Ct:
A Flexible Parallel Programming Model for Tera-scale Architectures

Primary Authors:

Anwar Ghuloum, Eric Sprangle, Jesse Fang, Gansha Wu, Xin Zhou

Key Contributors:

Biao Chen, Yongjian Chen, Zhao Hui Du, Zhenying Liu, Mohan Rajagopalan, Byoungro So, Zhi Gang Wang

Table of Contents

1.0 New Opportunities, New Challenges

Processor architecture is evolving towards more software-exposed parallelism through two features: multiple cores and wider SIMD ISA. At the same time, graphics processing units (GPUs) are gradually adding more general purpose programming features.

Two key software development challenges arise from these trends. First, how do we mitigate the increased software development complexity that comes with exposing parallelism to developers? Secondly, how do we provide portability across (increasing) core counts and SIMD ISA?

Researchers at Intel's Microprocessor Technology Lab have developed a new programming model called Ct to address both of these challenges. Ct is a *deterministic* parallel programming model intended to leverage the best features of emerging general-purpose GPU (GPGPU) programming models while fully exploiting CPU flexibility. A key distinction of Ct is that it is a comprehensive data parallel programming model that gives programmers the flexibility to write code for multiple processor architectures. By contrast, most GPGPU programming models take are designed around the underlying constraints of the architecture for which code will be written.

1.1 Process and Architecture Trends

CPU designs are increasingly power-constrained. With every new silicon process generation, linear dimensions shrink by 30%, which has the following implications:

- The area of die needed to hold a constant number of transistors is reduced by half ($0.7^2 = 0.5$); alternatively, twice the number of transistors will fit within the same die area.

- The capacitance of each transistor shrinks by 30%;

- The maximum voltage decreases by approximately 10%; and

- The switching time of a transistor shrinks by 30% (at maximum voltage)

Based on the above, we can compute the implications for power scaling. Specifically:

Power scales in proportion to the number of transistors * capacitance per transistor * voltage^2 * frequency, which = $2 * 0.7 * 0.9^2 * 1/.7 = 1.62x$ per generation

While voltage may drop by slightly more than 10% per generation, or capacitance may drop by slightly more than 30%, this does not substantially affect power scaling trends. Another way of looking at this is that silicon scaling improves transistor density by 50% per generation, but only reduces power 20% per generation.

As a result, power efficiency is the ultimate goal because power consumption is the ultimate limiter to improving computational performance in silicon technology:.The first order design concern for tera-scale architectures is to improve computational power efficiency (MIPS/watt) over traditional processors on parallel workloads.

While increasing core count will continue to scale performance, the hardware and power requirements related to exposing parallelism in traditional out-of-order processors must be addressed. Two of the main techniques for writing explicit parallel code are long vector ISA and simultaneous multi-threading. Each of these techniques has strengths and weaknesses, depending on the type of code the techniques are intended to run.

Simultaneous multi-threading is a powerful technique for hiding memory latency in applications with poor locality, eliminating the need for more expensive out-of-order techniques. But it requires more area, so it should be used only to the point where it effectively hides memory latency. Future architectures will rely on longer SIMD vectors (e.g. 4, 8, 16 elements per register) to improve power efficiency, but generally

there is a "harder" limit on the effectiveness of increasing vector length: Increasing vector length creates inefficiencies for algorithms that inherently use shorter or unaligned vectors. Given the types of code that we believe will be important in the future (graphics rendering, media, etc, which utilize streaming memory), architectures will have to strike the right balance between these techniques. For example, we expect it is appropriate to scale the number of cores at about 2x per process generation (tracking Moore's Law), but the SIMD width will scale much more slowly.

These issues present the software developers with several architectural variables to which they must adapt.

1.2   Programming Tera-scale Architectures

Independent software vendors (ISVs) are excited by the peak throughput benefits of future Intel architectures, but at the same time they are concerned about the burden on the application developer to write explicitly parallel code. Using threads and vector intrinsics gives maximum flexibility to the programmer, but at great expense of programmer productivity, application portability and scalability. Moreover, evolving vector lengths and core counts are going to create performance scaling problems for software developers. Backwards compatibility in CPUs guarantees functional correctness but does not address performance scaling, which may regress as architecture evolves!

GPU hardware and software vendors have taken steps toward solving this problem. Languages for GPUs relieve the programmer of having to think about threads or SIMD width by supporting, for example, the DirectX programming model. In this model, a single data element (pixel) of a collection (display surface) is processed independent of neighboring data elements (pixels). This simpler model of computation reflects the underlying GPU architecture and is often called *streaming data parallelism*.

GPU programming models are constrained in such a way that the compiler and runtime can reason about the application and extract the parallelism automatically. Examples of this include DirectX, CUDA, and Cg. If the programmer can reformulate the application to work under GPU constraints, the

compiler/runtime can do the rest automatically. However, reformulating the application to fit these constraints often requires considerable programmer effort, and can result in significantly less efficient software algorithms. For example, it is difficult to operate efficiently on linked lists or compressed data structures, so applications that would naturally like to use these types of algorithms must be reformulated to use algorithms more consistent with GPGPU models.

Intel architecture is more general purpose than GPU and other coprocessor architecture. Unlike GPUs, Intel architectures have

1. Inter-core communication through substantial, coherent cache hierarchies

2. Efficient, low latency thread synchronizations across the entire processor array

3. Narrower effective SIMD width

More general purpose hardware allows Intel architecture to run more general purpose software algorithms (for example, algorithms that employ linked lists). So while Intel architecture can run applications written to use a GPU programming model, these applications are more constrained than necessary. That is, it makes sense to define a constrained programming model so the compiler and runtime can extract the parallelism. However, a model that is less constrained than most GPGPU models is highly desirable, so that applications do not have to be reformulated as substantially. At a high level, this is the goal of Ct: to define a constrained programming model that efficiently and portably targets highly parallel general purpose cores, such as Intel multi-core and tera-scale systems. To ease incremental adoption, Ct seamlessly extends C/C++ and can be used with legacy threading APIs.

