. STRIDER allows grouping and handling the arrays of different element sizes in a monochromatic. The last parameter of set_wave(), called amplitude, scales the offset independently for each array.
```c
void nested_loop(int fid)
{
    uint32_t i_start = 0;
    uint32_t i_stop;
    uint32_t unitsz = A.dim[0];
    IterSpace is;
    Iterator iti, ito;
    BlkDesc blk;
    MonoChromatic mci, mco;
    set_blockdesc(blk, BLK_TYPE1, 1, unitsz, NULL, 0);
    alloc_iterbases(&is, 4);
    set_ibasis(&is, 4, 0, nx, A.dim[4], 3, 4, FWRD);
    set_ibasis(&is, 3, 1, ny-1, A.dim[3], 2, 3, BWRD);
    set_ibasis(&is, 2, 1, nz-1, A.dim[2], 1, 2, FWRD);
    set_ibasis(&is, 1, 0, 5, A.dim[1], 2, 1, FWRD);
    init_iterspace(&is, unitsz);
    i_start = 0;
    i_stop = get_num_iterations(&is);
    compute_bounds(&i_start, &i_stop, 1, IDSpe, Nspe);
    alloc_iterator(&iti, &is);
    alloc_iterator(&ito, &is);
    set_iterator(&iti, i_start, i_stop);
    set_iterator(&ito, i_start, i_stop);
    alloc_monochromatic(&mci, 2, 1, &iti);
    set_wave(&mci, 0, A.ea, 0, &blk, NULL, NoApnd, 1);
    set_wave(&mci, 1, B.ea, 0, &blk, NULL, NoApnd, 1);
    alloc_monochromatic(&mco, 1, 1, &ito);
    set_wave(&mco, 0, C.ea, 0, &blk, NULL, NoApnd, 1);
    set_kernel(fid, kernel_nested_loop);
    set_double_param(pass.dt, 0);
    strider_nLmD_f_pn(fid, &mci, &mco, NULL, buflen);
    free_monochromatic(&mci);
    free_monochromatic(&mco);
    free_iterator(&it);
    free_iterbases(&is);
}
```

Figure 3.2: A naïve SPE offloader for the loop shown in Figure 3.1.
Table 3.3: The loop parameters in the code shown in Figure 3.5.

<table>
<thead>
<tr>
<th>Induction variable</th>
<th>Array dimension</th>
<th>Start</th>
<th>End</th>
<th>Step size</th>
<th>Loop level</th>
<th>Step direction</th>
</tr>
</thead>
<tbody>
<tr>
<td>i</td>
<td>2</td>
<td>0</td>
<td>nx</td>
<td>3</td>
<td>2</td>
<td>forward</td>
</tr>
<tr>
<td>j</td>
<td>1</td>
<td>1</td>
<td>ny-1</td>
<td>2</td>
<td>1</td>
<td>backward</td>
</tr>
</tbody>
</table>

Table 3.4: The descriptor of array transformed for the code shown in Figure 3.5 from the original form in Table 3.2.

<table>
<thead>
<tr>
<th>Dim 2 size</th>
<th>Dim 1 size</th>
<th>Element size</th>
</tr>
</thead>
<tbody>
<tr>
<td>nx</td>
<td>ny</td>
<td>nz×6×8</td>
</tr>
</tbody>
</table>

Lines 19–26 perform iteration space decomposition and define the loop iterator of each SPE. Note that SPEs perform the decomposition in a distributed manner, that is, each SPE receives the iterator of the entire loop and computes the local iteration space that the SPE will execute. Staticaly distributed iteration partitioning is efficient, whereas the fine-grain task partitioning scheme that STRIDER uses also preserves load balancing. More sophisticated dynamic iteration space partitioning schemes are possible but we do not examine them in this work.

Lines 28–32 specify the monochromatics for the input arrays (A and B) and the output array (C). In this example, all three arrays have identical ranks, dimensions, and access patterns, therefore STRIDER reuses a single iteration space descriptor (is) for all three arrays and allocates one monochromatic for input and one for output. The input monochromatic merges arrays A and B, whereas the output monochromatic includes array C. Line 34 links the external kernel, shown in Figure 3.3. Line 35 passes a parameter to the kernel. The PPE performs parameter passing by directly setting the values via the effective addresses of variables in SPE local stores. Line 36 invokes the STRIDER buffer manager for the monochromatics. Finally, Figure 3.4 shows how an application initiates offloading. This example contains two sections of code. The first one copies the array parameters to the data structures on the local store of each SPE via the pointers. The pointer points an effective address transparently translated to a physical address. The array parameter specifies the dimension and address of an array that will be used throughout the program execution. Then, it makes the structures available for subsequent offloading. This mechanism is based on an existing work [11]. The second section calls the subroutine shown in Figure 3.2 on each SPE.
```c
void kernel_nested_loop(uint32_t buf,
  uint32_t buflen, uint32_t *idx)
{
  uint32_t i;
  dt = peek_double_param(0);
  A = (double*) get_buf_ptr(buf, Ibuf, 0);
  B = (double*) get_buf_ptr(buf, Ibuf, 1);
  C = (double*) get_buf_ptr(buf, Obuf, 0);
  for (i=0; i<buflen ; i++)
    C[i] = dt*A[i] + B[i];
}
```

Figure 3.3: The user kernel for the loop shown in Figure 3.1.

```c
strider_start();
MMGP_offload();
for(int i=0; i<Nspes; i++)
{
  ((struct param_t *)param[i])->A = A;
  ((struct param_t *)param[i])->B = B;
  ((struct param_t *)param[i])->C = C;
  MMGP_start_SPE(i, INITPARAM);
}
MMGP_wait_SPE(INITPARAM);
MMGP_offload();
for(int j=0; j<Nspes; j++)
{
  int i = get_speOrder(j);
  ((struct param_t *)param[i])->dt = dt;
  MMGP_start_SPE(i, id_nested_loop);
}
MMGP_wait_SPE(id_nested_loop);
strider_finish();
```

Figure 3.4: The PPE offloader for the loop shown in Figure 3.1.
void nested_loop(int fid)
{
    uint32_t i_start = 0;
    uint32_t i_stop;
    uint32_t unitsz = A.dim[1]*A.dim[0];
    uint32_t blkW = (nz-2)*unitsz;
    int32_t phase = unitsz;

    IterSpace is;
    Iterator iti, ito;
    BlkDesc blk;
    MonoChromatic mci, mco;

    set_blockdesc(blk, BLK_TYPE1, 1, blkW, NULL, 0);
    alloc_iterbases(&is, 2);
    set_ibasis(&is, 2, 0, nx, A.dim[4], 3, 2, FWRD);
    set_ibasis(&is, 1, 1, ny-1, A.dim[3], 2, 1, BWRD);
    init_iterspace(&is, A.dim[2]*unitsz);
    i_start = 0;
    i_stop = get_num_iterations(&is);
    compute_bounds(&i_start, &i_stop, 1, IDSpe, Nspes);

    alloc_iterator(&iti, &is);
    alloc_iterator(&ito, &is);
    set_iterator(&iti, i_start, i_stop);
    set_iterator(&ito, i_start, i_stop);

    alloc_monochromatic(&mci, 2, 1, &iti);
    set_wave(&mci, 0, A.ea, phase, &blk, NULL, NoApnd, 1);
    set_wave(&mci, 1, B.ea, phase, &blk, NULL, NoApnd, 1);
    alloc_monochromatic(&mco, 1, 1, &ito);
    set_wave(&mco, 0, C.ea, phase, &blk, NULL, NoApnd, 1);

    set_kernel(fid, kernel_nested_loop2);
    set_double_param(pass.dt, 0);
    strider_nLmD_f_pn(fid, &mci, &mco, NULL, 1);

    free_monochromatic(&mci);
    free_monochromatic(&mco);
    free_iterator(&it);
    free_iterbases(&is);
}

Figure 3.5: An optimized alternative to the one shown in Figure 3.2 using blocking.
The major inefficiency of the code shown in Figure 3.2 is that a transfer block only consists of two array elements of type double. Thus, this entails a large number of transfers, carrying significant overhead. Figure 3.5 shows an alternative. In this approach, the array is considered as a virtual two-dimensional array with a larger element size as shown in Table 3.4, and the original 4-level loop is converted to a 2-level loop as shown in Table 3.3. The transfer block size becomes \((nz-2)\times6\times8\) bytes which is set in line 14 in Figure 3.5. This size is actually smaller than the element size of the virtual array. Thus, a phase is used to adjust the starting address of the transfer block in lines 29, 30, and 32 in Figure 3.5, to facilitate folding the original 4-dimensional loop into 2 dimensions. Note that the unit stride of the iteration space is still set to \(nz\times6\times8\) in line 18. In STRIDER, a programmer can write offloaders to switch between different implementations, as those shown in Figure 3.2 and Figure 3.5, for better performance on the fly based on observations on runtime parameters, such as the dimension of array, and buffer sizes.

### 3.2 Programming Complexity

Table 3.5 shows the lines of code added for each application using each programming model while converting the serial implementation to a parallel counterpart. The serial implementations do not include vectorized kernels, and run only on PPE. On the other hand, we use vectorized kernels for offloading in parallel implementations. We do not count comments or empty lines. The line counts shown in Table 3.5 slightly differ from those shown in [23]. It is partly because the baseline codes have been further optimized as we discussed in Section 2.3.5. In addition, we exclude the kernels themselves in counting the lines added to support parallel execution. This is to count the lines relevant to manage the locality of data. Overall, STRIDER reduces the number of lines by a factor of 4.3\(\times\) compared to the hand-optimized implementations. It adds 46% more lines compared to SEQUOIA without considering task mapping files in SEQUOIA. However, by counting the lines written in task mapping files, SEQUOIA adds 29% more lines compared to STRIDER. As we discussed in Section 2.3.5, the difference in performance with PBPI mostly stems from the runtime performance and the NUMA-aware page allocation. However, the difference with Jacobi is largely attributed to the existence of abstraction to take advantage of temporal locality. In this case, STRIDER only adds 12% more lines while outperforming SEQUOIA by a factor of 2\(\times\). STRIDER can serve as an optimization module for a source-to-source compiler of OpenMP style programming models. Cellgen has partially utilized STRIDER as a back-end runtime module [23]. However, the OpenMP style model requires extra expressions to take full advantage of the optimizing capability of STRIDER. Appendix B shows how SEQUOIA and STRIDER apply to these applications.
### 3.3 Related Work

Automated caching and prefetching in software-managed local memories has been studied extensively due to its importance for performance and programmer productivity. Prior solutions can be categorized into solutions that delegate data caching to programmers through a simplified API which hides the details of prefetching [41, 42, 51], solutions that semi-automate caching and prefetching by leveraging data annotations provided by programmers [21, 28, 43, 44], and solutions that hide all details of caching and prefetching from programmers, presenting them instead with a high-level programming API, such as OpenMP [23, 31, 47, 48].

Library frameworks that delegate software caching and prefetching to programmers such as IBM’s ALF [42], RapidMind [41], and CellMT [51] provide APIs to express contiguous (unit-stride) and non-contiguous (non-unit-stride) data transfers to and from parallel tasks executing on heterogeneous cores. The implementation of these data transfers is hidden in the respective runtime libraries. ALF and RapidMind delegate caching to programmers and support static prefetching schemes via programmer annotations that pin-point opportunities for double- or triple-buffering. CellMT on the other hand implements multithreading on SPEs, which simplifies the task of overlapping computation with communication via a thread-level API. The available documentation of these frameworks does not address issues related to optimizing the code for preparing and scheduling strided data transfers. STRIDER optimizes strided data transfers through monochromatics, multi-level decomposition of data accesses and platform-specific optimizations. Furthermore, STRIDER provides variable-depth buffering for improving temporal locality, a capability not available in existing library frameworks. Finally, STRIDER provides a high-level, yet rich API for controlling the blocking and partitioning of data transfers to programmers and compilers alike.

Parallel programming models such as CellSs [43, 44] and SEQUOIA [21, 28] rely on user-provided data annotations to perform data transfers to and from software-managed local memories. Similarly to library frameworks, these programming models delegate the responsibility of caching to programmers. The models provide static prefetching capabilities through double-buffering or

<table>
<thead>
<tr>
<th>Application</th>
<th>Serial</th>
<th>Offload</th>
<th>Cell SDK</th>
<th>SEQUOIA</th>
<th>STRIDER</th>
</tr>
</thead>
<tbody>
<tr>
<td>Jacobi</td>
<td>38</td>
<td>16</td>
<td>+658</td>
<td>+122 (+99)</td>
<td>+137</td>
</tr>
<tr>
<td>Fixedgrid</td>
<td>5,565</td>
<td>110</td>
<td>+755</td>
<td>+91 (+73)</td>
<td>+163</td>
</tr>
<tr>
<td>PBPI</td>
<td>8,623</td>
<td>57</td>
<td>+634</td>
<td>+114 (+120)</td>
<td>+179</td>
</tr>
</tbody>
</table>

Table 3.5: Lines of code added for each combination of programming method and application. `offload` shows the number of lines in the serial version corresponds to the offloaded computation. The numbers inside of parenthesis show the line counts of the task mapping file of SEQUOIA written in XML format by programmers.
triple-buffering of selected data streams of array elements. Similar optimizations are provided by the compilers of certain stream programming languages [52, 53]. STRIDER introduces optimizations of data transfers that extend beyond a single stream with a single stride, while being suitable as a replacement of the data transfer framework in the aforementioned programming models. Recent extensions of both CellSs [54] and SEQUOIA [55] include runtime techniques (based on scheduling tasks that access common data blocks on the same core) and static techniques (based on loop fusion for reducing data reuse distance) for improving temporal locality. STRIDER’s variable depth buffering is complementary to these techniques, none of which is applicable to stencil codes. Variable depth buffering targets temporal locality of partially overlapping data blocks accessed by multiple tasks scheduled on the same core and optimizes effectively stencil computations.

Recently, implementations of OpenMP [23, 47] on Cell have demonstrated high performance in parallel applications, compared to hand-crafted implementations of the same implementations using the IBM SDK. Earlier implementations based on a software cache that exhibited low efficiency [8] have been replaced with implementations that include direct data buffering, multi-buffering, and loop optimizations such as fusion and tiling [31, 47] to improve performance. Related work on polyhedral models [48, 56] enables the automatic generation of DMA transfers from affine nested loops. IBM’s DBDB framework [31] is perhaps the closest to STRIDER. STRIDER departs from DBDB in several aspects, including the methods for grouping data accessed with multiple strides, the methods for optimizing the code that prepares and schedules DMA lists, and the methods that improve temporal locality. A point-to-point comparison between STRIDER and IBM’s OpenMP compiler technology would be instrumental but is not feasible due to the proprietary nature of IBM’s code. Our qualitative comparison suggests that STRIDER provides supplemental optimizations that can benefit OpenMP compiler technologies on multi-core processors with software-managed memories.

### 3.4 Future Work

We are currently exploring STRIDER extensions including automatic data aggregation for dependent tasks based on input from dependence analysis and automatic derivation of optimal block shapes and sizes. We are also introducing new capabilities for dependence-driven parallel execution, in particular, pipelining and wavefront execution schemes [57]. Future plans include further integration of STRIDER with the OpenMP standard, exploration of STRIDER as a back-end for advanced static optimization frameworks [48], and introduction of techniques to exploit on-chip communication and direct core-to-core transfers to further reduce memory latency in STRIDER.

Since IBM’s cancellation of next generation Cell processors, we engaged in porting STRIDER
to platforms based on graphics processing units (GPUs) for supporting applications with low computational density, experimental multi-core processor architectures with explicitly managed memories [58], and Intel SCC [15]. We are also investigating specific aspects of STRIDER that can benefit multi-core processors with coherent caches and software prefetching [10, 59].

In a Blue Gene/Q compute chip (BQC), there is a level 1 cache prefetch module (L1P) for each core interfacing L1 and the rest of the memory subsystem [10, 60]. L1P supports two types of prefetching mechanisms to hide access latency: stream-prefetch and list-prefetch. In both cases, prefetching is driven by the L1 cache miss. Prefetching requires identification of future accesses. Stream-prefetch targets the sequential accesses on contiguous memory addresses, where the address of future access is determined by the fixed stride that equals the size of a prefetch line. List-prefetch targets repetitive accesses with a sequence of addresses. Programmers rely on compiler directives to mark a section of code to record the addresses of L1 cache misses and to save them into a list in the main memory managed by the device driver. Then, the prefetch engine relies on the list to identify future accesses. As the list resides on the main memory, it is possible to replace it with a predefined list of addresses prepared by the block access list generator of STRIDER [10]. Further investigation is necessary to identify which specific methods or interfaces enable the replacement of lists. The DMA handler of STRIDER is not needed as the prefetch engine schedules accesses based on L1 misses. Software’s access generation combined with hardware’s scheduling is a promising combination to handle applications with regular access pattern and non-uniform workload. It is not clear, however, whether it is useful to explicitly generate redundant addresses to mimic the temporal locality, which STRIDER manages directly using variable-depth buffering. There are as many prefetch engines as the number of hardware threads. Thus, the access pattern grouping of STRIDER can help effectively handle multiple streams per prefetch engine.

