Parallelizing a Sparse Domain Distribution in Chapel

Randy Dodgen
CS380P Spring 2010 - Final Project

May 17, 2010

Cray, Inc.'s Chapel programming language seeks to provide syntactic and library support for a variety of parallel-programming concepts. Data-parallel applications are supported via the concepts of domains and distributions. A Chapel domain represents some set of indices, under which arrays may be declared and accessed. These domains can be used to represent dense, arithmetic sets of indices, as one would find in 'arrays' of many other languages. However, Chapel domains extend further, and can describe other varieties of index sets - sparse, for example. In Chapel, a distribution provides a storage representation for domains and their associated arrays of data, and may expose concurrency in how it provides access to the stored indices and data elements. The focus of this project is extension of one of Chapel's currently-serial sparse distributions to admit more concurrency.

Please also see the `README` in the `randy/` directory of the submission.

1 Motivation

The current Chapel release includes a number of distributions. For example, the block distribution can perform a typical block allocation of data to some number of machines. The block distribution, however, supports only arithmetic domains. Support exists for sparse domains, but the needed distributions have not been fully implemented. While the block distribution provides concurrency when accessing elements distributed across machines as well as those on the same machine, the current sparse distributions have not yet been parallelized.

This project entails extension of Chapel's sparse distributions. It was selected due to the potential benefit to the Chapel language, as well as the opportunity to get familiar with the languages strengths and weaknesses.

Specifically, the goals were to extend the existing compressed sparse row (CSR) distribution to take advantage of multiple cores, and then multiple machines (locales). The multiple-core version was planned as a milestone on the way to a complete multiple-locale version.

2 Distribution Structure

2.1 Distributions, domains, and arrays

The Chapel project aims to support user-defined distributions, written in Chapel. All distributions provided in the current release are written this way. The Chapel library provides base classes to represent distributions, domains, and arrays, with a defined interface by which glue code and the compiler can employ a distribution.

To define a distribution, one provides at minimum three classes:

- The distribution, which must act as a factory for creating domains. A distribution class may implement one or multiple factory methods, corresponding to the domain types supported (sparse, arithmetic, etc.)
- The domain, which defines a set of indices. Major operations include tests for membership, iteration, and in the case of sparse domains, addition and removal of individual indices from the set. Domains must also behave as a factory for arrays. There may be multiple domains associated with a single distribution instance.
- The array, which translates indices in the domain to actual data. The array must provide iteration over its data elements, as well as a means to translate a valid index into a data element. There may be multiple array instances per domain.

2.2 Leaders and Followers

In addition to providing the means of storage, domains and arrays are themselves responsible for introducing concurrency during iteration. The end-user of a distribution (and its generated domains and arrays) is presented with a simple interface to this concurrency; he declares blocks of parallel iteration, with iterations understood to occur in no guaranteed ordering. For example, for a domain `myDom`, the loop body will execute for each index in the domain:

```
forall i in myDom { ... }
```

Iteration over arrays produces values, instead of indices. Of course, values and indices in isolation are of limited utility. Iteration of multiple items can be 'zippered':

```
forall (i, val) in (myDom, myArray) { ... }
```

Assuming that `myArray` is an array with the domain `myDom`, this example will produce a loop iteration per pair of index and value, and `myDom(i)` would equal `val`. This relation suggests that the parallel iteration is not entirely independent and unconstrained. Zippering domains, arrays, and other iterable constructs in this manner requires that each participant agree on iteration order to some degree.

The distribution interface has a concept of 'leader' and 'follower' iterators. This tackles the issue of multiple iterable items, each with their own ideas of how to manage concurrency; in the prior example, one of `myDom` and `myArr` serves as the leader, and both serve as followers. The leader uses Chapel's constructs for task and locality management to spawn tasks (perhaps also on remote locales), and within each task, generates items to control the follower. Each generated item invokes all of the follower iterators. For a particular item, the followers must agree on a semantically meaningful, serial iteration order; followers processing the same item from the leader have their generated data 'zipped' together, and each of the resulting tuples is a result item for the parallel zippered iteration (in the examples above, each tuple induces a loop iteration).