1.3   Ct: Nested Data Parallelism

It often is convenient for the programmer to think of the computing resources provided by a multi-core CPU as an engine for data-parallel computation. The basic idea is that *applications exhibit a lot of parallelism through operations over collections of data*. Abstracting the underlying hardware threads, cores, and vector ISA as computation over collections of data greatly simplifies the task of expressing parallelism in an architecture-independent fashion. Ct provides a

nested data-parallel abstraction that is initially familiar to most programmers, but which provides extended functionality to address irregular algorithms.

Irregular algorithms are broadly defined as those that :

- require dynamic data types, such as sparse matrices, linked lists, or trees;

- have high likelihood of contended synchronization, such as reductions and prefix-sums in which elements of a collection are "summed" using a combining operator; and/or

- have moderate control flow, such as well-structured conditional nests, nested loops, and recursive functions.

Intel multi-core architecture (including tera-scale architecture) addresses these requirements efficiently, whereas GPUs generally do not. Ct aims to address the software stack by providing a data parallel model that supports irregular algorithms, whereas GPGPU programming models do not. An important benefit of Ct for the software developer is that it scales *forward* with increasing core count and vector ISA width. For example, a Ct application will scale from

1.3.1   The Importance of Determinism

Like many of its flat data-parallel brethren, Ct is *deterministic*. Determinism guarantees that program behavior is identical, on one core or many cores. This essentially eliminates an entire class of programmer errors—namely, data races. Ct also provides a predictable high-level programming model. As a result, the average programmer can use Ct operators with a basic understanding of the cost and scalability of their use. This is difficult to achieve for unconstrained threading models.

The Ct programming model accomplishes the difficult task of combining powerful, high-performance data parallel constructs with completely deterministic behavior. This is essential for tera-scale programming models, to enable the development of programs that are both powerful and easy to write.

Figure 1-1 The Ct API in the software development process.
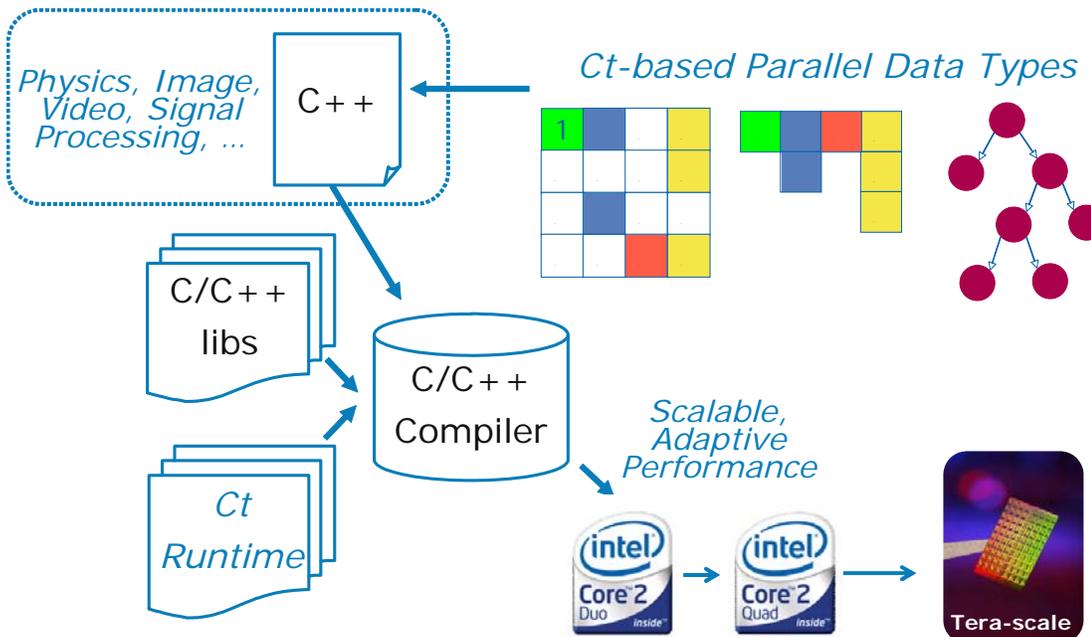
dual- and quad-core systems to tera-scale systems.

2.0 Sparse Matrices: An Example

2.1 The Challenge of Sparse Matrices

Many useful data structures have unusual (or irregular) organization because of the efficiency of such representations. For example, dense (or fully populated) matrices are required when the problem formulation dictates that most elements of the matrix are significant. However, often the particular algebraic formulation of a problem sparsely populates elements in the matrix with meaningful values. One example is large scale physics simulation. In these simulations, the logical size of a dense matrix might be hundreds of megabytes. By contrast, a sparse matrix representation that only stores non-zero matrix elements might only hold one megabyte of data. This degree of data compression is essential in desktops, mobile, and console gaming platforms.

Dense representations simplify the parallelization of such code. Walking through the elements in a dense representation is often performed in regular patterns, such as column or row order. This means that the control paths and data access patterns are very predictable (one feature of regularity in an algorithm). For example, the code in Example 2-1 can be used to walk over a dense matrix A in row order.

Example 2-1: Dense matrix traversal

```
for (row = 0; row < row_num; row++) {
    for (col = 0; col < col_num; col++) {
    …touch elements of A[row][col]…
    }
}
```

Generally there are two factors that make this a relatively manageable exercise in parallelization:

1. Depending on the algorithm being implemented, there may be parallelism in one or both of the enclosing loops. For large matrices, these loops provide sufficient parallelism that can be trivially decomposed into parallel sub-loops.

2. The data accessed within one or both of these loops is distinct (or *independent*, in compiler terms).

Sparse matrices are much more difficult to parallelize. There are many forms of sparse matrix. We will address two forms that pose unique challenges for programmers: Compressed Sparse Column (CSC) and Compressed Sparse Row (CSR) matrices.

The basic idea of CSC and CSR is to store only non-zero elements of the matrix, in column or row order, respectively. With each non-zero element, the programmer will also store the row or column index.

Consider the sparse matrix in Example 2-2. In CSR and CSC formats, the matrix would be stored as three vectors, the nonzero values, the row or column pointers, and the column and row indices, respectively. Schemas for traversing the two representations are shown in the Example.