The Intel Xeon Phi co-processor has a high off-chip memory access latency, a small last-level cache capacity and in-order pipelines [59]. Therefore, applying a prefetching technique is critical to achieve a good performance. The Xeon Phi supports prefetching based on both software and hardware. The former requires inserting prefetch instructions. Programmers rely on either the high-level compiler directives or the low-level intrinsics. *Prefetch distance*, defined as the number of loop iterations a prefetch instruction is inserted ahead, is important to effectively hide access latency. The sophisticated compiler automatically determines a prefetch distance by considering the latencies in the memory hierarchy and the estimated load in the memory subsystem, the time to execute loop computations. If an application specific data blocking is preferred and the optimal block shape is determined at run time, STRIDER may provide a similar optimization for such a dynamic execution. Given the run time estimation of prefetch distance, STRIDER can apply variable-depth buffering. Dynamic adjustment of buffering depth is left for future work.
3.5 Summary

STRIDER is a library framework for programming applications that access multi-dimensional arrays on multi-core processors with explicitly managed memories. STRIDER offers a higher abstraction than existing library frameworks [42, 61] and higher flexibility than existing code generation frameworks [21, 23], while sustaining higher performance. STRIDER provides abstractions that are absent in the existing high-level frameworks [21, 23] at the cost of a marginal increase in lines of code to be written by programmers. It is currently presented as a library framework, and will evolve into a code generation framework with a polyhedral loop analyzer without losing flexibility [48]. Access generation optimization in STRIDER will be particularly useful on architectures supporting software-prefetching with user-supplied access lists [10].
Chapter 4

Identifying Scalability Bottleneck from the Data with Heavy-tailed Properties

This Chapter and Chapter 5 discuss the study for scaling irregular graph-based applications. EpiSimdemics is an agent-based simulation framework for contagion simulation that can be used to model a wide range of epidemic scenarios, as well as the impact of counter measures [62, 63]. It has been used in multiple time-critical studies, including investigations into methods for protecting members of the National Guard during a pandemic [64, 65] and for determining potential outcomes during the early days of the H1N1 pandemic [66]. Integration into the 24-hour decision cycle of the federal government’s response to such a crisis demands highly optimized modeling software. The analysts performed course-of-action analyses to estimate the impact of policy maker scenarios such as closing schools and shutting down workplaces.

EpiSimdemics is implemented using the Charm++ programming model [67] and has shown good scalability up to modest cluster sizes of several hundred to a thousand cores [68]. When scaling to a larger number of cores, however, EpiSimdemics faces severe scalability bottlenecks that prohibit its use for large, time-sensitive problems.

In this chapter, we analyze the source of these scalability limitations and discuss a set of novel strategies to overcome them. We first focus on the impact of using graph data with heavy-tailed degree distributions, which is common in social network graphs [69–72]. We show how such a property limits scaling and eventually how we transform a graph to achieve a better load balance while partitioning the graph. While data locality is critical for minimizing the cost of non-local accesses, load balance is also important. The challenge is that load balance may sacrifice locality or vice versa. The heavy-tailed property of a graph makes the locality optimization difficult or ineffective.
To enable the application-specific handling of data, we introduce a workload model that allows state-of-the-art graph partitioners to use custom, application-specific load balancing constraints for EpiSimDemics in addition to custom data partitioning. We also propose a technique to preprocess the input graph to split heavy nodes, which enables graph partitioners to produce a more balanced workload distribution. Chapter 5 discusses remaining bottlenecks and demonstrates the scalability of the application by the optimizations tailored to this particular application as well as other optimizations implemented in the underlying runtime system.

4.1 EpiSimdemics

EpiSimDemics is a contagion diffusion simulation code, which relies on several underlying base technologies [62, 73, 74]. Here, we describe the contagion simulation algorithm and provide other relevant background material necessary to understand the problem.

EpiSimDemics is an agent-based simulator that models the spread of contagious disease over social contact networks. It is based on a hybrid time-stepped, discrete-event simulation (DES) approach. Its input is a bipartite graph consisting of person and location nodes, with edges between them representing a visit by a person to a specific location at a specific time. We formally represent our bipartite graph as $G = (V_\phi, V_\lambda, E)$, where $V_\phi$ is the set of person vertices, $V_\lambda$ is the set of location vertices, and $E$ is the set of edges $\{(v_i, v_j) | v_i \in V_\phi, v_j \in V_\lambda\}$. This graph is a synthetic network based on census and other data [1, 2]. Table 4.1 lists the population sizes of some representative US states, from the smallest state to the largest. These sets are chosen across a diverse geographic region. In the following we will refer to these data sets simply by the acronym of the respective state.

We call the bipartite graph the person-location graph. The average number of visits are 5.5 for person nodes and 21.5 for location nodes. With the edge weight representing multiple visits to the same location, the average degree is 3.86 for person nodes and 15.1 for location nodes. The person-location graph is used to implicitly construct a person-person graph, whose edges represent the co-location of two people in time and space and which is ultimately used to determine any disease transmission between co-located people. This conversion leads to a factor of nine decrease in average degree between the graphs, and helps to reduce the communication overhead. On the other hand, this also may aggravate load imbalance. For example, in a household, a school-aged child may contact a hundred others at school while the parents only meet a dozen co-workers per day. As a school is filled with such children, the variation in the amount of computation for potential disease transmission is far greater between homes and schools than between parents and children.

A person-location graph transforms into a person-person graph as locations process visit messages
received. Each location stores visit messages until it receives all of the messages destined to it. Once receiving the entire set of messages, it begins converting each message into two events, *arrival* and *departure*. These events are stored in a list ordered by the time of the event. Upon constructing the list, each location reads its event list in order. Then, it adds a visitor into its set of current visitors as it encounters an arrival event, and removes a visitor from the set as it encounters a departure event. When removing a visitor, we compute the between-host disease propagation, as described in [75]. If the visitor is infectious, we compute the probability of disease transmission for each pair of departing visitor and susceptible visitor. Similarly, if the visitor is susceptible, we compute the probability for each pair of the departing visitor and infectious visitor. For the co-location period $t$ of an infectious visitor $i$ and a susceptible visitor $s$, we compute the probability as

$$p = 1 - (1 - r_i r_s)^t$$  \hspace{1cm} (4.1)$$

where $r_i$ is the infectivity, the likelihood of a visitor $i$ infecting any other per unit contact time, and $r_s$ is the susceptibility, the likelihood of a visitor $s$ getting infected by any other per unit contact time. These parameters are demographic dependent [75]. The probability is used to make a binary decision. If it is above a threshold, the susceptible is infected. The assumption here is that the interactions between any pair of agents as well as the spread of infections are independent. As we discuss the algorithm in Section 4.1.1, we choose a simulation time step smaller than the latent period to uphold the validity of the assumption.

A person’s health state is tracked using a probabilistic timed transition system (PTTS), a finite state machine with the addition of a dwell time (the time a person will remain in a state before automatically transitioning to the next state) distribution for each state, and sets of probabilistic transitions between states. Different sets of transitions are used, depending on the treatment received by the person, such as vaccination [73]. Each state has a set of attributes, including infectivity level and susceptibility level, which are used in the disease transmission calculation as

<table>
<thead>
<tr>
<th>Data name</th>
<th>Visits</th>
<th>People</th>
<th>Locations</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>US</strong> (United States)</td>
<td>1,541,367,574</td>
<td>280,397,680</td>
<td>71,705,723</td>
</tr>
<tr>
<td><strong>CA</strong> (California)</td>
<td>183,858,275</td>
<td>33,588,339</td>
<td>7,178,611</td>
</tr>
<tr>
<td><strong>NY</strong> (New York)</td>
<td>98,350,857</td>
<td>17,910,467</td>
<td>4,719,921</td>
</tr>
<tr>
<td><strong>MI</strong> (Michigan)</td>
<td>52,534,554</td>
<td>9,541,140</td>
<td>2,490,068</td>
</tr>
<tr>
<td><strong>NC</strong> (North Carolina)</td>
<td>47,130,620</td>
<td>8,541,564</td>
<td>2,289,167</td>
</tr>
<tr>
<td><strong>IA</strong> (Iowa)</td>
<td>15,280,731</td>
<td>2,766,716</td>
<td>748,239</td>
</tr>
<tr>
<td><strong>AR</strong> (Arkansas)</td>
<td>14,803,256</td>
<td>2,685,280</td>
<td>739,507</td>
</tr>
<tr>
<td><strong>WY</strong> (Wyoming)</td>
<td>2,756,411</td>
<td>499,514</td>
<td>144,369</td>
</tr>
</tbody>
</table>

Table 4.1: Population data of various sizes based on a 2009 American Community Survey and other data [1, 2]. US includes 48 contiguous states and DC.
in Equation 4.1. The PTTS implements the SEIR model, where the state of an agent at any given time is either susceptible, exposed, infectious, or removed [75]. The transition from susceptible to exposed is triggered by infection, and the other transition occurs spontaneously after a random period of dwell time. The disease model specifies the distribution of the dwell time in addition to the sets of transitions between states. It is used to construct a PTTS. EPI SIM DEMICS can simulate simultaneous contagion diffusion using multiple PTTSs given multiple disease models.

EPI SIM DEMICS has a domain-specific language for specifying complex interventions and behavior, such as vaccinations, school closures, and anxiety levels [73]. A person’s actions can depend on demographic information, such as age and income, and dynamic information such as current symptom level or the number of people who are currently ill. Both the disease dynamics and the scenario intervention introduce dynamism to the system. As a result, the underlying graph transforms, and the agent behavior changes as well as the state. Furthermore, the feedback from the changes recommends revising interventions. In other words, the agent, the network, and the intervention co-evolve. However, we do not enable intervention dynamics for this study as we discuss in Section 4.3.1.

4.1.1 The EPI SIM DEMICS Core Algorithm

We take advantage of the fact that most infectious diseases have a latent period, the time between a person becoming infected and being able to infect others. This lets us process all of the interactions for a time step in parallel, without affecting causality. The basic algorithm for each time step (currently one simulation day) is shown in Figure 4.1.

While this design requires two global synchronization points for each iteration and therefore leads to a bulk-synchronous model, EPI SIM DEMICS typically only requires the execution of a moderate number (120–180) of fairly long simulation iterations on most inputs, representing three to four months of simulated time. This helps mask the effects of the bulk synchronous model on scalability. We label sub-steps 1-2 as person phase, and sub-steps 3-5 as location phase.

4.2 Scalability Challenges with Social Network Data

The input graphs to EPI SIM DEMICS are derived from real-world social contact network data [2], and follow heavy-tailed degree distributions. Such data may further present a heavy-tailed load distribution to an application as more messages require an increased amount of processing. This property presents unique challenges for scalability as:
1. In parallel, each person recalculates their health state and decides on the locations to visit, based on their current normative schedule, health state, and public policy such as school closings. For each location visited, the object representing the person sends a “visit” message to the object representing the visited location with the ID and the health state of the person, as well as the start time and the end time of the visit.

2. Synchronize to ensure all visit messages have been received, as receivers have no prior knowledge of how many messages to expect and from whom.

3. In parallel, each location constructs a sequential and local DES by converting each visit message into an arrive event and depart event. The DES is executed, computing the interactions between each pair of susceptible and infectious people who are at the location at the same time. For each interaction that results in disease transmission, an “infect” message is sent to the infected person.

4. Synchronize to ensure receipt of all infect messages.

5. In parallel, each person that received an infect message updates its health state.

6. Global system state is updated (e.g., number of currently infected people).

Figure 4.1: EPISIMDEMICS core algorithm for each time step. We label sub-steps 1-2 as person phase, and sub-steps 3-5 as location phase.

- Heavy-tailed load distributions make load balancing difficult when partitioning.
- Partitioning to optimize locality of data by minimizing total edge cuts may often only be achieved by sacrificing load balancing, and vice versa.
- Balancing non-local accesses across distributed memories while minimizing their sum and balancing the computational load simultaneously is extremely challenging.

These challenges have a profound impact on our decisions regarding how to handle and scale such data sets, as we show in the rest of the paper.

4.2.1 A Simple Example

Here, we present a simple example of the aforementioned challenges. Figure 4.2(a) and (b) show the optimal partitioning in terms of load balancing without considering edge cuts, and edge cuts without considering load balancing, respectively. In this example, the most heavily-loaded node (Node 1 with weight 8) has the most edges and balancing load requires cutting all edges around
Figure 4.2: The optimal 5-way partitioning result for minimizing (a) load imbalance and (b) edge cut. Node 1 has weight 8, Nodes 2-5 have weight 2, and Nodes 6-13 have weight 1 each.

<table>
<thead>
<tr>
<th>Case</th>
<th>(a)</th>
<th>(b)</th>
</tr>
</thead>
<tbody>
<tr>
<td>total cuts</td>
<td>8</td>
<td>6</td>
</tr>
<tr>
<td>max load per partition</td>
<td>8</td>
<td>10</td>
</tr>
<tr>
<td>max edge cuts per partition</td>
<td>8</td>
<td>6</td>
</tr>
<tr>
<td>number of partitions</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>total vertex weight</td>
<td>24</td>
<td></td>
</tr>
<tr>
<td>total number of edges</td>
<td>12</td>
<td></td>
</tr>
<tr>
<td>average number of edges per partition</td>
<td>2.4</td>
<td></td>
</tr>
<tr>
<td>average vertex load per partition</td>
<td>4.8</td>
<td></td>
</tr>
</tbody>
</table>

Table 4.2: The details of the example shown in Figure 4.2.

this node in Figure 4.2(a). Partitioning focusing on edge cuts, on the other hand, would lead to the distribution in Figure 4.2(b) with an edge cut of 6 vs. 8 in Figure 4.2(a). However, the ratio of the maximum load per partition to the average is 1.67 in Figure 4.2(a) and 2.08 in Figure 4.2(b), showing the advantage of the distribution in Figure 4.2(a). The average load per partition is 24/5 and the average number of edges per partition is 12/5 in this example. Note that even in the case (a) the load variance is 67%. With heavy-tailed distribution, a small number of cases with an extremely large quantity make it difficult to balance. We further investigate how such a property affects the scalability of an application in Section 4.2.2. In addition, the ratio of the maximum edge cuts per partition to the average number of edges is 3.33 in Figure 4.2(a) and 2.5 in Figure 4.2(b). This shows how the number of edge cuts among partitions is unbalanced compared to the hypothetical case where every edge is cut but evenly distributed among the partitions. Table 4.2 summarizes the observations.
Figure 4.3: The bound on the scalability \((S_{ub}/D)\) is shown in terms of the size of data \((D)\) and the exponent of the power-law degree distribution \((\beta)\), given the assumption on the load as a linear function of degree. The scalability is defined as the upper bound on speedup scaled by the size of data given the unlimited number of processors. This numerically shows Equation 4.2.

### 4.2.2 Limited Scalability with Power-law Data

We begin our analysis to gain further insight into how scalability is limited by the heavy-tailed property of data with the following assumptions for the simplicity of discussion. For a given graph \(G = \langle V, E \rangle\) with a set of vertices \(V\) and a set of edges \(E\), suppose that

- The number of vertices \((v \in V)\) with degree \(d\) follows a power-law distribution given by
  \[
  f = D \cdot \text{prob}(d) = D \cdot c \cdot d^{-\beta},
  \]
  where \(D\) is the data size, \(c\) is a scaling constant such that \(c \cdot \sum_1^\infty d^{-\beta} = 1\), and \(\beta > 1\) is the power-law exponent.
- The computational complexity of the work of a vertex \(v \in V\) is \(O(d_v)\), where \(d_v\) is the vertex degree.
- The computational load \(l_v\) of a vertex \(v \in V\) is approximated by
  \[
  l_v = \alpha \cdot d_v + \gamma \approx \alpha \cdot d_v,
  \]
  where \(\alpha\) is a model constant.

We denote the maximum of vertex loads, \(\max\{\{l_v|v \in V\}\}\), by \(l_{max}\), the maximum of vertex degrees, \(\max\{\{d_v|v \in V\}\}\), by \(d_{max}\), and the average of vertex degrees by \(d_{avg}\). Suppose we have a \(K\)-way partition of \(V\), \(P = \{p_i|i = 1, ..., K\}\), where \(V = p_1 \cup ... \cup p_K\) and \(p_i \cap p_j = \emptyset\) if \(i \neq j\). We define the load of a partition \(p \in P\) as \(L_p = \sum_{v \in p} l_v\), and the maximum of such quantities as \(L_{max}\). We represent the total sum of loads \(L_{tot}\) as \(\alpha \cdot \sum_{v \in V} d_v = \alpha \cdot D \cdot d_{avg}\). We define the estimated speedup as \(L_{tot} / L_{max}\), which is bounded by the upper bound on speedup, \(S_{ub} = L_{tot} / L_{max}\), as \(l_{max} \leq L_{max}\). As we increase the number of partitions \(K \to D\) for a large \(D\), \(L_{max}\) asymptotically approaches \(l_{max}\), especially for \(K \geq L_{tot}/l_{max}\). With power-law distribution, \(l_{max}\) will be far larger than the average.
As $L_{\text{tot}}/L_{\text{max}} = (\alpha \cdot D \cdot d_{\text{avg}})/(\alpha \cdot d_{\text{max}})$, $\log(S_{ub}) \leq \log(d_{\text{avg}} \cdot D) - \log(d_{\text{max}})$. From the power-law relationship, it is likely only one or very few vertices with $d_{\text{max}}$ exist. By approximating this quantity $f$ as 1 for $d_{\text{max}}$, we obtain $\log(D \cdot c \cdot (d_{\text{max}})^{-\beta}) = \log(1)$, and thus $\log(d_{\text{max}}) = \log(cD)/\beta$. This results in:

$$\log(S_{ub}/D) \leq \log(d_{\text{avg}}) - \frac{1}{\beta} \cdot \log(D) - \frac{1}{\beta} \cdot \log(c)$$

(4.2)

Given an unlimited number of processors, speedup is a non-decreasing function of $D$. To understand how the scalability depends on the size of power-law data, we further investigate the speedup scaled by $D$ as well as the dependence of other parameters such as $d_{\text{avg}}$ on $D$. The sum of probability $\sum_{d=1}^{\infty} c \cdot d^{-\beta} = 1$. Note that $1/c$ is a $p$-series known to converge with $\beta > 1$. Thus, both $\beta$ and $c$ are independent of $D$. The expected value of $d$, however, is $d_{\text{avg}} = \sum_{d=1}^{D-1} d^{1-\beta} / \sum_{d=1}^{\infty} d^{-\beta}$. Thus, $d_{\text{avg}}$ is bounded by a constant as $D \to \infty$ when $\beta > 2$. Note that we are particularly interested in cases of large $D$. How exactly $S_{ub}/D$ is bounded when $2 \geq \beta > 1$ is not known as $d_{\text{avg}}$ does not converge. Figure 4.3 numerically shows $S_{ub}/D$ over the sampled sets of $D = \{10^3, 10^5, 10^6, 10^7, 10^8, 10^9, 10^{10}\}$ and $\beta$ ranging from 1.2 to 4.0 by 0.2 interval. Table 4.3 lists the symbols used in this section.
Figure 4.4: The model to estimate the static workload of each location object on (a) Blue Waters and (b) Vulcan and (c) Sierra. These models approximate the mapping from application’s variables to the relative processing time of a location object. These models are platform-dependent, and built using piecewise linear regression and cross-validation. The model for Vulcan relies on the four-dimensional feature space while displaying here using two-dimensional features and the target. On the other hand, the other models rely on one-dimensional feature space. By approximating the model form as a linear function, we choose the feature space based on the five-fold cross-validation.