Consequently, for some set of iterable items to be zippered, they must be able to agree on a protocol for the data being passed from leader to followers. This is an issue that needs attention in the future; currently, interoperability between arrays and domains of different types is limited.

3 Design

The CSR distribution as packaged in the Chapel 1.1 distribution was modified in two stages - first to produce a multicore version, and then further to produce a multiple-locale version.

3.1 Multicore

The original CSR distribution generated no concurrency. The leader iterators generated a single, meaningless item in order to transfer control to single instances of the followers. Effort in producing a multicore version centered on re-working the leader / follower iterators in the domain and array classes to admit some amount of concurrency. The multi-core design was successfully implemented, although performance results were disappointing (See results).

3.1.1 Work Division

The internal structure that CSR prescribes is conductive to iteration across rows, rather than columns. Iterating over an entire row requires only determining the start and stop indices within the data arrays. In fact, since adjacent rows are stored contiguously, one can iterate across a range of rows, given only the start of the first row and the end of the last row. As such, the multicore CSR was designed to divide work on row boundaries.

A function was implemented to divide a CSR domain's row range into some number of chunks of adjacent rows, as evenly as possible. Since each chunk corresponds to work for a parallel task, the number of chunks in which to divide the row range must by convention respect a number of run-time configuration parameters (controlling the maximum tasks per locale for data-parallel iterators, etc.)

3.1.2 Leader / Follower Protocol

The default distribution for sparse domains (separate from CSR, although also fully serial) currently makes assumptions regarding interoperability (as mentioned, interoperability is an open issue). The default sparse distribution requires and enforces that its leaders / followers only pair with leaders / followers of directly related arrays or domains (an array can pair with a sibling array, or its own domain).

For simplicity, these assumptions were adopted into the multicore CSR implementation. Given those, any leader / follower protocol could be adopted among the CSR classes. The protocol implemented involves a simple wrapper type, which contains a row range. The domain and array leaders generate instances of it based on the work division described, and followers can walk their CSR structures with the benefit of contiguous rows.

This approach took advantage of Chapel's pervasive usage of generic types / functions, and its function resolution rules. Incompatible leaders / followers would eliminate the 'real' leaders / followers (due to their specific type requirement, for the private CSR type), and instead attempt to instantiate the general, type-unspecified versions. These versions, on selection, generate a descriptive error at compile time. This method was adapted from the existing CSR and default sparse distribution code.

3.1.3 Limitations

A number of shortcomings are detailed in the source for the multicore CSR distribution. Key issues are outlined here.

For simplicity in work division, the runtime parameter controlling minimum granularity is ignored - it would be preferable to observe it in some fashion, to allow control over the thread to data ratio.

The work division is based on evenly allocating number of rows, rather than number of elements. It may be more effective to allocate at row granularity, but to select work-item boundaries so as to have approximately the same number of non-zeroes in each work-item.

3.2 Multi-locale

3.2.1 Idiomatic Multi-locale Structure

The multi-locale block distribution was used as a model for re-structuring the CSR distribution for multiple-locale support. The pattern seen in the block distribution is presented in all of the multi-locale distributions examined.

In the idiomatic structure, each of the three classes forming a distribution package - array, domain, and distribution - gain a 'local' variant. The local companions of the three main classes exist per-locale, and store the actual data. They do not implement the actual distribution interface, nor do they inherit the associated base classes; they are purely a structural idiom for internal bookkeeping.

The main classes, in implementing the distribution interface, typically delegate to a particular, local instance corresponding to the right locale. To facilitate this, each class keeps an array of its local instances, indexed under the same domain as the array of target locales. For example, in response to the call to access an item by index, the block distribution calculates the owning block (and thus locale) for that index, and retrieves the data from that locale's instance.