Example 2-2: Sparse matrix representation and traversal

```
A = [[0 1 0 0 0]
     [2 0 3 4 0]
     [0 5 0 0 6]
     [0 7 0 0 8]
     [0 0 9 0 0]]
```

The CSR Representation of A:

```
 Values = [1 2 3 4 5 6 7 8 9]
 ColIdx = [1 0 2 3 1 4 1 4 2]
 RowP   = [0 1 4 6 8 9]
 for (row = 0; row < row_num; row++) {
     for (elt = RowP[row]; elt < RowP[row+1];
 elt++) {
         int col = ColIdx[elt];
         …touch elements of A[row][col]…
     }
 }
```

The CSC Representation of A:

```
Values = [2 1 5 7 3 9 4 6 8]

RowIdx = [2 1 3 4 2 5 2 3 4]

ColP  = [0 1 4 5 6 7 9]


for (col = 0; col < col_num; col++) {

   for (elt = ColP[col]; elt < ColP[col+1];
elt++) {

      int row = RowIdx[elt];

      …touch elements of A[row][col]…

   }

}
```

Two factors make this code difficult to parallelize efficiently.

1. The inner loop in both cases has a varying and unpredictable trip count. For example, the trip count might be determined by where a game player is looking in a particular scene in a game. This makes it difficult to predict the workload for each inner loop invocation. Thus it is difficult to balance the workload among threads.

2. There is an indirection through the column or row index array that will create aliases and dependences in most computations that might use sparse matrices.

Consider sparse matrix vector product (SMVP), a common kernel in gaming and RMS applications. In SMVP, a sparse matrix is multiplied by a vector. Like dense matrix vector multiplication, computing a sparse matrix vector product requires taking the inner product of each row of the matrix with the vector. For the CSR sparse matrix, the loop traverses the data in row order, similar to the dense computation. For the CSC sparse matrix, it is simpler to traverse the data in column order, updating the result vector when non-zero row elements occur.

It is worth observing some of the broader patterns of these computations in order to comprehend the implications for parallelism:

- In CSR, the expression *vec[ColIdx[elt]]* denotes that *vec* is permuted by the contents of *ColIdx*.

This looks similar to a *gather* operation in GPUs.

- In CSC, the expression *vec[col]* implied that each element of *vec* must be replicated (*ColP[col+1]-ColP[col]*) times. This can be viewed as a special kind of *gather* operation, but more complex than those that are typically supported in GPU hardware.

- In CSR, the left-hand side expression "*product[row]*+= …" denotes that we are summing (or *reducing*) all the elements computed in the right-hand side of that expression for the inner loop.

- In CSC, the expression "*product[RowIdx[elt]]* +=…" implies that we are performing what is called a *combining-send*, or alternatively a *multi-reduction* or *combining-scatter*.

Note that the last two observations above are similar, except that the data is effectively pre-sorted by destination for the CSR form. That is, the inner product to compute an element of the result for CSR occurs entirely within one invocation of the inner loop, whereas it takes place across many invocations of the inner loop for CSC.

We can now create a conceptual parallel pattern for performing sparse matrix computations: *First, permute the vector, then perform an element-wise product, and finally, perform some flavor of reduction.*

## 2.2 Unique Challenges for GPUs

Special purpose processors often lack many of the mechanisms required for efficient implementation of sparse matrix kernels. For example, GPUs lack the basic facilities required to support efficient, low-latency cross-chip inter-thread communication and synchronization. Cache coherence provides a vehicle for core-to-core communication, while high-performance interconnects provide low latency.

Inter-thread synchronization and communication of partially computed results is essential to support collective communication primitives, such as reductions and prefix-sums. For example, to reduce the values of an array, the standard approach is to use an algorithm to partition the elements of the array among threads,

reduce the values locally, and then combine the partial results from each thread. To combine the partial results from each thread, the threads must be able to communicate with each other.

For a general purpose multi-core architecture, synchronization and communication between threads requires the use of coherent memory locations and locks; no external memory bandwidth is consumed. For a GPU, on the other hand, the algorithm is quite costly in terms of memory bandwidth. The only mechanism for inter-thread synchronization and communication is through access to external memory. The GPU reduction algorithm must then repeatedly write and then read partial results to and from external memory. For an array of length *n*, the algorithm uses a progressive combining scheme whereby adjacent pairs of elements are combined with the results written out to memory in each phase. The first phase reduces to *n/2* elements, the second to *n/4* elements, and so on. In all, this requires *O(log n)* passes through memory, consuming *O(n*log n)* memory bandwidth.

Sparse-matrix operations require more complex flavors of collective communication primitives, such as *segmented reductions*, *multi-reduce*, and *prefix-sum*. Operations such as these are significantly more complex on special purpose processors, requiring additional passes through memory. This is a principal reason why such primitives historically have been restricted or unavailable on GPGPU programming
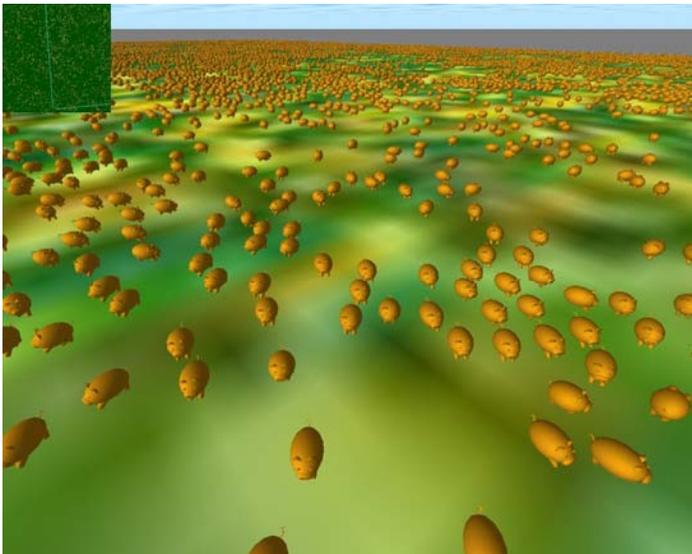
models.

on GPGPU programming models.