4.3 Handling the Social Contact Data

In this section, we describe our approaches to simultaneously balance workload and improve data locality. Section 4.3.1 describes how we define application-specific workload and estimate it. Section 4.3.2 describes various approaches to distribute workload among processors implemented in EPISIMDEMICS. Section 4.3.3 shows how the scalability is limited by the heavy-tailed property of our population data. Section 4.3.4 describes how we improve the scalability with an application-specific method to decrease work granularity.

4.3.1 Models to Estimate Workload

Balancing load distribution is important to achieve high scalability and performance. This requires a model to estimate the workload in our graphs, the weight of each work unit (vertex). We define load as the relative processing time for interaction computation of a location object. We observe that the amount of computation per person is roughly proportional to the number of messages that each person generates, which shows no extreme variance ($\text{avg} = 5.5, \sigma = 2.6$ for the US population data). Thus, we approximate the load of a person vertex as the number of messages the person generates. On the other hand, the computation per location varies significantly and requires a more detailed estimation. We adopt function approximation [76] using application-specific information relevant to the load.
We choose the linear form of piecewise linear regression as the representation of target function. We consider four variables in the context of application as model input: the sum of arrival-departure events ($N_e$), the sum of interactions ($N_i$), the sum of the reciprocal of interactions ($I_i$), and ($N_i / N_e)^2$. The choices are made for the efficiency of collecting data, training model and applying it while repeatedly validating against the model quality. We cross-validate various candidate models over the subspaces of the state variables for choosing a model in order to avoid overfitting [76]. Specifically, by approximating the model form as a linear function, we determine the feature space, on which the linear function is defined, based on five-fold cross-validation.

Note that we distinguish between static load and dynamic load. In EpiSimDEmics, the amount of computation is not deterministic. Two of the major sources of non-determinism are health state changes and interventions, described in Section 4.1. We do not attempt to address the dynamic load variation by static load balancing. Instead, we focus on the statically predictable portion of the workload by using a priori information such as $N_e$ and $N_i$. We count the pair of occupants who coexist in space and time potentially interacting with each other as (potential) interactions. Those who actually interact are only known at runtime as it depends on the disease state of each individual, and not used for modeling static load. We do not enable scenario intervention in this study.

Figure 4.4(a) shows a model that maps events to the load of location on Blue Waters listed in Table 5.1 as:
where $X$ is $N_e$, $Y$ is the load and $S(t) = 1/(1 + \rho \cdot e^{-t})$. $\varphi$ is the cross over point between the two linear models and determined experimentally. $\rho$ is set to adjust the smoothness of the transition from one model to another. In practice, we measure LocationManagers’ processing time due to the insufficient timer precision and apply it to an individual location by scaling $N_e$ with $\mu$.

Figure 4.4(b) shows a model that maps four variables to the load on the Vulcan platform listed in Table 5.1, but only visualizing with events (X1) and potential interactions (X2). We build this model by piecewise linear regression on data transformed using principal component analysis (PCA). The model computes the target $Y$ in a series of steps as follows:

$$X^a = \mu \cdot X$$
$$Y_a = 6.09 \times 10^{-6} + 7.72 \times 10^{-7} X'$$
$$Y_b = -1.25 \times 10^{-4} + 8.67 \times 10^{-7} X'$$
$$Y = Y_a \cdot S(\varphi - X') + Y_b \cdot S(X' - \varphi)$$

$Y = Y^a \cdot S(N_e - X_e) + Y^b \cdot S(N_e - \varphi)$

where $P^a$ and $P^b$ are obtained from PCA, and $C^a$ and $C^b$ are from linear regression as:

$$P^a = \begin{bmatrix} 0.4600087 & 0.8835353 & 0.0880759 \\ 0.8566902 & -0.4677173 & 0.2175372 \\ -0.2333965 & 0.0246153 & 0.9720700 \end{bmatrix}, \quad C^a = \begin{bmatrix} 2.0789e-05 \\ 2.7595e-06 \\ 4.3938e-06 \\ -3.7227e-06 \\ -6.5470e-10 \end{bmatrix}^T$$

$$P^b = \begin{bmatrix} -0.3432782 & -0.9376890 & -0.0538454 \\ 0.9135493 & -0.3466552 & 0.2127078 \\ -0.2181308 & 0.0238318 & 0.9756228 \end{bmatrix}, \quad C^b = \begin{bmatrix} -1.4288e-05 \\ -2.1692e-06 \\ 5.1251e-06 \\ -8.6395e-06 \end{bmatrix}^T$$

Figure 4.4(c) shows a model that maps events to the load of location on the Sierra platform listed in Table 5.1 as

$$X' = \mu \cdot X$$
$$Y_a = -1.2529 \times 10^{-5} + 4.7708 \times 10^{-7} X'$$
$$Y_b = -1.5255 \times 10^{-4} + 5.2019 \times 10^{-7} X'$$
$$Y = Y_a \cdot S(\varphi - X') + Y_b \cdot S(X' - \varphi)$$

where $X$ is $N_e$, and $Y$ is the load.

Figure 4.5(a) shows the distribution of in-degree per location which is the number of unique visitors.
Each visit creates an arrival event and a departure event while a visitor may visit the same location multiple times. The left plots in Figure 4.6 show the distribution of the load per location estimated by the model in Figure 4.4. We validate our model against runs on each platform, where we observe on average 5.5%, 6.1% and 7.3% error on Blue Waters, Vulcan and Sierra respectively. Note that our data does not follow the pure power-law and the load model is not exactly a linear function of the degree as assumed for the analysis in Section 4.2.2.

### 4.3.2 Distributing the Population Data

Scaling a parallel application requires consideration of both the load balance and the cost to access non-local data in distributing data to processors (or virtual processing units such as chares). We discuss four data distribution methods in this section, each of which handles load balance and data locality differently. We use them to expose the scalability bottleneck in Section 5.2.2.

We use two approaches as base cases to gain insight on how load balance and data locality affect the performance of our application. Originally, we assign objects to Charm++ chares in a round-robin (RR) fashion to approximate static load balancing. While this method requires the least effort to implement among those we discuss in this section, it completely ignores the data locality and only achieves suboptimal load balance. It relies on the modulo operation to map an object id to a processor id to which the object is assigned.

On the other hand, the zipcode-based distribution (Z) only considers data locality. It orders data objects by the associated zipcode and then divides them for an equal number of objects among processors. Figure 4.7 shows an example of distributing four locations between two processors. Each location has a zipcode and an id. We rely on zipcode for approximating the optimal locality. The reasoning behind this is that people visit locations within a certain distance from their homes (50 miles in our population data). We categorize locations into two types: home and non-home locations. We distribute locations of each type separately as illustrated in Figure 4.8. Once distributing home location objects, we assign each person object to the same processor as the one that owns the object of the person’s home. This co-location of homes and people is beneficial when a person visits home more often than a location of the other type. In our population data, a person typically visits a home at least twice per day and a non-home location mostly once.

An advanced approach may consider both the load and the locality in distributing data. Graph partitioning optimizes data locality in terms of the total number of remote accesses, which is counted by the edge cut, with a load balancing constraint. The load is application specific information. We use the load model discussed in Section 4.3.1 to estimate the workload of each work unit (vertex) in our graphs, and assign a weight to each vertex. The user may specify the tolerable variance in
Figure 4.6: The distribution of load per location vertex estimated by the model shown in Figure 4.4 with the property of data shown in Figure 4.5. The left plots show the distributions with the original graph. The right plots show those with the graph modified as we discuss in Section 4.3.4.
Figure 4.7: Two strategies to distribute location data approximating optimal locality using the zipcode. One (Z) is based on the zipcode only and the other (ZC) is on the zipcode and the sum of costs. In both cases, we sort locations by the zipcode, and then by the location id within a same zipcode. Then, we divide location data evenly between processors in terms of the number of locations (Z) or the sum of weights (ZC). With the former, one processor takes L1 and L2, and the other takes the rest. With the latter, one processor takes L1, L2, L3 and the other takes L4.

Figure 4.8: Each of the distribution strategies based on zipcode, Z and ZC shown in Figure 4.7, categorizes locations into two types: home and non-home. It handles locations of each type separately. Then, it assigns each person object to the processor on which the home location is.

the sum of load among partitions. A graph partitioner heuristically finds an optimal assignment of data to processors subject to the constraint. The distribution based on graph partitioning (GP) requires the most preprocessing as graph data needs to be labelled by the estimated load and then partitioned for each of the different number of processors to use in advance. An application relies on the mapping of vertices to processors generated by a graph partitioner to distribute data.

Another method is based on both zipcode and the estimated cost (ZC). This optimizes data locality using the zipcode same as method Z does but divides data among processors evenly in terms of the sum of the estimated costs rather than the number of objects as illustrated in Figure 4.7. This method not only requires each data object labeled by the zipcode but also by a weight in advance as estimated based on the load model discussed in Section 4.3.1. While ZC requires less effort
Table 4.4: Data distribution strategies. RR stands for round-robin, and mimics random distribution. GP stands for graph partitioning. Z stands for the distributions based on the zipcode only. ZC stands for the one based on the zipcode and the cost. Both Z, ZC, and GP considers locality, and RR ignores it. Both ZC, and GP considers load balance. GP optimizes locality subject to load balance constraints. It exercises load balance by a heuristic. ZC optimizes load balance subject to locality constraints. Both Z and ZC approximate the optimal locality by the zipcode and the object id. Although RR does not apply explicit load balancing, it makes an indirect attempt by randomizing the load assignment.

in data preparation compared to GP, there are disadvantages of the scheme. In ZC, the order of objects is strictly preserved to avoid the complication in addressing objects among the processors. It builds a map to match a range of zipcodes to each processor for forwarding each person and visit schedule to the processor where the associated home location is. To balance the load, ZC may slide a window of processor affinity over the ordered list of objects, but cannot shuffle objects among processors violating the order. On the other hand, GP does not maintain such an order, and is free to assign objects to any processor given the load balancing constraints. In addition, co-locating homes and people may be optimal for the locality of home locations but not necessarily so for non-home locations as we distribute different types of locations separately and attach people to their homes. In this sense, this approach optimizes load balance given a locality constraint. Figure 4.9 and Figure 4.10 respectively show the distribution of estimated computational load on home and non-home locations. The size of a dot is proportional to the load on the geographic location over grid with the resolution of 0.01°. The line connects successive locations in the sequence sorted by the zipcode and the id. The line serves as a space-filling curve [77], and is inspired by the work shown in [78].

Throughout the rest of the dissertation, we use the labels introduced in this section, RR, Z, GP, and ZC, for the distribution method used with or without -splitLoc to indicate whether the data have been decomposed as discussed in Section 4.3.4. Table 4.4 summarizes the data distribution strategies.

<table>
<thead>
<tr>
<th></th>
<th>RR</th>
<th>Z</th>
<th>ZC</th>
<th>GP</th>
</tr>
</thead>
<tbody>
<tr>
<td>Locality</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Load balance</td>
<td></td>
<td>locality</td>
<td>load balance</td>
<td>locality</td>
</tr>
<tr>
<td>Constraint Optimization</td>
<td>locality</td>
<td>load balance</td>
<td>locality</td>
<td>load balance</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>RR</th>
<th>Z</th>
<th>ZC</th>
<th>GP</th>
</tr>
</thead>
<tbody>
<tr>
<td>Locality</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Load balance</td>
<td></td>
<td>locality</td>
<td>load balance</td>
<td>locality</td>
</tr>
<tr>
<td>Constraint Optimization</td>
<td>locality</td>
<td>load balance</td>
<td>locality</td>
<td>load balance</td>
</tr>
</tbody>
</table>
Figure 4.9: Distribution of computational load on home locations. The line connects successive locations in the sequence ordered by the zipcode and the id. The size of a dot is proportional to the load on the location.
Figure 4.10: Distribution of computational load on non-home locations. The distribution spreads over a larger area than that of home shown in Figure 4.9 as people near state borders may visit locations across the borders.
4.3.3 Scalability Bottleneck of the Population Data

Among the four methods discussed in Section 4.3.2, we observe GP to perform the best as we discuss in Section 5.2.2. We rely on this approach to investigate the scalability with our data. Partitioning our population data sets in bipartite graphs, which represents the dual-phase computation discussed in Section 4.1, requires a special mechanism called multi-constrained partitioning [79, 80]. We use METIS for our investigation as it supports such a mechanism [79], allowing us to assign a vector of weights to each vertex. Each element of the vector is associated with a unique load balancing constraint for a specific phase of the computation. The objective of partitioning here is to minimize the communication between the phases subject to the load balance constraints for both phases. We define the weight of an edge by the maximum number of messages sent over the edge per iteration. However, this does not exactly model the cost of communication as the person phase computation overlaps the transmission delay, and we apply a message aggregation scheme. In reality, handling received messages adds the execution time on the critical path. We do not consider it for the analysis of data distribution in this chapter, but our experiments in Chapter 5 display it in Figure 5.9 and 5.14.

Our graph partitioning attempts fail to satisfy the balancing constraint of the second phase due to the significant variance in the load limiting the scalability as we discuss in Section 4.2.2. The left plots in Figure 4.11 show the estimated speedup, \( \frac{L_{\text{tot}}}{L_{\text{max}}} \), for location computation based on the load distribution across data partitions as a result of graph partitioning. This presents a tighter bound on the speedup for location computation than \( S_{ub} \), which we compute as \( \frac{L_{\text{tot}}}{l_{\text{max}}} \), by incorporating the load balancing achieved by graph partitioning. However, the effect of communication and the scaling of the person phase are not taken into account here. Table 4.5 shows that the graph partitioner achieves load balance close to the optimal \( S_{ub} \). Especially, it does so on average 99.999% of \( S_{ub} \) for the cases without the domain decomposition which we discuss in Section 4.3.4. Due to the extreme variance in the load, the load balancing quality of graph partitioner is not apparent. Section 4.3.5 discusses another bottleneck exposed as the domain decomposition mitigates the extreme variance. Although Figure 4.6 shows a similar load distribution in log-scale between IA and AR, \( t_{\text{max}} \) of AR is more than twice of that of IA, as shown in Table 4.5.

The left plots in Figure 4.12 show that the scalability \( \frac{S_{\text{max}}}{D} \) is reduced as the data size increases. Here, \( S_{\text{max}} \) represents the maximum of \( \frac{L_{\text{tot}}}{L_{\text{max}}} \) as a result of graph partitioning for various numbers of partitions up to 192K. In our heavy-tailed population data, the peak load grows as the graph size increases similarly in power-law graphs. Section 4.3.4 describes our decomposition strategy to break the heavy-tailed structure in our graphs.
Figure 4.11: The estimated speedup \( \frac{L_{tot}}{L_{max}} \) for location computation based on the load distribution across data partitions as a result of graph partitioning. This presents a tighter bound on speedup than \( S_{ub} = \frac{L_{tot}}{L_{max}} \) by incorporating the load balancing achieved by graph partitioning. It is evaluated for each of seven states over various number of partitions between 12 and 196,608. The left plots show the results with the original graph. The right plots show those with the graph modified as we discuss in Section 4.3.4.
Figure 4.12: The maximum of the estimated speedup shown in Figure 4.11. The left plots show the results with the original graph. The right plots show those with the graph modified as we discuss in Section 4.3.4. Each dot represents one of 48 contiguous US states and DC.
Table 4.5: The total load $L_{tot}$ and the maximum load per location before ($l_{max}$) and after ($\ell_{max}$) graph modification.

<table>
<thead>
<tr>
<th>Machine</th>
<th>load</th>
<th>CA</th>
<th>NY</th>
<th>MI</th>
<th>NC</th>
<th>IA</th>
<th>AR</th>
<th>WY</th>
</tr>
</thead>
<tbody>
<tr>
<td>Blue Waters</td>
<td>$L_{tot}$</td>
<td>545,577,441</td>
<td>282,939,700</td>
<td>151,205,758</td>
<td>135,179,837</td>
<td>43,949,221</td>
<td>42,318,671</td>
<td>7,818,010</td>
</tr>
<tr>
<td></td>
<td>$l_{max}$</td>
<td>254,931</td>
<td>347,461</td>
<td>234,065</td>
<td>160,397</td>
<td>45,246</td>
<td>97,323</td>
<td>32,566</td>
</tr>
<tr>
<td></td>
<td>$\ell_{max}$</td>
<td>2,906</td>
<td>2,005</td>
<td>2,182</td>
<td>2,012</td>
<td>1,894</td>
<td>1,738</td>
<td>1,561</td>
</tr>
<tr>
<td>Vulcan</td>
<td>$L_{tot}$</td>
<td>731,040,603</td>
<td>386,336,106</td>
<td>206,854,968</td>
<td>184,853,337</td>
<td>59,819,788</td>
<td>57,896,250</td>
<td>10,660,897</td>
</tr>
<tr>
<td></td>
<td>$l_{max}$</td>
<td>354,355</td>
<td>495,456</td>
<td>325,827</td>
<td>223,418</td>
<td>64,414</td>
<td>135,363</td>
<td>45,135</td>
</tr>
<tr>
<td></td>
<td>$\ell_{max}$</td>
<td>3,167</td>
<td>2,291</td>
<td>2,394</td>
<td>2,453</td>
<td>2,326</td>
<td>2,279</td>
<td>2,235</td>
</tr>
<tr>
<td>Sierra</td>
<td>$L_{tot}$</td>
<td>287,343,148</td>
<td>146,001,323</td>
<td>78,026,045</td>
<td>69,696,605</td>
<td>22,774,604</td>
<td>21,798,781</td>
<td>4,003,918</td>
</tr>
<tr>
<td></td>
<td>$l_{max}$</td>
<td>152,976</td>
<td>208,503</td>
<td>140,454</td>
<td>96,246</td>
<td>27,144</td>
<td>58,395</td>
<td>19,535</td>
</tr>
<tr>
<td></td>
<td>$\ell_{max}$</td>
<td>1,736</td>
<td>1,195</td>
<td>1,301</td>
<td>1,199</td>
<td>1,129</td>
<td>1,035</td>
<td>929</td>
</tr>
</tbody>
</table>

Figure 4.13: Two methods to split a heavily loaded node, exploiting extra parallelism available. A heavy node is split into a pair, nodes 1 and 14, dividing the load. In addition, the split pair may (a) or may not (b) split the set of edges connected to the original node, depending on how the computation depends on the data.