2.3 Sparse Matrices with Ct

Ct introduces a new (template-style) polymorphic type, called a TVEC. TVECs are write-once vectors that reside in a vector space segregated from native C/C++ types. For example, a vector of non-zeros for the stiffness matrix in a cloth simulation may be declared as:

```
TVEC<F64> nonzeros;[1]
```

The column indices for each nonzero and row sizes for the sparse matrix (assuming a Compressed Sparse Column representation) can be represented similarly:

```
TVEC<I32> RowIdx;
```

```
TVEC<I32> ColP;
```

The types of primitives used for this particular sparse matrix-vector product fall into three of the Ct operator categories:

- *Element-wise* operators that support simple unary, binary and n-ary operators, such as addition, multiplication, etc. For example, the following code performs an element-wise multiplication of two vectors:
  ```
  TVEC<F64> product = multiply(nonzeros,
  expv);
  ```
  …or, using operator overloading:
  ```
  TVEC<F64> product = nonzeros*expv;
  ```

- Collective communication operations, such as reduction, prefix-sum, or combining-send. Multi-core CPUs support these very efficiently, though many accelerators (including GPUs) support only a few of these operations efficiently. For example, to perform an indexed reduction, or a combining-send, of a vector, we use a multi-reduce primitive. In Ct, this primitive is invoked implicitly via reductions for indexed vector shapes. We apply the vector shape and perform the reduction:
  ```
  product =
  product.applyNesting(RowIdx,ctIndex);
  TVEC<F64> innerproduct =
  addReduce(product,RowIdx);
  ```

---

[1] For cloth simulation, we are likely to use a block sparse symmetric matrix, but we elide these details for simplicity of presentation.

- Permutation operations which allow both structured and unstructured reordering and replication of data. For example, the following code creates a new vector expv comprised of a variable number of copies (denoted by each element of cols) of each element of v:

```
TVEC<F64> expv = distribute(v,ColP);
```

Combining the code above, we can write a sparse matrix-vector product, as illustrated in Example 2-3.

Example 2-3: Sparse matrix vector product with Ct

```
TVEC<F64> sparseMatrixVectorProducSC(

  TVEC<F64> Values,

  TVEC<I32> RowIdx,

  TVEC<I32> ColP,

  TVEC<F64> v) {

    TVEC<F64> expv = distribute(v,ColP);

    TVEC<F64> product = Values*expv;

    product = product.applyNesting(RowIdx,
ctIndex) ;

    TVEC<F64> result = product.addReduce();

    return result;

}
```

3.0 The Ct API

3.1 Ct Vectors: TVECs

The TVEC, which is the basic type of vector in the Ct API, is a managed parallel vector. TVECs are allocated and managed in a segregated memory pool, to ensure the safety of parallel operation on the vectors. Normal C and C++ data structures are generally unsafe because of unrestricted effects and use of aliases.

Data must be explicitly copied into and out of vector space. The only operators allowed on TVECs are Ct operators, which are functionally pure. It is important to note that TVECs are logically passed around by value. This property guarantees the safety of parallelism and the aggressive optimizations that make parallelism efficient.

The base types of TVECs are drawn from a set of typical pre-defined scalar types. A TVEC variable may be declared without an instantiated base type, but the compiler must be able to infer the type. Examples of base types include I32 (32-bit integer), I64 (64-bit integer), F32 (Float), F64 (Double), and Bool (Boolean). Ct also includes a C struct-like base type for user-defined base types (such as color pixels) and fixed size arrays.

The internal representation of TVECs is opaque to the programmer and may include meta-data that is useful to the runtime. A TVEC may be declared as follows:

```
TVEC<I8> Red;

TVEC<F32> Xes;
```

Values in native C/C++ memory space are explicitly copied into and out of Ct managed vector space using copyin and copyout operators:

```
Red = copyin(CRed, Height*Width, I8);  //
Red ← CRed

copyout((void*)CXes,Xes);  // CXes ← Xes
```

There are several flavors of each of these operators for different data types and shapes.

3.2 Ct Operators

Ct operators are logically free of side effects, from the programmer's perspective. As such, each Ct operator logically returns a new TVEC (note that the C++

operator overloading is used to write "cleaner" or more readable code):

```
ScaledRed = Red*0.5; // ScaledRed  ←a new
TVEC
```

The API encompasses a broad range of rich functionality. Within each class of facility, element-wise, collective communication, and permutation operators, there are many subclasses of operations, each defined over all TVEC types. Each Ct operator generally has the form ctOpClass, where Op is the particular flavor of operator and Class is the operator class. For example, addReduce denotes the reduction operator using addition.

3.2.1 Element-wise Operators

Element-wise operators are typically referred to as "embarrassingly" parallel, requiring no interactions between the computations on each vector element. In functional languages, these can be implemented with map operations, while in OpenMP they can be implemented as parallel for loops. An example of an element-wise operation is the addition of two vectors:

```
TVEC<F32> A = B + C; // "+" resolves to add
```

Note that this code generically performs an element-wise addition of two vectors, regardless of the "shape" of the two vectors (i.e., their length, dimensionality, irregularity).

3.2.2 Collective Communication Operators

Collective communication operators tend to provide distilled computations over entire vectors and are highly coordinated. While they have a high degree of interference, they can be structured so that there is parallelism in colliding writes, and they typically scale in performance linearly with processor count, with little or no hardware support. These operators are called collective communication operators in MPI and reductions in OpenMP, though neither provides the rich set of operations that Ct does. In functional languages, these are termed *fold operations* or *list homomorphisms*.

There are two kinds of collective communication primitives in general, though there are several variations of each depending on the type of vector on which operations are being performed. Broadly, the

two fundamental types of operations are reductions and prefix-sums (also called scans). Reductions apply an operator over an entire vector to compute a distilled value (or values, depending on the type of vector). Prefix-sums perform a similar operation, but return a partial result for each vector element. For example, a addReduce sums over all the elements of a vector if the vector is flat. More concretely, `addReduce([1 0 2 -1 4])` yields 6. Likewise, `addPrefix([1 0 2 -1 4])` yields `[0 1 1 3 2]`.

If the vector is nested, the behavior will be as described in Section 3.2.4.

3.2.3 Permutation

A permutation operator in Ct is any operator that requires moving data from its original position to a different position. An example of this is a gather operation, which uses an index array to collect values of a vector in a particular order. Permutations run the gamut, from arbitrary permutations with arbitrary collisions (occurring when two values want to reside in the same location) to well-structured and predictable permutations where no collisions can occur. For collisions, it is recommended that programmers make use of the collective communication operators. An example of a well-structured (and efficient) permutation operator is pack, which uses a flag vector to select values from a vector in the source vector order. Hardware can typically support this operator fairly efficiently.

3.2.4 Nested Vectors

Ct's support for nested vectors is a generalization that allows a greater degree of flexibility than is otherwise found in most data parallel models. TVECs may be flat vectors or regular multi-dimensional vectors. They also may be nested vectors of varying length, which allows for very expressive coding of irregular algorithms, such as other variants of sparse matrix representations, or byproducts of divide-and-conquer algorithms.

The vector value `[a b c d e f]` is a flat (or 1-dimensional) vector. The vector `[[a b][c d e f]]` holds the same element values, but is a vector of

two vectors of lengths 2 and 4. The second vector might represent a partitioning of the first vector's data based on certain attributes (such as a threshold value).

Ct operators work on nested TVECs seamlessly. The behavior of element-wise operators is the same for nested TVECs as for flat vectors. For example, `add([[a b][c d e f]], [[g h][i j k l]])` yields `[[a+g b+h][c+i d+j e+k f+l]]`. The power of nested versus flat TVECs is primarily differentiated through the behavior of collective communication primitives.

Collective communication primitives applied to nested TVECs "respect the boundaries" of the subvectors by applying the operator to each subvector independently. For example, `addReduce([a b c d e f])` yields the singleton vector `[a+b+c+d+e+f]`, while `addReduce([[a b][c d e f])` yields the two-element vector `[a+b c+d+e+f]`.

The internal representation of a nested vector is opaque to the programmer. The nesting structure is accessible through several facility routines, but these routines generally should not be necessary. Other vector types supported by Ct are indexed vectors, where there are indices associated with each value.

3.3 Implementing Ct

Ct provides several levels of abstraction below the high-level API presented in this paper, to accommodate varying degrees of programmer expertise. For instance, for more sophisticated programmers, lower levels of abstraction expose task granularity and machine-width independent vector ISA intrinsics and optimizations.

Earlier generations of data parallel programming models and languages benefited from extraordinarily large data sets and highly parallel (high bandwidth) memory systems. Because of this, the optimization pressure on task granularity was not as critical as simply identifying parallel regions of code and basic operator implementations. For tera-scale architecture, such implementations would yield code that is memory bandwidth limited and overly burdened with threading overhead. The computational power of tera-scale systems must be coupled with intelligent optimization

that maximizes the amount of calculation per memory operation.

The top priority of the Ct compiler and runtime is to minimize threading overhead and make effective use of memory bandwidth. To accomplish this, Ct utilizes a fine-grained, data-flow threading model. Essentially, the Ct computation is decomposed into a task dependence graph that is optimized by the compiler by merging similar tasks into coarser-grained tasks.

The task dependence graph consists of data parallel *sub-primitives*. These sub-primitives are the building blocks of data parallel computing, comprising *local* phases of computation that entail no inter-processor synchronization and *global* phases that perform structured write combining and synchronization. Similar sub-primitives usually can be fused together into coarser grained tasks. This simultaneously increases task granularity and locality of data access, minimizing off-chip memory accesses.

Another distinction of Ct's approach is that threading decisions are made dynamically. Each task is represented by a spawn point, but the precise number of sub-tasks created is dependent on both the number of cores available and the size of the underlying vector being processed. The Ct runtime is highly adaptive to varying data sizes and core loads.
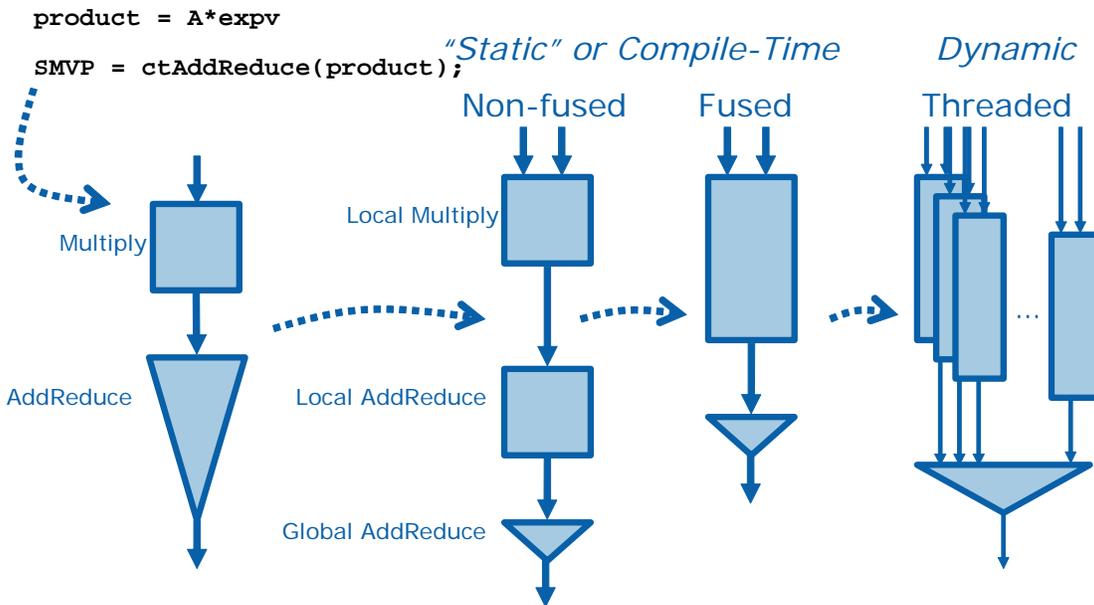
```
product = A*expv

SMVP = ctAddReduce(product);
```

Figure 3-1 How Ct is compiled.

12

4.0 Concluding Remarks: The Future of Ct

Ct implements many more features than described in this white paper, and it enables several additional key features that will be implemented in the coming months. One important future feature is *deterministic task parallelism*. As previously observed, determinism is a critical property for programmer productivity, guaranteeing that program execution is functionally predictable. Through determinism, data races are entirely eliminated as a class of programmer errors.

The underlying threading model used by Ct supports a fine-grained, adaptive dependent tasking model. While this is used to implement the higher-level data parallel constructs, revealing implicit task parallelism, Ct will include constructs for deterministic task parallelism.