4.3.4 Split Heavily-loaded Nodes

We develop a domain decomposition strategy to preprocess a graph before applying an existing distribution method and to prevent the structure of graph from being the major bottleneck of scalability. During preprocessing, we modify the graph, taking advantage of unexploited extra parallelism in the application to split the work units (vertices) with the highest loads. For example, node 1 in Figure 4.2(a) can be split into two: node 1 and node 14 shown in Figure 4.13. Ideally, the split work units have no overlapping dependency between them as in Figure 4.13(a). This not only splits the workload, but also divides the communication, which helps reduce the maximum of both load and degree of vertices. Depending on the dependency pattern in the application, a split work unit can either retain the entire set of edges as in Figure 4.13(b) or require additional
communication within the split pair.

In EPI SIM DEMICS, the interaction between people defines the dependence. People only interact when they are present in the same sublocation. This allows us to split locations without adding extra communication edges as shown in Figure 4.13(a). We split a heavy location into multiple locations, each of which contains an exclusive subset of sublocations of the original location. Heavy locations are split, instead of being distributed by sublocation to keep the size of the graph minimal for scalability and partitioning efficiency, and to efficiently maintain shared data. In the future, we plan to model inter-sublocation mixing within a location, such as elevators and hallways, as in Figure 4.13(b). Even in such a case, we can split the load by dividing the susceptibles while replicating the infectious in the redundant sublocation.

To determine how heavy a location is, we rely on a platform-independent approximation instead of the platform-dependent load model discussed in Section 4.3.1 and 4.3.2. This approximation is used to split locations in order to bound the location load while avoiding the need to collect model input data at the sublocation level. We first define a weight for each sublocation type. There are currently five sublocation types used in our population data: home, work, school, shop, and other. Then, we add up the sublocation weights in the location. The sublocation weight is defined as the average number of visits to the sublocation. As we are interested in the heaviest locations, we determine the sublocation weight based on the largest locations from each state in terms of the number of sublocations. Then, we divide locations heavier than a threshold as evenly as possible.

We choose a threshold \( t \) based on the total load \( L \) in the graph, the maximum number of partitions to use \( P \), and the largest weight of a sublocation \( m \). We approximate \( L \) and \( m \) based on the aforementioned platform-independent estimation. Then, we can set the threshold to \( t = \max(L/P/k, m) \) for an arbitrary \( k \) where \( k > 0 \), or to a fixed constant independent of \( L \) and \( P \). Figure 4.14 shows \( S_{ub} = \frac{L_{tot}}{l_{max}} \) as a result of various decomposition thresholds. As \( k \) becomes large, \( S_{ub} \) converges. The threshold with \( k=1 \) fails to achieve the optimal \( S_{ub} \) for TX due to the sub-optimal platform-independent estimation. The primary purpose of the decomposition is to modify a graph so that it is more amenable for load balancing while partitioning rather than balance load directly. Throughout the dissertation, we choose the fixed \( m \) as the threshold unless otherwise noted, although it is possible to choose a higher threshold than \( m \) to achieve a similar \( S_{ub} \) for certain data. A higher threshold would require splitting a less number of locations.

Finally, we apply a dynamic programming similar to the method discussed in [81] for partitioning a set of weights and evenly dividing a location into multiple locations in terms of weights. As a result of applying \( m \) as the threshold, the distribution of degree and the load in the graphs changes as in Figure 4.5 and Figure 4.6. The estimated speedup improves as shown in Figure 4.11. \( \frac{L_{tot}}{l_{max}} \) increases by a factor of, on average 89 (maximum 290, minimum 11) over 48 contiguous states and
Figure 4.14: Impact of decomposition threshold on the estimated speedup with Texas, New York, and Michigan data sets. We rely on a platform-independent load approximation for the decomposition and on the load model for Blue Waters (see Section 4.3.1) for the speedup estimation. Original represents the case without decomposition. Absolute represents the case with threshold fixed to $m$, the largest weight of a sublocation. Otherwise, we set the threshold as $t = \max(L/P/k, m)$ where $L$ is the sum of sublocation costs in the data, $P$ is the maximum number of partitions set to 192K. $S_{ub}$ represents the upper bound on speedup. $S_{max}$ represents the maximum of the estimated speedup as a result of graph partitioning for various numbers of partitions up to 192K.

DC. Figure 4.12(b) shows the resultant improvement on $S_{max}/D$. The modification reduces $d_{max}$ by a factor of, on average, 54 (maximum 341, minimum 12), while improving $S_{ub}$ by 84% on average and increasing $D$ by 5.25% at most.

4.3.5 Trade-off between Locality and Load Balancing in Graph Partitioning

Figure 4.14 also shows $S_{max}$, which represents the maximum of $L_{tot}/L_{max}$ as a result of graph partitioning for various numbers of partitions up to 192K. As $k$ becomes large, $S_{max}$ approaches to $S_{ub}$. For all data sets except NY and CA, $S_{max}$ reaches up to 99% of $S_{ub}$ with $P=192K$. For CA, $S_{max}$ scale further up to a larger $P$ than 192K as shown in Figure 4.11. For NY, $S_{max}$ reaches only 76% of $S_{ub}$ with $P=192K$ and 99.6% with $P=384K$. However, even for a larger data set TX, $S_{max}$ reaches 99.9% with $P=192K$. The location set of TX is 8% larger and the population size is 22% larger than those of NY. This requires further study on how the graph partitioner handles two different optimization targets, which are in a trade-off relationship. In addition, how the structural properties of graphs other than the degree, such as clustering, affects the performance is another interesting subject to pursue.

To see if the connectivity in graphs affects the load balancing quality of graph partitioning, we
Figure 4.15: The load balancing with the NY data decomposed as the case Absolute in Figure 4.14, improves as we relax the connectivity in the graph for partitioning. $MX_n$ represents the case where any vertex that has more than $n$ edges drops randomly chosen edges and leaves $n$ edges. $DA_n$ represents the case where any vertex that has more than $n$ edges drops all the edges. Full represents the case where no edge is dropped.

Drop some of the edges in the graph of NY data and repartition it. Note that this does not remove the actual communication in population data. It simply removes the edges representing the communication only for graph partitioning purposes. After repartitioning, we evaluate the estimated speedup based on the load distribution. Then, we compare the estimated speedup to the upper bound as shown in Figure 4.15. $DA_{350}$ achieves the best estimated speedup with an improvement by 6.2% compared to Full. The intuition and the observation are as follows. $DA_{350}$ leads to better load distribution than $MX_{350}$ as it is completely free for graph partitioners to relocate heavily connected vertices. However, we observe that the quality degrades when removing all the edges in the entire graph or excessively removing the edges of vertices, which have only one edge. $MX_{10}$ does not achieve a result as good as $MX_{350}$ possibly because a graph partitioner has a limited number of connected partitions to choose from when assigning a vertex with a limited connectivity. This analysis requires a further investigation to make sure if the sub-optimality results from the heuristics of the graph partitioner or an artifact of the implementation.

A further question is if modeling the trade-off relationship between the benefit of local accesses and the penalty of unbalanced load is feasible. For this, we need to model the cost of non-local accesses. However, how much cost a non-local access would add in the critical path is not easily identifiable. Especially, it is challenging to analyze or model such a cost when considering various schemes to utilize high bandwidth and to hide access latency—e.g., message aggregation and routing schemes and the non-blocking communications. Nonetheless, we can focus on the message handling cost at the source and the destination of which execution is on the critical path assuming that the entire transmission overlaps the computation. We plan to study this in the future.
4.4 Related Work

Among many agent-based epidemiological platforms are those developed by Eubank et al. [82, 83], Longini et al. [84], Ferguson et al. [85], and Parker et al. [86]. The system described in [85] is implemented for shared memory platforms and, thus, is limited by the amount of available shared memory. The works in [84] and [86] either use structured social contact networks, that are more amenable to efficient parallel computation but arguably less representative of real-world social networks, or lack the rich set of interventions required to accurately model real-world responses to pandemics. In any case, none of these have been shown to scale to more than a thousand or so cores. In addition, the Global-Scale Agent Model (GSAM) [86] is implemented in Java.

The Parallel Discrete Event Simulation (PDES) formulation by Perumalla et al. [87] has the best scaling of any individual level epidemiology simulation known to the authors, and is based on previous EpiSIMDEMICs work [62]. The system uses the same disease model and transmission function, but is based on a hierarchical social network construction similar to [84], a population of homogeneous agents, and lacks the ability to model interventions. Grefenstette et al. [88] implement an agent-based framework, FRED, which employs a census-based synthetic population, and supports scenario interventions. In the population model, each agent is associated with demographic and socioeconomic information, and its activities are tied to specific locations. It is implemented in OpenMP, and demonstrated the simulation of the entire US population using 16 threads on a shared memory platform (SGI Altix UV). The Global Epidemic and Mobility framework (GLEaM), by Broeck et. al, aims to simulate a geo-referenced mobile meta-population considering long-distance travels at the global scale that consist of 3,362 sub-populations in 220 countries, and to provide a user-friendly interface [89]. The simulation engine is based on the client-server model written in C++, which handles multiple runs simultaneously. FlueTE, by Chao et. al, is an individual-based simulation framework and supports intervention studies. However, the interaction in the population is organized by hierarchical social mixing groups [90]. The parallel version of the implementation relies on MPI, and demonstrated the simulation of the entire US population using 32 processors.

Traditional graph partitioning tools primarily focus on minimizing total edge cuts while enforcing load balance as a constraint rather than an optimization target [79, 91, 92]. Such partitioning methods have been most successful in areas as mesh-based PDE simulation, VLSI layout design, and sparse matrix decomposition [92, 93]. Although minimizing the total edge cuts limits the maximum edge cuts per partition, these tools do not balance edge cuts across partitions, which is also important for minimizing communication cost.

Partitioning extremely large, highly irregular data is left as an open problem. Abou-Rjeili et al. [94] propose a new clustering-based coarsening scheme for power-law graphs, which identifies and
collapses groups of vertices that are highly connected. The method described in [95] divides the vertices of a graph into $k$ almost equal groups such that the sum of the weight of the edges connecting vertices in different partitions is minimized. Pearce et al. [96] evenly divide sorted edge lists to reduce communication hotspots while accommodating high degree vertices over multiple partitions using ghosts. Gonzalez et al. propose the vertex-cut approach to distribute the computation of a vertex with a high degree for dividing edges evenly across partitions [97]. Our approach is different from these as we enhance application-specific load balancing with locality optimization while the existing works focus on locality. We split nodes with heavy computation and apply application-specific load balancing for locality optimization. Our approach may further take advantage of vertex-cut or ghosts for modeling inter-sublocation mixing within a decomposed location vertex while splitting its heavy load for allowing better load balancing.

While data from large scale socio-technical systems enable interesting studies [83, 98–102], social network graphs from the real world exhibit scale-free property following heavy-tailed degree distributions [103–106]. The structural properties of such graphs—for example, degree, diameter, and clustering—have been studied extensively [107–109]. However, there is little understanding on how such a property affects the scalability of applications. The examples of applications processing social network graphs include those that simulate various diffusion phenomena [74, 110, 111] based on graph dynamical systems (GDS) [112–115], analyze particular aspects of data, and calculate useful metrics of the graphs [98, 100, 116–118]. A number of general graph processing frameworks have recently been introduced to help with implementing applications for processing large scale graph data [117–123]. However, to what degree the performance of these frameworks can scale up is still unknown. To enable high scalability, these frameworks must expose the explicit control of locality and allow application-specific load balancing. This requires accurately modeling the cost of the vertex computation or the local function [74, 112].

In a GDS, messages are sent to immediate neighbors only [112, 124]. EPI$S$IMDEMICS implements a GDS, and the cost of the local function of a location type vertex is closely related to the number of messages received, which depends on the sum of in-edge weights. On the other hand, graph processing often requires multi-hop communication. For example, betweenness centrality is one of the common metrics computed in social network analysis [116, 125]. Goh et al. [126] show that the load distribution in scale-free data transport networks, in which every pair of nodes exchanges a packet via the shortest path, also follows a power law. The load, betweenness centrality in this case, is defined as the total number of messages passing through a vertex. Interestingly, they claim that the power-law exponent of the load distribution is approximately 2.2 insensitive to the power-law exponent $\gamma$ of the degree distribution as long as $2 < \gamma \leq 3$. Our analysis shows that the scalability with such a graph and the load defined as in Section 4.2.2 is the lowest when the exponent is around 2.2, as shown in Figure 4.3. Subgraph matching is one of the key operations
in graph processing [127]. Sun et al. compare a total of eleven implementations of five different subgraph matching algorithms in terms of space and time complexity as a function of the number of nodes and edges, and the average degree [127]. The time complexities of indexing in some of these implementations depend on the total number of edges or on the average degree multiplied by the number of nodes. This suggests that fine-grain parallelization of the works may expose the time complexity of an individual vertex as an order of degree.

4.5 Summary

Contagion simulations play an important role in understanding and combating epidemics. Modeling national and multinational regions with increasing accuracy, while adhering to strict deadlines, requires large computing resources. The extreme irregularity in the underlying graph data makes scaling contagion simulations challenging. In this chapter, we present advances in processing large graph data, especially those with heavy-tailed properties, that allow us to significantly improve the performance and scalability of EPI SIM DEMICS. These include domain decomposition and effective partitioning of the person-location graph to minimize the impact of the heavy-tailed load distribution of the graph data, as well as the communication to access non-local data. These methods are motivated by the insight gained from our analysis of the impact of power-law graphs on the scalability. The application-specific load model is crucial for this accomplishment. In addition, we introduce an application-specific locality optimization method, which requires less effort than the general state-of-the-art graph partitioning for assigning data to processors. We experimentally evaluate this in Section 5.2.2.
Chapter 5

Pushing the Limits of the Scalability of Social-network-based Simulations

In this chapter, we present the implementation and evaluation of a series of optimizations for irregular graph-processing applications, including message aggregation and low-level code optimizations. These optimizations, combined with the application-specific data decomposition and distribution discussed in Chapter 4, enables us to achieve an unprecedented strong scaling of a contagion simulation on over 352K cores of Blue Waters at the National Center for Supercomputing Applications (NCSA), one of the largest HPC systems in the world, as well as 128K cores of Vulcan at Lawrence Livermore National Laboratory (LLNL).

5.1 Communication Optimizations

EPI SIMDEMICS is a data-intensive simulation framework of contagion over social contact networks. It requires high memory and communication bandwidth while less sensitive to the latency. We optimize the communication by exploiting hierarchical multi-core nodes, aggregating small messages to increase bandwidth utilization and to reduce the overhead and size of messages, and using the advanced synchronization for irregular communication.

5.1.1 Message-driven Design

EPI SIMDEMICS was originally implemented in C++ and MPI. However, this model is not well suited for applications with such dynamic behavior, in this case caused by the arbitrary number of visit messages in each step. Therefore, the current version of the code was re-implemented in a
parallel language called CHARM++ [67, 128], which is a C++-based parallel programming model accompanied by a message-driven asynchronous runtime. It is portable and highly scalable and has shown significant performance benefits for applications that have dynamic computational and communication load imbalance and can benefit from the adaptive overlap of computation and communication [129–131]. The underlying idea is to over-decompose the computation in the application into smaller units called *chares*, i.e., into significantly more units than available physical processors, and to let the runtime then assign a set of work units to each physical processor, which enables a fine grained load balancing. Chares can either be data units, work units, or both. However, implementations must choose the right granularity of splitting work into chares to find the right tradeoff. A large number of chares, each with little work increases flexibility, but also results in higher overhead, while a small number of larger chares minimizes overhead but limits the ability to exploit over-decomposition.

Chares are migratable and the runtime provides a sophisticated load balancing framework with measurement-based and model-based load balancing strategies [132]. The presence of multiple chares on each processor also facilities adaptive overlap of computation and communication. However, the data decomposition into the appropriate number of chares is important to get a good starting distribution and to find the right tradeoff between too little work per chare (too much overhead) and too much work per chare (not enough opportunity for over decomposition).

We follow a two-level hierarchical data distribution technique to find the right tradeoff, as shown in Figure 5.1. At the first level, we create two types of chares, LocationManagers (LM) and PersonManagers (PM), each able to manage multiple second level objects representing individual locations and persons, respectively. We then distribute the person and location objects among the elements of the corresponding *chare arrays* (LM and PM). The individual chares in both arrays handle the computation and communication of all location or person objects assigned to them, respectively. This two-level hierarchy helps to efficiently manage an extremely large number of objects. The CHARM++ runtime then maps the chare arrays (representing LMs and PMs) to processes. As a consequence, different object to manager and manager to processor mappings can result in different communication patterns with potentially varying degrees of efficiency.