Ct supports a high-level performance model that can be used by the average programmer to guide algorithmic choice. For example, element-wise operators generally scale linearly in performance on Intel multi-core and tera-scale architecture for large vectors. If the operations are used for small vectors, the underlying runtime knows not to use as many cores for the computation, mitigating threading overhead. Similarly, collective communication operators generally scale linearly with core count, but have an additional cost associated with synchronization. The synchronization patterns vary by architecture, but are generally asymptotic in core count, not in vector size. So, for large vectors, the linear scaling component tends to dominate.

Ct provides a virtual laboratory in which to experiment with more exotic tera-scale programming features, such as lossy, real-time, and adaptive computation. Some of these features are used on an *ad hoc* basis in many high- performance algorithms and applications, notably those with real-time

constraints, such as media and gaming. A key objective of Ct is to create a framework in which such performance breakthroughs (sometimes pejoratively and unfairly misunderstood as *hacks*) are supported systematically.

5.0 Selected Bibliography

Guy Blelloch. Vector Models for Data-Parallel Computing. MIT Press. ISBN 0-262-02313-X. 1990.

Guy E. Blelloch, Siddhartha Chatterjee, Jonathan C. Hardwick, Jay Sipelstein, and Marco Zagha. Implementation of a Portable Nested Data-Parallel Language. Journal of Parallel and Distributed Computing (JPDC), 21(1), April 1994.

Manuel M. T. Chakravarty and Gabriele Keller and Roman Lechtchinsky and Wolf Pfannenstiel. Nepal --- nested data parallelism in Haskell. In R. Sakellariou, J. Keane, J. R. Gurd, and L. Freeman, editors, Proc. 7th International Euro-Par Conference, volume 2150 of Lecture Notes in Comput. Sci., pages 524--534. Springer-Verlag, 2001.

Allan L. Fisher , Anwar M. Ghuloum, Parallelizing complex scans and reductions, Proceedings of the ACM SIGPLAN 1994 conference on Programming language design and implementation, p.135-146, June 20-24, 1994, Orlando, Florida, United States.

Daniel P. Friedman and David S. Wise, Aspects of applicative programming for parallel processing. IEEE Transactions on Computers, C-27(4):289–296, Apr. 1978.

Anwar M. Ghuloum, Allan L. Fisher, Flattening and parallelizing irregular, recurrent loop nests, ACM SIGPLAN Notices, v.30 n.8, p.58-67, Aug. 1995.

Matthew Hammer, Umut A. Acar, Mohan Rajagopalan, Anwar Ghuloum, A Proposal for Parallel Self-Adjusting Computation, In Proceedings of the Workshop on Declarative Aspects of Multicore Programming (DAMP 2007), January 2007

David A. Krantz, Robert H. Halstead, Jr., and Eric Mohr, Mul-T: a high-performance parallel lisp. In Proceedings of the SIGPLAN '89 Conference on Programming Language Design and Implementation, pages 81–90, 1989.

Byoungro So, Anwar Ghuloum, Youfeng Wu. Optimizing data parallel operations on many-core platforms. First Workshop on Software Tools for Multi-Core Systems (STMCS), Manhattan, NY, 2006, pp.66--70.

Leslie G. Valiant. A bridging model for parallel computation. Communications of the ACM, 33(8):103--111, August 1990.

6.0 Appendix: Ct Coding Examples

6.1 Image Processing

Image processing generally is fairly straightforward in data parallel programming models, even those that are flat. There is a slight difference in coding style between the "streaming" style of local kernel specification and the global data parallel approach. In the kernel style, only the local computation is specified and is assumed to be applied at every pixel (similar to the graphics model). In the data parallel style, the entire image is operated on at once, though this approach can be relaxed through the use of Ct's generic operators (e.g., genBinary). However, the code generally is not substantially more or less compact with either approach, as illustrated in the color conversion code below.

```
TVEC<F32> colorConvert(TVEC<F32> rchannel, TVEC<F32> gchannel,
                       TVEC<F32> bchannel, TVEC<F32> achannel,
                       F32 a0, F32 a1, F32 a2, F32 a3)
{
    return (rchannel * a0 + gchannel * a1 + bchannel * a2 + achannel * a3);
}
```

Convolutions are somewhat more substantial and broadly useful in both image processing and more general signal processing domains. Because convolutions require a neighborhood of pixels to compute the filter, the source image is "shifted" about in order to place the required pixel at the computed pixel. These shifts are logically creating new values, though in this case, the compiler and optimizer trivially optimize away any copying and simply refer to a single source image. Following is an example of convolution code:

```
TVEC<F32> Convolve2D3x3(TVEC<F32> pixels, I32 channels, TVEC<F32> kernel) {
TVEC<F32> respixels;

    // directions[m][n] is a constant TVEC of size 2 with values {m-1, n-1}
  respixels += shiftPermute(pixels, directions[0][0]) * kernel[0][0];
  respixels += shiftPermute(psixels, directions[0][1]) * kernel[0][1];
  respixels += shiftPermute(pixels, directions[0][2]) * kernel[0][2];
  respixels += shiftPermute(pixels, directions[1][0]) * kernel[1][0];
  respixels += pixels * kernel[1][1];
  respixels += shiftPermute(pixels, directions[1][2]) * kernel[1][2];
  respixels += shiftPermute(pixels, directions[2][0]) * kernel[2][0];
  respixels += shiftPermute(pixels, directions[2][1]) * kernel[2][1];
  respixels += shiftPermute(pixels, directions[2][2]) * kernel[2][2];

  return respixels
}
```