### 5.1.2 CHARM++ ++ SMP Mode

Modern supercomputers have multiple cores per node. Moreover, moving towards exascale, there is a trend towards increasing the number of cores per node [133]. To achieve good scalability with modern supercomputers, where there are multiple cores per node, an application needs to exploit the benefits including inter-processor communication and shared data within a node, and overcome communication bottlenecks of multi-core nodes. To achieve these goals, we leverage
Figure 5.1: EpiSIMDEMCIS implemented in CHARM++. This conforms to the algorithm for a simulation iteration which consists of two computation phases, person phase and location phase, followed by a global state reduction, described in Figure 4.1. We create a chare array of each type, LocationManager (LM) and PersonManager (PM). We distribute location objects among the LM elements and person objects among the PM elements. The CHARM++ runtime then assigns a set of chares to each physical processor.
CHARM++’s symmetric multiprocessing (SMP) machine layer [134, 135], which instead of creating one OS process per core of an \( n \) core node, the runtime creates \( k \) OS processes per node, such that \( k < n \) and \( n/k \) is an integer. This allows shares within a process to leverage more efficient intra-node communication via shared memory for the following benefits:

1. inter-thread communication can be implemented with direct memory copy
2. the communication thread minimizes the interference between application compute functions and communication
3. sharing of read-only data across all threads reduces memory consumption

To enable this mode, CHARM++ spawns a separate communication thread in each of the \( k \) OS processes in addition to the \( n/k - 1 \) compute threads and then maps each thread to a separate core. The disadvantage of this approach is that it reduces the number of compute threads per node, since \( k \) cores are used as communication threads. However, the communication intensive nature of EPISIMDEMICS significantly benefits from the availability of a dedicated communication thread to offload messaging, leading to an overall increase in performance. The requirement for an application to take advantage of the CHARM++ SMP mode is the thread-safety. The CHARM++ API provides application programmers with an interface to locking mechanism to avoid a potential race condition. In addition, CHARM++ ++ supports a hierarchy of data sharing among chare objects. An object of group chare type is private to a processor and shared among the chare objects on the same processor. An object of node group chare type is visible within an address space and referred as an SMP node.

5.1.3 Synchronization with Irregular Communication

As discussed in Section 4.1, there is a need for global synchronization at the end of each phase of the simulation. After each person has sent its visit messages to locations, a global barrier must be enforced before locations can start computing infections. However, since the individual location chares have no prior knowledge of how many messages to receive and from whom, a simple barrier is not adequate. Instead, we need a mechanism to detect the condition when there are no messages awaiting processing or in transit.

For this purpose, CHARM++ provides two mechanisms Quiescence Detection (QD) and Completion Detection (CD). We implement EPISIMDEMICS such that it can switch between the two at compile time. QD requires global quiescence across the entire application [136]. On the other hand, CD can be applied to subsets of chares as long as the number of candidate producers is known a
Figure 5.2: Two AMD Opteron Interlagos processors of a XE6 node in Blue Waters. In CHARM++ SMP mode, we choose to assign a CHARM++ runtime process on core 0 of each core module (CM) and a worker thread to each of cores 2, 4, and 6 on each CM as we discuss in Section 5.2.2.

priori, which is given in our case. Completion is detected when the participating objects have produced and consumed an equal number of messages globally. In the future, EPI SIMDEMICS will be extended to perform multiple simulations simultaneously, using dynamic replication of state (chare arrays). This will require an approach that enables us to perform synchronization local to a module.

Both QD and CD create very little interference to application execution because the detection protocol ensures that they are initiated by idle processors.

5.1.4 Aggregating Fine-grained Messages

With the entire US population data, EPI SIMDEMICS generates more than 1.5 billion messages per iteration, each of which is around 40 bytes in size. Message aggregation to cope with an enormous volume of fine-grained messages is crucial for EPI SIMDEMICS to achieve good performance. EPI SIMDEMICS can choose between two message aggregation methods at compile time. One is manually implemented in the application, which refer as manual buffering, the other is offered by CHARM++ as a library called TRAM.

*manual buffering* is used in PMs when sending visit messages to LMs, since this can lead to a large number of small messages between two chares, caused by people visiting locations. Without
message aggregation, we send each message upon generating it, by passing it as a parameter to the remote entry method on the destination chare. With our message aggregation, we do not send each individual message immediately after generation. Instead, we temporarily store messages in buffers organized per destination. Each source chare (PM) manages a set of buffers containing outgoing messages for a unique destination chare (LM).

However, this becomes ineffective when buffering space is exhausted or a receiver cannot keep up with an enormous volume of inbound messages. To cope with such challenges efficiently, manual buffering offers two flushing mechanisms controlled by runtime parameters. The per-buffer flushing ensures that a single buffer does not hold more than a user-defined number of outgoing messages. The space-wise flushing limits the memory footprint such that the buffer space of a chare does not grow beyond a user-defined size in terms of the total number of outgoing messages of a source chare.

Moreover, to avoid saturating network resources, message aggregation can be set to switch between sending and receiving regularly. Without this, the outstanding inbound messages remain in the message queue managed by the runtime system and may exhaust the queue space allocated by the runtime. When interleaving is not sufficient to avoid the resource saturation, we can further group multiple person managers and serialize message generation within a group. However, if possible, it is best for performance to generate all the messages without switching between sending and receiving.

Figure 5.3: Two aggregation schemes for fine-grained messages. (a) In manual buffering, each source (blue) maintains a set of buffers, one per destination. When a buffer is full, the entire buffer is packed and sent out as an aggregate message. (b) TRAM aggregates messages per peers in a virtual mesh topology. Users can specify the dimension of the virtual mesh at runtime. It maintains a buffer per destination peer on each processor.
The Topological Routing and Aggregation Module (TRAM) in CHARM++ is a new feature that optimizes fine-grained communication [137]. This module was not available for the experiments on Blue Waters, but it was available for those on the other platforms. With TRAM, messages are forwarded to the destination by intermediate nodes called peers along the shortest path in a virtual mesh topology. A peer is located at the junction from one dimension to the other on a route. The routing aspect of TRAM allows aggregating data items whose destinations are different, as long as their paths share the same intermediate destination. In addition, the number of peers for any given process is low. TRAM aggregates messages per peers in the virtual mesh topology rather than per destination as in manual buffering. Thus, the memory footprint for the buffer space is usually small enough to fit in lower level cache memory. However, unlike an application agnostic aggregation in the runtime, manual buffering allows the removal of redundant information in an aggregate message to reduce the message size. Figure 5.3 illustrates how the two buffering schemes operate. Further details of TRAM including the analytical and experimental aspect of its behavior and the comparison between the two aggregation approaches are discussed in [137].

5.2 Performance Results

5.2.1 Test Platforms

Table 5.1 lists the platforms at various scales where we performed our experiments. Blue Waters and Vulcan are petascale systems. Cab and Sierra are Infiniband-based commodity clusters. While Blue Waters is a heterogeneous system of XE6 and XK7, our experiments were confined to the 22,640 XE nodes. Each XE node has two AMD Interlagos processors, one Gemini NIC, where the injection bandwidth is 9.6 GB/s, connected in a 3D torus and 64 GB of memory with 102.4 GB/s bandwidth [138–140].

Figure 5.4 shows an XE6 blade, and Figure 5.5 shows the entire Blue Waters system. There are a total of 5,688 XE6 blades as well as 1,056 XK7 blades in the entire system. EPI SIMDEMICS especially benefits from the unusually large memory per node, high memory and communication bandwidth, and the large number of cores in Blue Waters. Each Interlagos processor consists of eight Bulldozer core modules, with two integer units and one floating point unit per module, resulting in 16 core modules per node or 362,240 for the entire XE machine [141]. Each core module has 2 MB L2 cache. It is possible to issue two threads per core module by issuing one thread per integer unit. However, we experimentally determined that EPI SIMDEMICS performs the best with one thread per core module.
A compute node in Vulcan has a Blue Gene/Q compute chip (BQC), which has eighteen processor units. The unit is based on the PowerPC A2 core with specific additions such as a SIMD quad floating-point unit, and a Level-1 prefetching module (L1P), and a wake-up unit [60]. The eighteen cores share a total of 32 MB L2 cache slices and ten bidirectional links (1.8 GB/s for user data per direction) to neighboring chips in a 5D torus [142]. Among the eighteen cores, sixteen are available to users, one is reserved for OS, and the other is unused. While each core supports four hardware threads, we run a single thread per core. QLogic QDR offers 4 GB/s for data in Cab and Sierra [143].

For all the experiments, we fix the processor affinity of CHARM++ processes and threads to prevent OS from moving them. We perform experiments up to the limit of accessible resources on each platform. On Blue Waters, we use CHARM++ version 6.4, built for gemini_gni-craxyxe, hugepages (8MB) and smp, using GNU programming environment 4.1.40. The Cray balanced injection feature was not enabled. On the other platforms, we use CHARM++ version 6.5. On Vulcan, we built CHARM++ runtime for pamilrts-bluegeneq and smp, using gcc 4.4.7. On Cab and Sierra, we built CHARM++ runtime for mpi-linux-x86_64 and smp, using icc 12.1.5 and MVAPICH2 MPI v1.7.
Figure 5.5: NCSA Blue Waters, pictured on October 22, 2013. There are total 5,688 XE6 blades, shown in Figure 5.4, as well as 1,056 XK7 blades in the entire Blue Waters system.

<table>
<thead>
<tr>
<th>Name</th>
<th>Blue Waters</th>
<th>Vulcan</th>
<th>Cab</th>
<th>Sierra</th>
</tr>
</thead>
<tbody>
<tr>
<td>Manufacturer</td>
<td>Cray</td>
<td>IBM Blue Gene/Q</td>
<td>Cray X-treme X</td>
<td>Dell Xanadu 3</td>
</tr>
<tr>
<td>Machine</td>
<td>XE6/XK7 NCSA</td>
<td>LLNL</td>
<td>LLNL</td>
<td>LLNL</td>
</tr>
<tr>
<td>Site</td>
<td>NCSA</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CPU</td>
<td>Opteron 6276 2.3 GHz (Interlagos)</td>
<td>Power BQC 1.6 GHz</td>
<td>Xeon E5-2670 2.6 GHz (SandyBridge)</td>
<td>Xeon X5660 2.8 GHz (Westmere)</td>
</tr>
<tr>
<td>Cores/Node</td>
<td>16 Core Modules (CM)</td>
<td>16</td>
<td>16</td>
<td>12</td>
</tr>
<tr>
<td>Total Cores</td>
<td>XE 362,240 CM XK 33,792 CM 4,224 GK110</td>
<td>393,216</td>
<td>20,480</td>
<td>21,756</td>
</tr>
<tr>
<td>Mem/Node</td>
<td>64 GB</td>
<td>16 GB</td>
<td>32 GB</td>
<td>24 GB</td>
</tr>
<tr>
<td>BW/Node</td>
<td>102.4 GB/s</td>
<td>42.6 GB/s</td>
<td>51.2 GB/s</td>
<td>32 GB/s</td>
</tr>
<tr>
<td>Interconnect</td>
<td>3D torus</td>
<td>5D torus</td>
<td>QDR</td>
<td>QDR</td>
</tr>
<tr>
<td>BW/Node</td>
<td>9.6 GB/s</td>
<td>20 GB/s</td>
<td>5 GB/s</td>
<td>5 GB/s</td>
</tr>
<tr>
<td>Rmax (PF/s)</td>
<td>NA</td>
<td>4.29</td>
<td>0.3474</td>
<td>0.1667</td>
</tr>
<tr>
<td>Rpeak (PF/s)</td>
<td>13.13</td>
<td>5.03</td>
<td>0.4260</td>
<td>0.2437</td>
</tr>
<tr>
<td>Rank</td>
<td>NA</td>
<td>9</td>
<td>94</td>
<td>263</td>
</tr>
</tbody>
</table>

Table 5.1: Platforms used. Blue Waters and Vulcan are petascale systems. (*)www.top500.org, Nov 2013.)
5.2.2 Scaling Analysis

Uncovering Layers of Scalability Bottleneck

The performance of the initial CHARM++ implementation of EPI SIM DEMICS is shown as unoptimized in Figure 5.6(a), which does not scale beyond a thousand cores. Our first attempt was to remove inefficiencies in the code as much as possible using techniques such as branch reduction, loop unrolling, inlining as well as algorithmic improvements for data locality. We further reduce the memory footprint to roughly 50% by redesigning data structures and message types. The optimized curve and the ratio in Figure 5.6 shows the outcome of such efforts in strong scaling tests on Blue Waters for processing NC data. Ratio shows the ratio of the hardware counter stats [144] of optimized to that of unoptimized. Data accesses are reduced by 20% as shown by the ratio of L1-D accesses. The number of branch operations as well as the mispredictions of conditionals are reduced by 20% - 50%. The L2 miss ratio and TLB misses are reduced respectively by as large as 40% and 60%. As the number of nodes used increases, the reduction decreases as well as the amount of data per node. The scaling is further improved by taking advantage of SMP as discussed in Section 5.2.2. However, Figure 5.6(a) suggests that this is not sufficient and we are discovering another bottleneck.

EPI SIM DEMICS is a highly data-intensive application in the sense that there is little floating point computation. Instead, as shown in Figure 5.7(a), there are large portions of memory access and branch instructions, which contributes to about 58% and 19% of the total instructions, respectively.
70% of the latter are conditionals. Figure 5.7(b) shows that the misprediction ratio of conditional branches is 3.1% and the miss ratio of L2 data caches is 11.8% on average. We collect the hardware counters during processing the US data on Blue Waters excluding initial data loading. In addition, the person phase and the location phase takes about 34% and 64% of execution time. The reduction of scenario variables takes up the rest of the execution time.

CHARM++ Runtime and Optimizations

EPI SimDEMICS benefits from a system configuration that handles intensive network I/O and avoids memory access contention, while utilizing all core modules to achieve the best scaling. There are a total of four NUMA regions per XE6 compute node and four core modules per region. We identified the optimal configuration, which has one process per NUMA domain and one thread per core module (three worker threads per process), based on experimental results shown in Figure 5.8. While the memory access cost is uniform among cores on Vulcan, we also observed the same configuration performs the best on Vulcan balancing the load between a communication thread and the worker threads. Additionally, except on Blue Waters, we exclude the processor of rank 0 for application’s major workload such that its resource is dedicated to handling global reduction.

Application-specific Data Partitioning

Figure 5.9 shows a breakdown of execution time for each core. The timeline plots show 25th iterations of EPI SimDEMICS, while running MI data distributed and decomposed with seven
different methods, discussed in Section 4.3.2 and 4.3.4, over 4K core modules (3072 cores). Due to space limitations, we plot the results for a uniformly sampled subset of 332 compute cores. Each bar represents the activity of a processor over time progressing from left to right. Both RR and GP show extreme variance in location load distribution. Figure 4.6(b) shows how the distribution of load per work unit is skewed without decomposition. GP improves data locality, leading to less message exchange and less interruption to message generation than RR. RR-splitLoc, ZC-splitLoc and GP-splitLoc shows significant improvement over both RR and GP by reducing the load granularity to allow better load balance. However, Z-splitLoc does not attempt to balance load. This results in higher location load on certain PEs and extra interference for receiving more messages (see Figure 5.10) which creates variation in person computation. The time above the plot shows the time taken for the iteration while running the CHARM++ Projections visual analysis tool.

Figure 5.10 shows the number of bytes received per core from 25th iteration to 27th from the same tracing runs shown in Figure 5.9. With RR, there is a peak which almost reaches 11 MB. This occurs because the particular processor has locations receiving a large number of visits without decomposition. Similarly with Z-splitLoc, there is a single PE received as large as 7 MB. However, the maximum bytes sent out per PE was around 5 MB in this case. RR-splitLoc removes such peaks. GP-splitLoc and ZC-splitLoc further improve locality and reduce the overall volume of communication. While GP significantly reduces the volume of communication compared to that of RR, the largest peak almost reaches 8 MB. The graph partitioner does not balance communication evenly, but only minimizes a total volume [145]. Figure 5.11 shows the imbalance in the distribution of edges. Each data point represents the maximum per-partition edge cuts as a result of graph partitioning for Vulcan. The line of matching orders represents a reference case of the ideally
Figure 5.9: Timeline plots of an iteration of EpiSimdemics using MI data and 4K cores on Vulcan (sampled to show 332 PEs). The blue and orange components show the person and location computation, respectively. The red represents the message handling of TRAM (see Section 5.1.4 and Figure 5.10). The time taken for the iteration is shown above each plot.
balanced all-remote-communication, which is calculated as the number of total edges divided by the number of partitions hypothetically imagining that all edges are cut. With CA, the maximum per-partition edge cuts is 7.1 times larger than the all-remote-communication case with 98,303 data partitions. On the other hand, with NY data, the ratio is 3.3. The average ratio is 4.8 among the four large states, with which the maximum number of edges per location is smaller than the average per partition.

Figure 5.12 and 5.13 shows the strong scaling of EPI SIMDEMICS with state data listed in Table 4.1. These show the effectiveness of the decomposition discussed in Section 4.3.4, and match the general trends of the estimated cases shown in Figure 4.11. The decomposition significantly improves the performance with all dataset. Both GP-splitLoc and ZC-splitLoc improve locality and load balance. ZC-splitLoc performs similarly as GP-splitLoc does with less preparation effort as discussed in Section 4.3.2. Although Z-splitLoc does not optimize load balance, unbalanced load become problematic when distributed either over a very large or a very small number of processors. The peak speedups achieved by the actual runs without using decomposed data are on average 133% of that of the estimated speedup ($L_{\text{tot}}/L_{\text{max}}$), maximum 147%, and minimum 128% among the seven states’ data. This is due to the fact that the estimation does not include person phase computation. On the other hand, the peak speedups achieved using decomposed data are quite
Figure 5.11: The maximum per-partition edge cuts (GP-splitLoc). Each data point represents the maximum per-partition edge cuts as a result of graph partitioning for Vulcan. The line of matching orders represents a hypothetical reference case where all edges are cut but they are evenly distributed across partitions, which is essentially calculated as the number of total edges divided by the number of partitions.

low, on average 27%, maximum 55%, and minimum 17.6%. This suggests that there are further bottlenecks including the one shown in Figure 5.11.