6.2   Sparse Linear Solvers

Sparse linear solvers are quite common in high-performance applications, such as physics simulations and many scientific and machine learning applications. One technique that is commonly used is the preconditioned conjugate gradient method, illustrated in the Ct code below, which produces a cloth simulation. Note that the key kernel used is a CSR sparse matrix vector product.
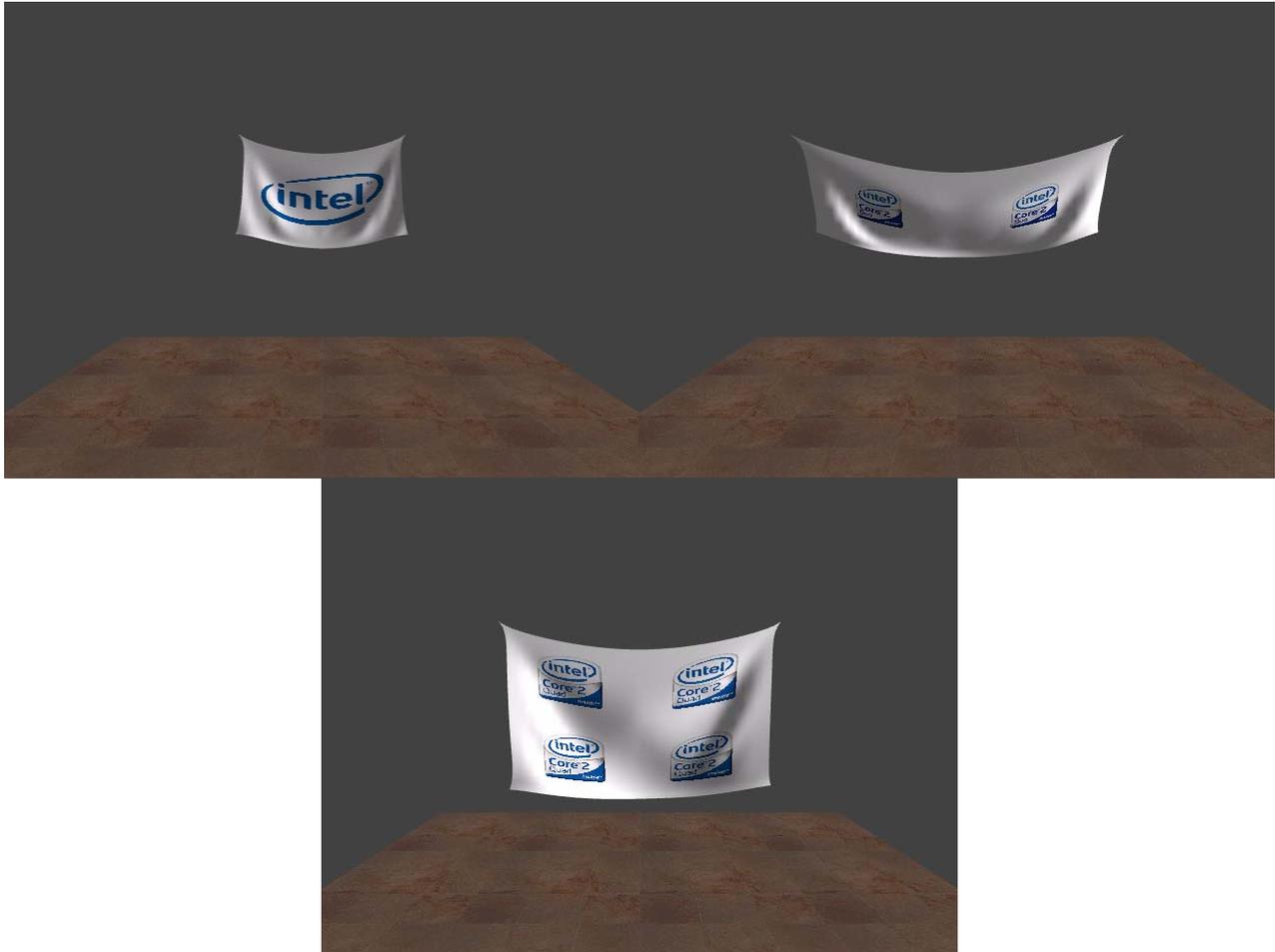

Figure 6-1 Ct-based Cloth simulation on Core 2, Core 2 Duo, Core 2 Quad

First, we define a C structure to contain the linear system.

```
typedef struct {
    CTSparseMatrix          A;
    TVEC<F64>               b;
    Float                   e;          // Epsilon

    CTSparseMatrix          pM;         //Preconditioning matrix.
    CTSparseMatrix          pMinv;      //Inverse preconditioning matrix.
} CTLinearSystem;
```

The solver uses the sparse matrix vector product kernel (called smvMul below).

```
TVEC<F64> pCG(CTLinearSystem &lsys, TVEC<F64> x0){

    int i = 0;

    F64 alpha, delta0=0, delta1;

    TVEC<F64> v_x  = x0;

    TVEC<F64> v_r  = lsys.b - smvMul(lsys.A, v_x);

    TVEC<F64> v_pr = smvMul(lsys.pMinv, v_r);

    delta1 = addReduce(v_r * v_pr);

    while( (delta > delta0 * lsys.e) && (i < IterationMax)){

        TVEC<F64> v_q  = smvMul(lsys.A, v_pr);

        alpha        = addReduce((v_pr * v_q));

        alpha        = delta/alpha;

        v_x          = v_x + (v_pr * alpha);

        v_r          = v_r - (v_q * alpha);

        TVEC<F64> v_s  = smvMul(lsys.pMinv, v_r);

        delta0       = delta1;

        delta1       = addReduce(v_r * v_s);

        v_pr         = v_s + (v_pr * (delta1/delta0));

        i++;

    }

    return v_x;

}
```

This cloth simulation requires a bit more code to integrate forces in the system and detect collisions. Collision detection, in particular, is extremely challenging to parallelize for flat data parallel systems, but simplified greatly through nested data parallelism.

6.3 Sorting

While sorting is typically used as an illustrative algorithm, the kernel illustrated below forms part of the implementation for quick KD-tree construction in Ct. The quicksort is simpler to use to illustrate the tradeoffs in implementation.