A similar set of experiments conducted on Blue Waters show the results comparable to those we observe on Vulcan. Figure 5.14 shows the timeline plots of EPI SIMDEMICS while running MI data over 4K core modules (3072 cores) on Blue Waters for two iterations (25th and 26th). Due to space limitations, we present the results for a uniformly sampled subset of 384 cores. Both RR and GP show extreme variance in location load distribution without decomposition. GP improves data locality, leading to faster message generation than RR. RR-splitLoc as well as GP-splitLoc shows significant improvement over both RR and GP by reducing the load granularity.

Figure 5.15 shows the number of bytes received per core over six iterations from 25th iteration to 30th iteration from the same tracing runs shown in Figure 5.14. With GP, there is a peak exceeding 12 MB, which corresponds to the peak orange bar in Figure 5.14. This occurs because the particular processor has the location with the maximum load without decomposition. The average volume of communication per core is less than that of RR, which is around 4 MB. This is due to the fact that the graph partitioner does not balance communication evenly, but only minimizes its total volume.

Figure 5.16 shows the strong scaling of EPI SIMDEMICS with state data on Blue Waters. Again, we observe the patterns on Blue Waters similar as on Vulcan. The peak speedups achieved by the
Figure 5.12: Strong scaling performance of EPISIMDEMICs in speedup for selected state data on Vulcan (BG/Q).
Figure 5.13: Strong scaling performance of EpiSim DEMICS in execution time for selected state data on Vulcan (BG/Q).
Figure 5.14: Timeline plots of two iterations of EPISIMDEMICs using MI data and 4K core modules on Blue Waters (sampled to show 384 compute PEs). Each bar represents the activity of a processor over time progressing from left to right.

Figure 5.15: Amount of data received per PE in MB over six iterations using MI data and 4K core modules on Blue Waters while relying on manual buffering (see Section 5.1.4 for further details). GP has the highest maximum communication volume while GP-splitLoc has the lowest. Both have smaller average communication volume than RR and RR-splitLoc.
Figure 5.16: Strong scaling performance of EPISEMDEMICS in execution time for selected state data on Blue Waters (XE6)
actual runs without using decomposed data are on average 75% of that of the estimated speedup ($S_{ab}$), maximum 92%, and minimum 50% among the seven states’ data. On the other hand, the peak speedups achieved using decomposed data are quite low, on average 6.3%, maximum 9%, and minimum 4.5%.

Figure 5.17 shows the results of the largest scale experiments that were run. We used 352K core-modules (using 99.5% of the XE6 compute nodes) and achieved a speedup of 58,649 for an efficiency of 16.3% using the entire US population data. These numbers represent the best scaling for an individual-based epidemic diffusion simulation by a factor of 5.3. In addition, it shows the results from the other platforms used. For weak scaling, we choose a subset of the US data by state such that the number of people per processor remains approximately constant. With the largest dataset fixed, we demonstrate weak scalability by using less than 1/4 of the available memory, which allows us to run up to 2K cores on Blue Waters, 8K cores on Vulcan, and 4K cores on Cab. One observation we made is that the execution time using Vulcan (BG/Q) is slower than the other platforms, which is especially noticeable in weak scaling results. We plan to investigate this difference further in the future. Our current conjecture is that this is due to the fact that the A2 processor used in BG/Q wastes five cycles to redirect to the predicted target by flushing following fetches even for the taken-branches [146]. In addition, the L2 cache in an A2 processor is shared among cores [60]. Each thread in the current version of EPISIMDEMCIS has a private stream of data, potentially leading to cache thrashing, which is a similar problem described in [147]. The processor is also optimized for floating point performance [60].

5.3 Future Work

The workload in EPISIMDEMCIS contains both deterministic and non-deterministic portions. We focus on the former in this paper, and are currently investigating the latter. CHARM++ runtime offers measurement-based load balancing (LB) framework based on the principle of persistence. Since our application can have highly dynamic computation, this is not sufficient. Our plan is to address the dynamism by the application-specific workload prediction. The goal is to avoid incurring excessive overhead by initiating LB phases without a sufficient gain in performance as in [148], but by using application-specific information.

Finding the optimal state between balancing the time-varying non-uniform load and the spatial and temporal locality is an intractable task. We plan to investigate whether we can design a simple local interaction mechanism that leads to an emergent self-organization, such as swapping loads belonging to overlapping hypergraph edges that span to different processors [92,93,149,150], or artificially introducing a new vertex in a graph or merging vertices.
Figure 5.17: The weak scaling (left) and strong scaling (right) performance of EPIDEMICS. The optimizations discussed in Section 5.2.2 cause the difference between red (circle) and green (triangle) points for each number of cores used in subfigure (a), and between red (circle) and blue (diamond) points in subfigures (c) and (e). The difference between blue (diamond) and green (triangle) points is attributed to the domain decomposition discussed in Section 4.3.4. TRAM and mbuf represent two different message aggregation methods described in Section 5.1.4. For strong scaling, we use as much resources as allowed on each platform and the whole continental US population data. Subfigure (b) shows the largest strong scaling up to 352K cores.
To reduce the execution time on the Blue Gene/Q platform, we plan to apply a platform-specific optimization to improve L2 cache utilization as well as application-specific optimizations to minimize branching. Currently, in EPI SIMDEMICS, each compute thread processes a private data stream. We will take advantage of the CkLoop feature offered in CHARM++ to implement fine-grained parallelism [151, 152] such that the threads sharing L2 cache can work on a shared data stream.

5.4 Summary

In this chapter, we introduce the implementation of EPI SIMDEMICS in CHARM++, and discuss various optimizations applied to enhance the scaling performance. In addition to the optimizations discussed in Chapter 4, which take advantage of application semantics, we discuss communication optimizations including message aggregation, efficient symmetric multiprocessing and optimized synchronization as well as low-level code optimizations.

As a result of these optimizations, we have shown unprecedented scaling of an individual-based contagion diffusion model, scaling to over 352K cores on Cray XE6, a five-fold increase over the previous state-of-the-art achievement on Cray XT5 while still increasing parallel efficiency compared to the largest prior run at 64K cores. The improved turn around times will have a positive impact on the ability of policy makers to respond to emerging pandemics in the future. We also evaluate the impact of the locality optimizations and the static load balancing techniques discussed in Chapter 4 on the performance on various platforms including the Cray XE6 system, the IBM Blue Gene/Q system and the Infiniband-based clusters. We verify that the improvements in scaling are consistent across the platforms used.
Chapter 6

Conclusion

A plethora of data presents unprecedented opportunities as well as challenges for both data-intensive science and high performance computing. The volume of data growth stems from online activities on the Internet, ubiquitous sensors and various social networks. Data is the key element for a data-intensive science to enable new discoveries. For example, socio-technical sciences study the dynamics of complex large scale real-world networks by various methods including simulation on the network data or analysis of the structure [73, 75, 99, 153–155]. Activities of scientists from other domains include processing a large amount of signals from astronomical interferometry or atmospheric sensors in real-time or at high-throughput, or analyzing the massive simulation logs of the cellular level interactions in biological organizations [110, 156]. Making effective use of the massive volume of data generated at a rapid rate demands us to push the boundary of high performance computing, especially in terms of the scalability. Generic methods used to work reasonably well for a broad range of applications are not sufficient any more to support extreme scale.

Data-intensive scientific applications have a nontrivial portion of the execution time attributed to accessing and manipulating data. This dissertation is devoted to identifying the key techniques to optimize accessing data in memories for achieving high scalability. This includes enhancing temporal and spatial locality of data, as well as improving the bandwidth and hiding the latency for non-local accesses. For the experiment, we choose the platform environments that provide us with the full control of temporal or spatial locality in order to investigate how these aspects of data access impact the performance. We employ two cases that differ in the accesses pattern and the workload variation to investigate how application-specific information can be used to optimize data accesses. This study offers valuable lessons for designing future platforms by presenting optimization opportunities in the perspective of applications.
In the former case, we explore the enabling techniques for application-specific optimization with regular access pattern and uniform workload. These techniques not only allow the efficient management of data locality but also facilitate the optimal locality for an individual application. We present STRIDER, a runtime framework, to demonstrate such techniques on multi-core processors with software-managed memories [20, 157]. STRIDER optimizes the critical path of scheduling data transfers for multi-stride accesses in regular nested parallel loops, and distribute accesses between cores. STRIDER offers applications the precise control of data prefetching in terms of when and what to access as well as the control of what to keep for how long and what to write back at which moment. These allow applications to hide access latency while minimizing the overhead of generating accesses. In particular, STRIDER contributes new methods to improve temporal locality on software-managed memories. The prototype of STRIDER on the IBM Cell processor performs competitively to hand-optimized code and better than contemporary language frameworks, in both non-trivial parallel applications and important application kernels.

We also show that such techniques are accessible to application programmers requiring only a reasonable amount of programming effort. STRIDER transparently optimizes accesses for multi-dimensional arrays in nested parallel loops, given a high-level specification of loops and their data access patterns. When comparing STRIDER with the state-of-the-art framework SEQUOIA, based on the language and the compiler [21], we show that STRIDER requires only a marginal increase in the code sizes while greatly improving the application performance due to the abstraction for temporal locality, which is absent in SEQUOIA, and the superior runtime performance. STRIDER has served as an optimization module for a source-to-source compiler, which supports OpenMP style programming model [23, 49]. While it benefits the STRIDER’s runtime performance for strided accesses, the pure OpenMP style model requiring compiler translation does not provide as much fine-tuning opportunities as STRIDER allows and is not as flexible as STRIDER. An interesting future work to pursue is whether a small number of additions to the set of directives would unblock the pass to a greater set of optimization opportunities without making the programming needlessly complicated, or whether a completely new design of a set of directives would be appropriate.

Specifically, STRIDER’s transparent optimizations include the access pattern grouping, the multi-stride optimization by blocking and strip-mining, and the aggregate task blocking. In addition, STRIDER offers fine-tuning opportunities via a high-level API in the variable-depth buffering, the multi-level loop partitioning, the data blocking, and the flexible management of simultaneous accesses. STRIDER is a library-based framework, and versatile enough to support compiler optimization as well as stand-alone programming. The determination of fine-tuning parameters can be deferred until execution. It also promotes platform specific optimizations including the NUMA-aware page allocation and the data alignment.

Although the Cell B.E. architecture has retired from the latest HPC, the lessons learned from the ex-
experience are still valid, and provide useful guides for designing future platforms. Exposing locality controls are necessary to empower application-specific optimization and attain sustainable and scalable performance. However, for a new processor architecture and the optimization capabilities to be well received in the field, the programmability needs to be supported at an acceptable level, or at least be offered for multiple levels even with reduced capabilities. For example, in the latest Blue Gene/Q, prefetching is driven by cache misses rather than prediction [10, 146]. This may not be optimal in terms of performance, but it alleviates the programmer’s burden of scheduling accesses and managing temporal locality. This further eases the prefetching data for a regular access pattern, but with non-uniform or even dynamic workload per fetched data item. For a series of repetitive accesses, this can be as simple as marking a section of code using compiler directives for recording and replaying accesses. This study opens up new opportunities for runtime systems, architectural supports, and programming frameworks.

In the latter case, we uncover the challenges of scaling an application with irregular access pattern and non-uniform workload. Especially, we choose a contagion simulation application, EpiSimDEMICS, based on large-scale social networks. The input data set are synthesized based on publicly available information to match the statistical properties of the real-world population at the street block level [2]. In this case, the major challenges originate from the particular properties of the data as well the application behavior. The heavy-tailed [104–106] and irregular connectivity in a graph leads to an extreme variance in the workload distribution as well as the communication. In addition, graph-processing applications tend to generate a large number of small messages to communicate due to little spatial locality in graph data [158]. Both are features that are commonly found in applications that process large-scale graph data from real-world social networks. These features have been known to be performance bottlenecks.

Nonetheless, our work is the first to experimentally and analytically investigate the impact of the heavy-tailed property of data on the scalability on petascale platforms. Our study highlights the holistic view of the problem. While the majority of existing works focus on the locality aspect of the problem, it is crucial to consider the workload and the trade-off between load and locality. Our static spatial locality optimization takes load information into account. The load is application-specific as it depends on what operations are applied to the given work unit data. Furthermore, it depends on the platform that executes the operations. Such pieces of information have been ignored in the existing works studying general graph partitioning methodologies. However, these are the keys to achieving extreme performance scaling as they allow us to overcome the major bottlenecks we discuss in this dissertation.

Scalability amplifies the pragmatic value of EpiSimDEMICS. The strong scalability enables time-critical studies in response to a pandemic [64, 65]. The weak scalability fosters simulating a larger population. We present an implementation of EpiSimDEMICS in CHARM++ [67] that enables future
research by social, biological and computational scientists at unprecedented data and system scales. We present application-specific processing of graph data and demonstrate the effectiveness of these methods on a Cray XE6 and IBM Blue Gene/Q. Both are currently the top petascale HPC systems in the world. The fastest known simulation [87] reported a speedup of 10,000 on 64K cores (15.2\% efficiency) on Cray XT5. Our aforementioned approach, combined with the message aggregation, the algorithmic optimization, and the low-level code optimization, achieves a speedup of 14,357 (22\% efficiency) on the same number of cores on Cray XE6. Further, we demonstrate that our implementation of EpiSIMDEMICS can scale up to 360,448 cores and achieve a speedup of 58,649 (16.3\% efficiency), more than a five-fold increase in the number of cores with slightly improved efficiency. We also demonstrate the good weak scalability of EpiSIMDEMICS while occupying only a quarter of available memory space with data across the platforms we use.

EpiSIMDEMICS implements a graph dynamical system (GDS) [112, 124], and, thus, carries inherently dynamic computation. The load is not only non-uniform but also time-varying. The states of the system being studied, and the underlying network, and the update scheme (scenario intervention) co-evolve. The local function (vertex computation) changes accordingly. In this dissertation we have focused on the deterministic portion of the computation and suppressed the dynamic behaviors from scenario intervention. We plan to address the non-deterministic portion of the computation as we enable the full dynamism especially for scenario intervention studies. CHARM++ runtime offers measurement-based load balancing (LB) framework based on the principle of persistence assumption [159]. Our plan is to cope with the dynamism by an application-specific prediction of workload. This will help us to avoid incurring the excessive overhead by initiating LB phases without a sufficient gain in performance, as in [148], by using application semantics. Our future work includes finding the optimal state between balancing the time-varying non-uniform load and the spatial and temporal locality. Meanwhile, another direction to pursue is to focus on the primary load balancing objective while reducing the impact of the secondary locality objective. The secondary objective in this case includes both minimizing the impact of the total communication as well as the imbalance across processors. Especially, static analysis on the data flow may allow us to overlay persistent channels with constraints on bandwidth and latency [137, 160]. In addition, we plan to study how other structural properties of a graph in addition to the degree affect such an optimal state. Our next step is to go beyond the national-scale simulation towards the planetary-scale [161].

In this dissertation, we identify important aspects of data accesses and explore various optimization techniques to minimize the negative impact of data accesses on the scalability of data-intensive applications. This study provides valuable lessons for designing future architectures, runtime systems and programming models to support performance scaling for data-intensive scientific applications.
Bibliography


[10] I-Hsin Chung, Changhoan Kim, Hui-Fang Wen, and Guojing Cong. Application Data


Supercomputing, ICS ’09, pages 36–45, June 2009.


SC ’06, November 2006.


[54] Pieter Bellens, Josep M. Perez, Felipe Cabarcas, Alex Ramirez, Rosa M. Badia, and Jesus


[80] Bruce Hendrickson. Load balancing Fictions, Falsehoods and Fallacies. Applied Mathematical


[129] Abhinav Bhatel, Sameer Kumar, Chao Mei, James C. Phillips, Gengbin Zheng, and


[136] Parallel Programming Laboratory, Department of Computer Science, Univ. of Illinois, Urbana, IL. The Charm++ Programming Language Manual, (Version 6.4.0), 2012.


[164] Scott Clark, Kent Haselhorst, Kerry Imming, John Irish, Dave Krolak, and Tolga Ozguner.


Appendix A

Programming for Cell B.E. Processors

In this appendix, we provide various information, definitions and terms that we use in this dissertation. Section A.0.1 discusses the common challenges in programming for explicitly managed memories. Section A.0.2 and A.0.3 respectively presents the details of the processor architecture and the memory subsystem. Section A.0.4 discusses the specific challenges in programming for the particular processor we employ as the experimental platform. Section A.0.5 lists the information that a programmer needs to be aware of to write a code optimized in terms of performance.

A.0.1 Explicitly Managed Memory (EMM) Hierarchies

Multicore processors with explicitly managed memory (EMM) hierarchies originated in the domain of game graphics and are now emerging as general-purpose high-end computing platforms [14, 18]. More recently, processor vendors for mainstream computing markets such as Intel and AMD have introduced similar designs [162]. All of these processors have data-parallel components as accelerators. This acceleration is achieved through multiple scalar or single-instruction, multiple-data (SIMD) cores, high on-chip bandwidth, and explicit data transfers between fast local memories and external dynamic RAM (DRAM). Explicit data transfers enable programmers to use optimal caching policies and multiple streaming data buffers that allow overlapping computation with data transfer latency [7]. Managing the memory hierarchy in multicore processors requires trade-offs in terms of performance, code complexity, and optimization effort. Multicore processors based on coherent hardware-managed caches provide the abstraction of a single shared address space. This abstraction frees programmers from having to explicitly manage data as it moves through the memory hierarchy and between cores. The hardware automatically synchronizes data in main memory and across the caches in each core so that all cores have a consistent view of shared memory. In contrast, software-managed local memories introduce disjoint address spaces that
programmers are responsible for keeping consistent. Programmers are responsible for explicitly managing locality. Programmers must decide when to place data in local memories, which data to replace, and which data layout to use in local memories, possibly different from that on off-chip DRAMs [163].

A.0.2 Cell B.E. Architecture Overview

The Cell Broadband Engine is a heterogeneous multi-core processor [14]. One of the cores, the Power Processing Element (PPE), is a 64-bit two-way SMT PowerPC core with 32-KB L1-I cache, 32-KB L1-D cache, and 512-KB L2 cache. The other eight cores on the Cell B.E. are 128-bit SIMD-RISC processors, called Synergistic Processing Elements (SPEs), and typically used as accelerators for data-intensive computation. Each SPE has a 128-bit data path, 128 128-bit registers, and 256 KB of software-managed local store. An SPE can issue two instructions per cycle into two (odd/even) pipelines. One pipeline implements floating point instructions, whereas the other implements branches, load/stores and channel communication instructions, which we will describe shortly. All SPU instructions are inherently SIMD operations that the pipeline can run at four granularities: 16-way 8-bit integers, eight-way 16-bit integers, four-way 32-bit integers or single-precision floating-point numbers, or two 64-bit double-precision floating-point numbers.

The PPE accesses memory (the effective-address space) with load and store instructions as a conventional processor with hardware managed caches. The SPEs access memory with non-blocking Direct Memory Access (DMA) commands that move data between off-chip shared main memory and a private LS or between LS and LS. The SPU fetches instructions and accesses data only on its private LS, and there is no cache between SPU and LS. DMAs are posted with channel
commands from the SPE core to the Memory Flow Controller (MFC). Each MFC has a queue which can hold up to 16 outstanding DMA requests, each of which can send or receive up to 16 KB of contiguous data. The effective addresses of DMAs are translated into physical addresses using the PowerPC memory translation mechanisms.

The PPE, the SPEs, the Memory Interface Controller (MIC) and the Bus Interface Controller (BIC) are interconnected via the Element Interconnect Bus (EIB), as illustrated in Figure A.1. The EIB has two rings moving data clockwise, two rings moving data counterclockwise, and one address ring. The EIB operates at half the processor clock frequency and its maximum theoretical bandwidth is 204.8 GB/s. (This assumes all units connected to the EIB can snoop one address per EIB cycle and for each snooped address the EIB transfers a maximum of 128 bytes.) Actual point-to-point bandwidth ranges from 78 to 197 GB/s, depending on several factors, including the position of the units (SPEs, PPE, controllers) on the ring. The Bus Interface Controller (BIC) can be used to connect two Cell processors [164], as in the IBM QS20 Cell Blade system. A BIC controls seven transmit and five receive Rambus FlexIO links operating at 5 GB/s each, which are configured into two logical interfaces. Both of them can be configured to support I/O Interface (IOIF) non-coherent protocol. Otherwise, one of them can be set to support Broadband Engine Interface (BIF) coherent protocol to organize multiple Cell processors in a NUMA configuration. The peak bandwidth of BIF in a QS20 blade is 20 GB/s [165]. However, the actual bandwidth observed is lower and asymmetric [166].

A.0.3 Off-chip Memory and Local Storage Accesses

The on-chip Memory Interface Controller (MIC) interfaces Element Interconnect Bus (EIB) and the off-chip shared memory subsystem. It provides two channels using the dual XDR memory controller [164]. The PlayStation 3 has four 64-MByte XDR DRAM devices which together support total 256-MByte off-chip main memory, and each channel supports two devices as illustrated in Figure A.2. An IBM QS20 Cell blade has eight of such a device per node, and supports total 1GB off-chip main memory. The data bus between a XDR DRAM device and MIC operates at 3.2 Gbps. The maximum bandwidth per channel is 12.8 GB/s

Inside each XDR DRAM device, there are independent memory banks [39]. The MIC supports four, eight and sixteen memory banks on each of the two channels by sixteen logical banks rotating over the physical banks. The memory address maps to the physical memory, rotating over the banks and channels on a naturally aligned 128-byte basis [39, 164]. In natural alignment, the size of a data item is the same as the size of alignment unit and the address of an item is aligned, such that successive access on a naturally aligned array always points to the aligned addresses. Each channel supports one read queue and one write queue, holding up to 32 reads and 32 writes in each respective queue. A memory bank can handle only one operation at a time such as read, write,
Figure A.2: The memory interface controller (MIC) and the off-chip shared memory subsystem

and refresh. While a bank busy, the MIC places other accesses to the bank into the queues. The MIC starts accessing a free bank if there is a command waiting for the bank in a queue. Therefore, the peak utilization of the memory subsystem is achieved when the addresses of memory locations to access are evenly distributed over all the banks.

Accesses on each channel are 1 to 8, 16, 32, 64, or 128 byte basis. Writes of 16 bytes or more, but less than 128 bytes, can be written directly to memory using a masked-write operation but writes of less than 16 bytes require a read–modify–write operation. Due to the limited buffers for read–modify–write operations, the read part of the read–modify–write operation takes precedence over normal reads, while the corresponding write part of the operation takes precedence over normal writes [14]. When accessing data spread over non-contiguous region of memory—such as accessing a column of a matrix using a DMA list command—padding may help avoid the overhead of read-modify-write operations on a piece of data less than 16 bytes. In addition, keeping the number of memory operation commands in queue as little as possible increases the chance of further performance enhancement by the MIC fast-path mode, where an incoming request can bypass an empty request queue and be dispatched immediately [14]. The details of how a DMA command is issued, queued and processed, and how the bus resource is requested and allocated are described in [14, 167, 168].

An SPU accesses the local storage (LS) by the quadword (16 bytes), and loading and storing a sub-
quadword scalar costs extra overhead associated extracting from and inserting into a quadword. A scalar being loaded must be rotated into the preferred slot in the aligned quadword. A scalar store requires a read, an insert and a write operation [39, 168]. Such overhead can be avoided if a scalar array is padded such that each element is placed at the preferred slot. A properly padded array not only useful for the removal of rotating and masking overheads but also for vectorization.

### A.0.4 Cell B.E. Programming Challenges

We use the Cell Broadband Engine (Cell B.E.) processor as an experimental testbed to implement the auto-tuning framework for managing data locality in multiprocessors with EMM hierarchies. The native method of programming for Cell B.E. is to directly use IBM Software Development Toolkit (SDK) [61]. In the rest of the chapter, we refer this toolkit as Cell SDK. Cell SDK exposes the architectural details to programmers. It provides the low-level libraries for thread-based parallelization, and a set of DMA commands based on get/put interfaces to manage data transfers. Programming in Cell SDK is analogous to, if not harder than, programming with the message passing interface (MPI) or Pthreads on a typical cluster or multiprocessor. Programmers need a deep understanding of both thread-level parallelization and the Cell hardware. Cell SDK requires programmers to explicitly identify and schedule all data transfers. Further, programmers are solely responsible for synchronizing threads running on different cores, maintaining data coherency, aligning data, and setting up and sizing buffers to achieve computation/communication overlap.

While there exist high-level programming models transparently managing data transfers, hand-tuned parallelization has its advantages: Programmers, who have understanding of application algorithms and the Cell B.E. architecture, (i) can improve data locality, (ii) eliminate unnecessary data transfers, and (iii) optimally schedule data transfer and computation on cores.

In contrast to programming for homogeneous multicore processors and shared-memory symmetric multiprocessors (SMPs), programming for the Cell B.E. further presents unique challenges [39]. We summarize them as follows:

- **Local memory spaces:** To use of Cell processors effectively, it is recommended to parallelize and offload computation to SPEs as much as possible. The challenge is that the memory that an SPE has direct access to is not managed in the traditional method of managing memory hierarchies. In other words, there is no hardware mechanism automatically staging data. Programmers must transfer data between the local store and the shared main memory explicitly through direct memory accesses (DMAs).

- **Small local storage:** Each SPE has a local storage of 256 Kbytes, which contains both the code and the data used by the SPE. As a result, the larger the code size, the less the space left for
data. Thus, streaming data is crucial to make efficient use of the limited storage as well as to achieve sustained performance by hiding latency.

- **Latency hiding**: If programmers can anticipate which data will be used in the future computations, they can initiate the DMAs to load such data in advance while computing on current data.

- **Data alignment**: The DMA engine allows naturally aligned transfers of 1, 2, 4, or 8 bytes, or a multiple of 16 bytes to maximum of 16K bytes aligned to a 16-byte boundary.

- **Strided access**: A single DMA command only transfers a block of contiguous data. To transfer non-contiguous data, such as the columns in a matrix, to and from main memory, programmers must construct DMA lists. Each entry in a DMA list generates a separate DMA. It is programmers responsibility to ensure that the memory address for each subsequent entry in the list adheres to the stride they want to access with as well as the alignment constraints.

- **Small numbers of DMA identifiers**: There are only 32 tags available to keep track of the completion of non-blocking DMA transfers. This limits the number of uniquely identifiable schedules of simultaneous transfers.

The DMA list mechanism allows handling a bundle of transfers using a single interface call and a single tag. A DMA list command can process a list of up to 2048 DMA transfers in a batch. A DMA list contains the list of effective addresses to access in the memory, specifically, only the lower half of each total 64-bit effective address (EAL). Issuing a DMA list command requires the five parameters as follows:

1. the size of the list
2. the address of the list on LS
3. the starting local store address (LSA) for transferring data to and from
4. the common effective address high (EAH) for all the addresses in the list
5. the DMA tag for the bundle of transfers

The mechanism has further constraints as follows:

- A DMA list command can only be issued by the program running on the SPE of the corresponding local store for the transfers.

- As the effective address high (EAH) is specified only once in a DMA list command, every transfer in a list is expected to access the memory location with the same EAH.

- A list stored in the LS must align on an 8-byte boundary.
When transferring using a DMA list command, the LSA is internally incremented based on the amount of data transferred by each transfer element. However, if the starting LSA for each transfer element in a list does not begin on a 16-byte boundary, then hardware automatically increments the LSA to the next 16-byte boundary. Programmers are responsible to write codes accessing LSA accordingly.

### A.0.5 Challenges to Achieve Sustained DMA Performance

In addition to the constraints to ensure the correctness of execution, there exist further conditions to achieve the sustained performance for accessing off-chip main memory, as partially explained in Section A.0.3 and other documents [39, 61]. We summarize the recommended practices as follows:

- **Use 128–byte aligned data if possible:** The Cell processor supports up to 16 memory banks, and the memory interface controller (MIC) can access different banks simultaneously. The physical address space spans across 16 memory banks with a 128-byte natural alignment in round-robin fashion. To achieve the peak performance, it is important to simultaneously utilize all the banks using the addresses aligned to 128-byte basis and the transfer size multiple of physical banks times 128 bytes.

- **Mix reads and writes:** Each of the two channels in MIC can hold up to total 32 reads in the read queues and 32 writes in the write queues. The MIC arbiter alternates dispatching between the read queue and the write queue after every minimum of eight dispatches from each queue or until the queue becomes empty. Thus, it is better to keep both queues filled.

- **Use data padding to 16 bytes:** Write operations of multiples of 16 bytes but not larger than 128 bytes can directly write to memory using a masked–write operation. Writes less than 16 bytes require a read–modify–write operation. As the number of buffers for such operations are limited, reads/writes with multiples of 16 bytes are given higher priority than the others to prevent the limited resource from being occupied for longer duration.

- **Use DMA lists:** Each SPE can issue up to sixteen outstanding DMAs. Sixteen batched DMA commands achieve the performance close to that of pure continuous non-blocking DMA.

- **Avoid overusing DMA lists:** A DMA list allows to run computation and DMA commands and simultaneously. However, there is overhead associated with dereferencing each element in a list and inserting it into the DMA queue. Thus, a DMA list should be used when there is sufficient computation to overlap with, and be short enough not to add excessive much overhead.

- **Merge DMAs or reduce the number of DMA calls:** The overhead of coherence protocol
initiating each DMA is non-trivial.

- Utilize EIB bandwidth for communication between SPEs: The EIB bus offers a far greater total peak bandwidth (204.8 GB/s) than that of the off-chip memory (25.6 GB/s). Applications should make use of the EIB bandwidth as much as possible for communication between SPEs. However, such communication must be carefully arranged as each of four EIB rings allows only up to three concurrent data transfers of non-overlapping paths.
Appendix B

Application Implementations on Cell B.E.

In each of three PBPI implementations, there are three major loops offloaded to SPEs. Due to the space limitation, we only show the loop that is called the most often in Figure B.2. STRIDER is a library-based framework, and as such does not require any special compiler. Figure B.3 shows the code written in STRIDER to offload the loop shown in Figure B.2. The counterpart is not shown here but plugged into the runtime framework in line 39 of the code shown in Figure B.3. On the other hand, SEQUOIA is a language-based framework, Thus, programmers write the loops to offload in the SEQUOIA language as shown in Figure B.4 and B.5. Then, the SEQUOIA compiler translates the code to another code such that the native compiler can create an executable using the translated code. Programmers may choose to provide a custom kernel, instead of relying on the one generated by the compiler base on the analysis of the computation described in a task as in Figure B.4. This is particularly useful when there is potential performance improvement using SIMD operations. SEQUOIA compiler does not handle such an architecture specific syntax. Instead, it provides a task mapping method to link the code generated to transfer data with the code given to efficiently compute on the data. The primary use of a task mapping file is to support architecture independent abstraction by the language while allowing it to efficiently map to the underlying architecture. It is programmer’s responsibility to write such a mapping as shown in Figure B.7. Finally, the main application code calls the offloaded task as shown in Figure B.6. Figure B.8 and B.9 respectively show the SPE offloader and PPE offloader of Fixedgrid implementation in STRIDER. Figure B.10, B.11, B.12, and B.13 show the code added for implementing Fixedgrid in SEQUOIA. Figure B.14, B.15, and B.16 show the code added for implementing Jacobi in STRIDER. Figure B.17, B.18, B.19, and B.20 show the code added for implementing Jacobi in SEQUOIA.
double* A = buf0;
double* B = buf1;
for (ri=0; ri<repeat; ri++) {
    double residual = 0.0;
    for (i=1; i<=M; i++) {
        for (j=1; j<=N; j++) {
            double resid = A[i][j] - B[i][j];
            residual += resid*resid;
        }
    }
    double* tmp = A;
    double* A = B;
    double* B = tmp;
}

Figure B.1: The serial implementation of Jacobi.

void loop1(int N, double *pl, double *pr, double *tl, double *tr, double *pn) {
    int i;
    double t1, t2;
    for(i=0; i<N; i++) {
        t1 = pl[0]*tl[0] + pl[1]*tl[1] + pl[2]*tl[2] + pl[3]*tl[3];
        t2 = pr[0]*tr[0] + pr[1]*tr[1] + pr[2]*tr[2] + pr[3]*tr[3];
        pn[0] = t1 * t2;
        pn[1] = t1 * t2;
        pn[2] = t1 * t2;
        pn[3] = t1 * t2;
        pn = pn + 4;
        pl = pl + 4;
        pr = pr + 4;
    }
}

Figure B.2: One of three loops of PBPI in the original serial implementation.
int loop1(unsigned int pn_a, unsigned int pl_a, unsigned int tl_a, \
    unsigned int pr_a, unsigned int tr_a, int nPattern)
{
    unsigned int elesz = sizeof(double) * 4;
    unsigned int i_start, i_stop;
    BlkDesc blk;
    IterSpace is;
    Iterator iti, ito;
    MonoChromatic mci, mco;
    Waves waves;
    DMA_get_prof(1, (void*) tl, tl_a, 16 * sizeof(double), 0, NULL, 0);
    DMA_get_prof(1, (void*) tr, tr_a, 16 * sizeof(double), 1, NULL, 0);
    DMA_wait_prof("loop1 tl", 0, 1, 0);
    DMA_wait_prof("loop1 tr", 1, 1, 0);
    set_blockdesc(blk, _BLK_TYPE1, 1, elesz, NULL, 0);
    alloc_iterspace_1D(&is);
    set_iterbasis_1D(&is, 0, nPattern, nPattern, 1, _FORWARD);
    init_iterspace(&is, elesz);
    i_start = 0;
    i_stop = get_num_iterations(&is);
    compute_bounds(&i_start, &i_stop, 1, IDspe, Nspes);
    alloc_iterator(&iti, &is);
    alloc_iterator(&ito, &is);
    set_iterator(&iti, &ito, &waves, 1, 0);
    set_iterator(&iti, i_start, i_stop);
    set_iterator_as(&ito, &iti);
    alloc_waves(&waves, 3);
    set_monochromatic(&mci, &iti, &waves, 2, 0);
    set_wave(&mci, 0, pl_a, 0, &blk, 1);
    set_wave(&mci, 1, pr_a, 0, &blk, 1);
    set_monochromatic(&mco, &ito, &waves, 1, 2);
    set_wave(&mco, 0, pn_a, 0, &blk, 1);
    set_kernel(1, kernel_1);
    strider_nlmD_Stride_f_pn(1, &mci, &mco, NULL, L1NIter);
    free_waves(&waves);
    free_iterator(&iti);
    free_iterator(&ito);
    free_iterbases(&is);
    return 1;
}
void task<leaf> Loop1::Leaf(in double pl[N], in double pr[N], \
  in double tl[M], in double tr[M], out double pn[N])
{
  unsigned int i, j, NQ;
  double t1, t2;