The problem with recursive sorting (and divide-and-conquer algorithms, in general) is that the (superficial) data parallelism is maximized at the root of the algorithm's call graph and minimized at the leaves. Similarly, task parallelism is (superficially) minimized at the root and maximized at the leaves. Taking the task versus data parallelism view of quicksort, it appears to be difficult to fit into a single programming model. The approach is illustrated in the code and Quicksort graphic below.

```
TVEC<F64> ctQsort(TVEC<F64> Keys) {

  TVEC<F64> pivot, lowerKeys, pivotKeys, upperKeys;

  TVEC<Bool> pivotFlags;

  I32 pivot;
```

```
   if (length(Keys) == 0)

      return Keys


      pivot = extract(Keys, 0)


      pivotFlags = lessThan(Keys, Pivot);

      lowerKeys = pack(Keys, pivotFlags);

      pivotFlags = equal(Keys, Pivot);

      pivotKeys = pack(Keys, pivotFlags);

      pivotFlags = greaterThan(Keys, Pivot);

      UpperKeys = pack(Keys, pivotFlags);


      return cat(ctQsort(lowerKeys),

        cat(pivotKeys,ctQsort(upperKeys));
}
```
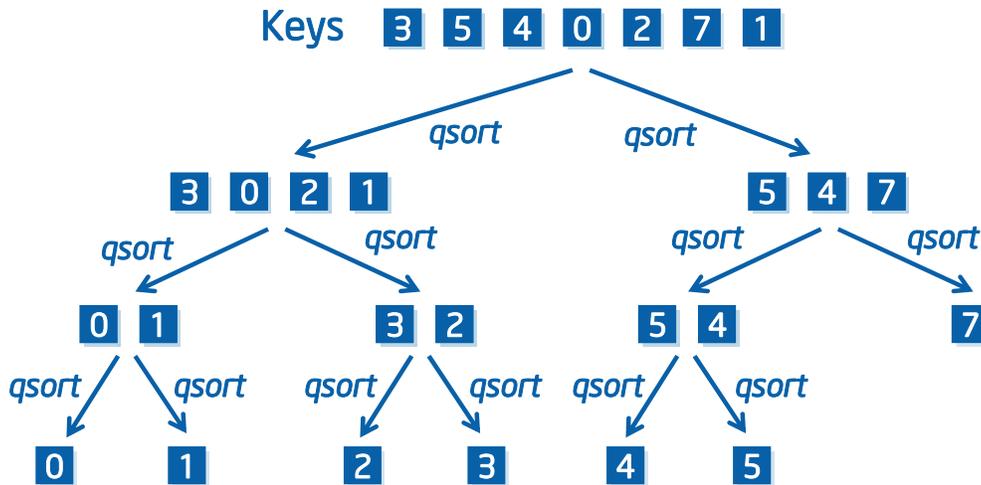

Figure 6-2 Decreasing data parallelism and increasing task parallelism in Quicksort.

Nested data parallelism unifies data parallelism and divide-and-conquer task parallelism. Sub-tasks are not created as parallel threads, but rather are manifest as partitions in a nested vector. We use partition to create these sub-partitions, and then recurse on the entire vector. The power of nested data parallelism for this type of irregular control flow is clearly illustrated through the resulting unification of data and task parallelism, as shown below:

```
TVEC<F64> ctQsort(TVEC<F64> Keys, TVEC<Bool> Mask) {

  TVEC<F64> pivot, partitionedKeys,

  TVEC<Bool> newMask;

  TVEC<I32> pivotPartitions

  if (equals(orReduce(orReduce(Mask)), 0)) // since mask is nested, have to do this twice

     return flatten(Keys);
```

```
    pivot = extract(Keys, ctNewVector(ctNumPartitions(Keys),0,I32));


    pivotPartitions = compare(Keys, Pivot, Mask);

    partitionedKeys = partition(Keys, pivotPartitions);

    partitionedMask = partition((Mask && pivotPartitions), pivotPartitions);


    return ctQsort(partitionedKeys, partitionedMask);
}
```
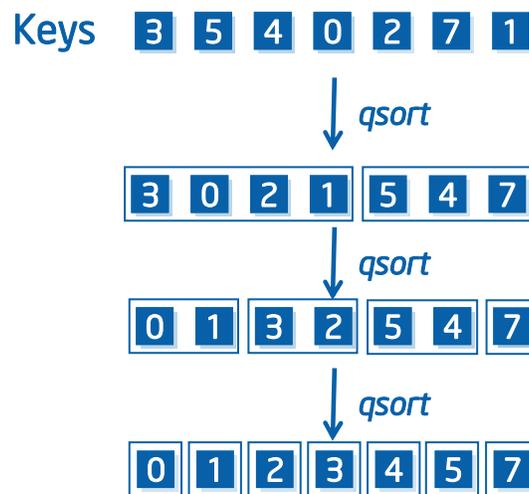
Figure 6-3 Unifying Quicksort's data and task parallelism in Ct via nested data parallelism.

7.0 Appendix: Typical Ct Operators

Table 7-1   A selection of typical Ct Operators

Facilities
Managed Vector/Native Space Copying
copyIn, copyin2D, copyin3D, copyout
Vector Generators
cat, repeat, replicate, replace, index, copy, newVector
Vector Utilities
extract, copy, length
Nested Vectors
newNestedVector, applyNesting, copyNesting, setRegular2DNesting, setRegular3DNesting, getNesting, getNestAsVec

Element-wise
Vector-Vector
add, sub, mul, div, equal, min, max, mod, lsh, rsh, greater, less, geg, leq, neq, ior, and, xor, power, divTan, select, map
Vector-Scalar (also, Scalar-Vector variants exist)
addVectorScalar, subVectorScalar, subScalarVector, mulVectorScalar, divVectorScalar, divScalarVector, equalVectorScalar, minVectorScalar, maxVectorScalar, modVectorScalar, lshVectorScalar, rshVectorScalar, greaterVectorScalar, lessVectorScalar, geqVectorScalar, leqVectorScalar, neqVectorScalar, iorVectorScalar, andVectorScalar, xorVectorScalar, genVectorScalar, map
Unary
abs, not, log, exp, sqrt, rsqrt, sin, cos, tan, asin, acos, atan, sinh, cosh, tanh, floor, ceiling, round, map

Collective Communication
Reduction
addReduce, mulReduce, minReduce, maxReduce, andReduce, iorReduce, xorReduce, reduce
Scan/Prefix-Sum
addScan, mulScan, minScan, maxScan, andScan, iorScan, xorScan, scan

Permutation
Pack/Unpack
pack, unpack
Scatter/Gather
scatter, gather
Shift/Rotate
leftShiftPermute, rightShiftPermute, leftRotatePermute, rightRotatePermute, shiftDefaultPermute, rotateDefaultPermute
Partition
partition, unpartition
Miscellaneous
defaultPermute, omegaPermute, butterflyPermute, distribute