  NQ = N/4;
  j=0;
  for(i=0; i<NQ; i++)
  {
    j += 4;

    t1 = pl[j]*tl[0] + pl[j+1]*tl[1] + pl[j+2]*tl[2] + pl[j+3]*tl[3];
    t2 = pr[j]*tr[0] + pr[j+1]*tr[1] + pr[j+2]*tr[2] + pr[j+3]*tr[3];
    pn[j] = t1 * t2;

    pn[j+1] = t1 * t2;

    pn[j+2] = t1 * t2;

    pn[j+3] = t1 * t2;
  }
}

void task<ext> Loop1::Leaf_ext(in double pl[N], in double pr[N], \
  in double tl[M], in double tr[M], out double pn[N]);

Figure B.4: Two leaf task definitions of PBPI in SEQUOIA. We do not use the definition, Leaf in our study. Instead, we use the external leaf task, Leaf_ext, the body of which is not shown here. It utilizes SIMD operations which are not handled by the SEQUOIA compiler. Sequoia allows to use such a task via the task mapping as shown in Figure B.7.
void task<inner> Loop::Inner(in int lt[LL], in double pl[N], \
    in double pr[N], in double tl[M], in double tr[M], out double pn[N])
{
    int looptype = lt[0];
    tunable T0;
    if (looptype == 1) {
        mapseq (unsigned int i = 0 : (N+T0-1)/T0) {
            Loop1(pl[i*T0;T0], pr[i*T0;T0], tl[0;16], tr[0;16], pn[i*T0;T0]);
        }
    } else if (looptype == 2) {
        // omitted for space
    }
    else if (looptype == 3) {
        // omitted for space
    }
}

void task<inner> Loop1::Inner(in double pl[L], in double pr[L], \
    in double tl[M], in double tr[M], out double pn[L])
{
    tunable T;
    mappar (unsigned int i = 0 : (L+T-1)/T) {
        Loop1(pl[i*T;T], pr[i*T;T], tl[0;16], \
            tr[0;16], pn[i*T;T]);
    }
}

Figure B.5: Inner task definition of PBPI implemented in SEQUOIA.

sqSize_t numels1[1] = {4*g_ds.nPattern};
sqSize_t numels3[1] = {16};
sqSize_t numels4[1] = {4};

sqArray_t* sq_pl = sqAllocArray(sizeof(double), 1, numels1, "sq_pl");
sqArray_t* sq_pr = sqAllocArray(sizeof(double), 1, numels1, "sq_pr");
sqArray_t* sq_pn = sqAllocArray(sizeof(double), 1, numels1, "sq_pn");
sqArray_t* sq_tl = sqAllocArray(sizeof(double), 1, numels3, "sq_tl");
sqArray_t* sq_tr = sqAllocArray(sizeof(double), 1, numels3, "sq_tr");
sqArray_t* sq_lt = sqAllocArray(sizeof(int), 1, numels4, "sq_lt");

....

loop_entrypoint(sq_lt, sq_pl, sq_pr, sq_tl, sq_tr, sq_pn);

Figure B.6: Task offloading via the entrypoint call of PBPI implemented in SEQUOIA.
Figure B.7: Mapping specification of PBPI implemented in SEQUOIA.
int discretize_col(void) {
    static const unsigned int elesz = sizeof(double);
    unsigned int i_start, i_stop;

    BlkDesc blk;
    IterSpace is;
    Iterator iti, ito;
    MonoChromatic mci, mco;
    Waves waves;

    set_blockdesc(blk, _BLK_TYPE2, NROWS, elesz*VECTOR_LENGTH, NULL, ROW_SIZE);
    alloc_iterspace(&is, 2);
    set_iterbasis(&is, 1, 0, NCOLS, ROW_LENGTH, VECTOR_LENGTH, 2, _FORWARD);
    set_iterbasis(&is, 2, 0, NROWS, NROWS, NROWS, 1, _FORWARD);
    init_iterspace(&is, elesz);

    i_start = 0;
    i_stop = get_num_iterations(&is);
    compute_bounds(&i_start, &i_stop, 1, IDSpe, Nspes);

    alloc_iterator(&iti, &is);
    alloc_iterator(&ito, &is);
    set_iterator(&iti, i_start, i_stop);
    set_iterator_as(&ito, &iti);

    alloc_waves(&waves, 4);
    set_monochromatic(&mci, &iti, &waves, 3, 0);
    set_wave(&mci, 0, conc_a, 0, &blk, 1);
    set_wave(&mci, 1, wind_v_a, 0, &blk, 1);
    set_wave(&mci, 2, diff_a, 0, &blk, 1);
    set_monochromatic(&mco, &ito, &waves, 1, 3);
    set_wave(&mco, 0, conc_a, 0, &blk, 1);

    push_double_param(pass.dt);
    set_kernel(COL_DISCRET_OFFLOAD, kernel_col);
    strider_nlmD_Stride_f_pn(2, &mci, &mco, NULL, 1);

    init_param_stack();
    free_waves(&waves);
    free_iterator(&iti);
    free_iterator(&ito);
    free_iterbases(&is);

    return 1;
}

Figure B.8: The SPE offloader for the column discretization task of Fixedgrid implemented in STRIDER.
void offload_discretize_row(real_t dt) {
    uint32_t i, j;
    MMGP_offload();
    for (j=0; j < Nspes; j++) {
        i = get_sigord(j);
        ((struct pass_t *)pass[i])->dt = dt;
        MMGP_start_SPE(i, ROW_DISCRET_OFFLOAD);
    }
    MMGP_wait_SPE(ROW_DISCRET_OFFLOAD);
}

void offload_discretize_col(real_t dt) {
    uint32_t i, j;
    MMGP_offload();
    for (j=0; j < Nspes; j++) {
        i = get_sigord(j);
        ((struct pass_t *)pass[i])->dt = dt;
        MMGP_start_SPE(i, COL_DISCRET_OFFLOAD);
    }
    MMGP_wait_SPE(COL_DISCRET_OFFLOAD);
}

int main(void) {
    ....
    cellstrider_start();
    init();
    for (iter=1, G->time = G->tstart; G->time < G->tend; G->time += G->dt, ++iter)
    {
        start_saprc99(G);
        for (k=0; k<NLOOKAT; k++)
        {
            offload_discretize_row(G->dt/2.0);
            offload_discretize_col(G->dt);
            offload_discretize_row(G->dt/2.0);
        }
        update_model(G);
    }
    cellstrider_finish();
    return 0;
}

Figure B.9: The PPE offloader of the \texttt{Fixedgrid} implementaion in STRIDER.
Figure B.10: The external leaf task of Fixedgrid in SEQUOIA. The bodies utilizing SIMD operations are not shown here.

Figure B.11: Task offloading via the entrypoint call of Fixedgrid implemented in SEQUOIA.
```c
void task<inner> discretize_all::Inner(in int dims[N4], \
in double DXYT[DD], in double wind[SZ1][SZ2], \
in double diff[SZ1][SZ2], inout double conc[SZ0][SZ1][SZ2])
{
  int row_or_col = dims[3];
  if (row_or_col == IS_ROW) {
    mapseq (int i=0 : NSPEC) {
      discretize_all_rows(dims[0;4], DXYT[0;2], wind[0; NROWS][0; NCOLS], \
      diff[0; NROWS][0; NCOLS], conc[i; 1][0; NROWS][0; NCOLS], \
      conc[i; 1][0; NROWS][0; NCOLS]);
    }
  }
  else {
    mapseq (int i=0 : NSPEC) {
      discretize_all_cols(dims[0;4], DXYT[0;2], wind[0; NCOLS][0; NROWS], \
      diff[0; NCOLS][0; NROWS], conc[i; 1][0; NROWS][0; NCOLS], \
      conc[i; 1][0; NROWS][0; NCOLS]);
    }
  }
}

void task<inner> discretize_all_cols::Inner(in int dims[N4], \
in double DXYT[DD], in double wind[SZ1][SZ2], in double diff[SZ1][SZ2], \
in double concIn[SZ0][SZ1][SZ2], out double concOut[SZ0][SZ1][SZ2])
{
  mappar (int k=0 : NCOLSV) {
    discretize_all_cols(dims[0;4], DXYT[0;2], \
    wind[0; NROWS][k*VECTOR_LENGTH; VECTOR_LENGTH], \
    diff[0; NROWS][k*VECTOR_LENGTH; VECTOR_LENGTH], \
    concIn[0; 1][0; NROWS][k*VECTOR_LENGTH; VECTOR_LENGTH], \
    concOut[0; 1][0; NROWS][k*VECTOR_LENGTH; VECTOR_LENGTH]);
  }
}

void task<inner> discretize_all_rows::Inner(in int dims[N4], \
in double DXYT[DD], in double wind[SZ1][SZ2], in double diff[SZ1][SZ2], \
in double concIn[SZ0][SZ1][SZ2], out double concOut[SZ0][SZ1][SZ2])
{
  mappar (int j=0 : NROWSV) {
    discretize_all_rows(dims[0;4], DXYT[0;2], \
    wind[j*VECTOR_LENGTH; VECTOR_LENGTH][0; NCOLS], \
    diff[j*VECTOR_LENGTH; VECTOR_LENGTH][0; NCOLS], \
    concIn[0; 1][j*VECTOR_LENGTH; VECTOR_LENGTH][0; NCOLS], \
    concOut[0; 1][j*VECTOR_LENGTH; VECTOR_LENGTH][0; NCOLS]);
  }
}
```

Figure B.12: Inner task definitions of Fixedgrid implemented in SEQUOIA.
Figure B.13: Mapping specification for the column discretization task of **Fixedgrid** implemented in **SEQUOIA**.
void jacobi_2D(int fid)
{
    uint32_t rowSz = pass.rowSz;
    uint32_t colSz = pass.colSz;
    uint32_t writeSz = pass.writeSz;
    uint32_t readSz = writeSz + 2*BPAD;
    uint32_t IRefDepth = 2;
    double residual = 0.0;

    MonoChromatic mci, mco;
    IterSpace isi, iso;
    Iterator iti, ito;
    BlkDesc blki, blko;

    set_blockdesc(blki, _BLK_TYPE1, 1, readSz*sizeof(double), NULL, 0);
    set_blockdesc(blko, _BLK_TYPE1, 1, writeSz*sizeof(double), NULL, 0);

    uint32_t D1Eo = rowSz + BPAD;
    uint32_t D2Ei = colSz + IRefDepth;

    alloc_iterbases(&isi, 2);
    set_iterbasis(&isi, 2, 0, D2Ei, Nrow, 1, 1, FWRD);
    set_iterbasis(&isi, 1, 0, rowSz, writeSz, writeSz, 2, FWRD);
    init_iterspace(&isi, sizeof(double));

    alloc_iterbases(&iso, 2);
    set_iterbasis(&iso, 2, 1, D2Ei-1, Nrow, 1, 1, FWRD);
    set_iterbasis(&iso, 1, BPAD, D1Eo, writeSz, writeSz, 2, FWRD);
    init_iterspace(&iso, sizeof(double));

    alloc_iterator(&iti, &isi);
    alloc_iterator(&ito, &iso);

    uint32_t i_start = 0;
    uint32_t i_stop = get_num_iterations(&isi);
    compute_bounds(&i_start, &i_stop, 1, IDspe, Nspes);
    set_iterator_w_refdepth(&iti, &i_start, &i_stop, IRefDepth);
    int32_t diff_beg, diff_end;
    get_iter_diff(&iti, &diff_beg, &diff_end, 0, 0);
    i_start -= diff_beg;
    i_stop -= diff_end;
    set_iterator_w_refdepth(&ito, &i_start, &i_stop, 0);

    Waves waves;
    alloc_waves(&waves, 2);
    set_monochromatic(&mci, &iti, &waves, 1, 0);
    set_i_wave(&mci, 0, pass.A, 0, &blki, 1);
    set_monochromatic(&mco, &ito, &waves, 1, 1);
    set_o_wave(&mco, 0, pass.B, 0, &blko, 1);

    set_uint_user_param_at(0, 1);
    set_uint_user_param_at(1, BPAD);

Figure B.14: The SPE offloader of the Jacobi implementation in STRIDER: Part 1. The rest is shown in Figure B.15.
set_uint_user_param_at(2, writeSz);
set_double_user_param_at(3, residual);
strider_nLmD_Stride_f_pn_vbd(fid, &mci, &mco, NULL);
pass.residual = (double) get_double_user_param_at(3);
free_waves(&waves);
free_iterator(&iti);
free_iterator(&ito);
free_iterbases(&isi);
free_iterbases(&iso);
}

Figure B.15: The SPE offloader of the Jacobi implementation in STRIDER: Part 2. The rest is shown in Figure B.14.

strider_start();
MMGP_offload();
for (i=0; i< Nspes; i++) {
  ((struct param_t *)param[i])->rank = myid;
  ((struct param_t *)param[i])->decompo_opt = decompo_opt;
  MMGP_start_SPE(i, INITPARAM);
}
MMGP_wait_SPE(INITPARAM);
for (ri=0; ri<repeat; ri++)
{
  double residual = 0.0;
  MMGP_offload();
  for (i = 0; i < Nspes; i++)
  {
    speids[i] = MMGP_get_SPE();
    ((struct pass_t *)pass[speids[i]])->A = (unsigned int) (buf0-BPAD);
    ((struct pass_t *)pass[speids[i]])->B = (unsigned int) (buf1-BPAD);
    ((struct pass_t *)pass[speids[i]])->Nc = chunkSize;
    ((struct pass_t *)pass[speids[i]])->rowSz = gridSize;
    ((struct pass_t *)pass[speids[i]])->Ncol = stride;
    ((struct pass_t *)pass[speids[i]])->colSz = rows;
    MMGP_start_SPE(speids[i], id_jacobi_2D1);
  }
  for (i = 0; i < Nspes; i++)
  {
    MMGP_wait_one_SPE(id_jacobi_2D, speids[i]);
    residual += (double) ((struct pass_t *)pass[speids[i]])->residual;
    MMGP_put_SPE(speids[i]);
  }
}
strider_finish();

Figure B.16: The PPE offloader of the Jacobi implementation in STRIDER.
void task<leaf> jacobi::leaf(in int lt[LL], in double A[MB][NB], \
    out double B[M][N], inout double Resd1[R1])
{
    int i,j;
    double resid, residual = 0.0;
    for (j=0; j<N; j++) {
        for (i=0; i<M; i++) {
                        + A[i+1][j+BPAD-1] + A[i+1][j+BPAD+1])/4.0;
            resid = A[i+1][j+BPAD] - B[i][j];
            residual += resid * resid;
        }
    }
    Resd1[0] = residual;
    if (MB-2 != M) {
        for (j=0; j<N; j++)
            B[MB-2][j] = A[MB-1][j+BPAD];
    }
}

void task<ext> jacobi::leaf_ext(in int lt[LL], in double A[MB][NB], \
    out double B[M][N], inout double Resd1[R1]);

Figure B.17: The leaf task of the Jacobi running on SPEs implemented in SEQUOIA.
```c
void task jacobi(in int lt[LL], in double A[M][N], \n    out double B[M][N], inout double Resd1[R1]);
void task jacobi1(in int lt[LL], in double A[MB][NB], \n    out double B[M][N], inout double Resd1[R1]);
void task<inner> jacobi::inner(in int lt[LL], in double A[M][N], \n    out double B[M][N], inout double Resd1[R1], out double Resd2[R2])
{
    tunable Mblk;
    tunable Nblk;
    tunable MAXL;

    int looptype = lt[0];
    int _N = lt[2];
    int Nblk_rem = lt[3];

    if (looptype == 1) {
        mappar (int j=0 : _N/Nblk, int i=0 : (M-2+Mblk-1)/Mblk) {
            jacobi1(lt[0;4], A[i*Mblk; Mblk+2][j*Nblk; Nblk+BPAD*2], \n                B[i*Mblk+1; Mblk][j*Nblk+BPAD; Nblk], Resd1[(j%12)*24+(i%12)*2; 2]);
        }
    } else {
        // To handle the remaining part not covered by the major blocking.
        // Not shown here in the interest of space.
    }
}

void task<inner> jacobi1::inner(in int lt[LL], in double A[M][N], \n    out double B[M][N], inout double Resd1[R1])
{
    tunable Mblk;
    tunable Nblk;
    int _N = lt[2];

    mappar (int j=0 : _N/Nblk, int i=0 : (M-2+Mblk-1)/Mblk) {
        jacobi1(lt[0;4], A[i*Mblk; Mblk+2][j*Nblk; Nblk+BPAD+2], \n            B[i*Mblk+1; Mblk][j*Nblk+BPAD; Nblk+1], Resd1[(j%12)*24+(i%12)*2; 2]);
    }
}

Figure B.18: The inner task of the Jacobi running on SPEs implemented in SEQUOIA.
```
sqSize_t A_numElmts[2] = { col_len, row_len };  
sqSize_t B_numElmts[2] = { col_len, row_len };  
sqSize_t numl[1] = {4};  
unsigned int redbuflen = 288;  

sqSize_t numR[1] = {288};  

sqArray_t* sq_l = sqAllocArray(sizeof(int), 1, numl, "sq_l");  
sqArray_t* R_sq = sqAllocArray(sizeof(double), 1, numR, "R_sq");  
sqArray_t* A_sq = sqAllocArray(sizeof(double), 2, A_numElmts, "A");  
sqArray_t* B_sq = sqAllocArray(sizeof(double), 2, B_numElmts, "B");  

for (ri=0; ri<repeat; ri++) {  
    double residual = 0.0;  
    memset(R_sq->ptr, 0, redbuflen*sizeof(double));  
    jacobi_inner_entrypoint(sq_l, A_sq, B_sq, R_sq);  
    for (i=0; i<redbuflen; i+=2)  
        residual += ((double*)R_sq->ptr)[i];  
    sqArray_t* swp_sq = A_sq;  
    A_sq = B_sq;  
    B_sq = swp_sq;  
}  

Figure B.19: The entrypoint of the Jacobi running on PPE implemented in SEQUOIA.
Figure B.20: The task mapping file written for the Jacobi implemented in SEQUOIA.