3.2.2 Implementation Status

Extension to multi-locale operation from the multi-core base was not completed; most of the implementation is finished, but needs debugging and testing. The multi-core distribution was modified to split each of the main classes into global and local versions. The global versions populate their arrays of per-locale local instances, and delegate to them for most operations. A few operations were disabled (and intentionally crash if called), and are annotated as such in the source.

3.2.3 Work Division

The issue of work division for a distribution has the challenge that it must support multiple domains at once, and make considerations for efficient zippering of them and their arrays. Choosing new work divisions as domains are added would be costly.

The multi-locale CSR takes another hint from the block distribution. The block distribution takes a `boundingBox` parameter on construction, allocates blocks of it per locale, and subsequently uses that allocation as a mapping from domain index to owning locale. For some of the domains sharing an instance of the block distribution, there may exist domain indices outside of the bounding box. Indices outside of the bounding box map to the 'closest' locale. Thus, for a poor pairing of bounding-box and domain, the distribution could map all indices in that domain to the same locale.

Multi-locale CSR behaves similarly, but with a restricted problem. Since all CSR domains are of rank 2, and since rows are allocated as indivisble units, its 'bounding box' equivalently can be a range of rows. The user specifies this as `distributedRowRange` to the CSR constructor. Rows are divided as evenly as possible inside the distributed row range; the locales owning the highest and lowest rows of this range are responsible for rows outside of the distributed range.

4 Benchmark

Sparse matrix-vector multiplication was selected to provide a performance benchmark for the CSR modifications. It was selected since it plays to CSR's strengths. Since each unit of execution in parallel iteration of a CSR domain, as implemented, walks across contiguous rows, each execution unit will simply walk the dense vector with a regular pattern of access. Since there are no dependencies between elements in the sparse matrix, only an efficient parallel iteration is needed; expense lookups by element index are not required.

The multiplication loop went through a few iterations. The first was elegant, utilizing slicing of domains via ranges, and a plus-reduction:

```
var result : [1..matrix_rows] real =
    [ row in 1..matrix_rows ] + reduce (matrix(row, 1..) * vector);
```

Unfortunately, though this worked for dense domains, slicing is not currently supported for sparse domains. Further, there was no clear guarantee that `vector` would be accessed in the multiple-locale case - each parallel iteration may induce communication to the locale owning `vector`. Though at some point, there may exist an automatic optimization to replicate `const` arrays, this is currently not the case (this was determined via Brad Chamberlain).

The final iteration is instead:

```
forall ((i,j), val) in (MatrixDom, matrix) do local {
    localResultVectors(here.id)(i) +=
        val * replVector(here.id)(j);
}
```

This introduces a number of optimizations. A special `Private` distribution (applied to `localResultVectors` and `replVector`) is used to provide local result vectors to accumulate into, and replicated copies of the vector. The `Private` distribution can only produces one-dimensional arithmetic domains, with an index per locale The data for each index is stored on the locale with the corresponding ID. It is thus guaranteed that indexing a `Private`-distributed domain with `here.id` (the current locale's ID) will be local. In the above usage, the element type of the Private domains are Before the multiplication loop, the vector is stored to every index in `replVector`. There is a local result vector per locale, and they are summed after the multiplication loop to produce the final result. A `local` block wraps the loop body; this results in an assertion that no communication occurs within the block, as was the goal.

Due to further advice from Brad Chamberlain, the matrix is explicitly zippered with its domain, resulting in the domain leading (generating row ranges), and the domain and matrix following (producing co-ordinates and values). It may appear more natural to instead iterate over only the domain, and use the resulting indices to index the matrix within the loop body. In the future, both alternatives should result in the same behavior, but as explained by Brad Chamberlain, the compiler does not currently perform this optimization.

5 Testing

5.1 Benchmark Support

The SPMV benchmark was designed to aide in testing. A compile-time constant controls the matrix domain and distribution. This allows the same code to be compiled multiple times, producing multiple binaries from which to collect data. The provided Makefile produces binaries in the following configurations:

- Dense arithmetic domain, using a block distribution
- Sparse domain, using the (multicore) CSR distribution
- Sparse domain, using the default and currently serial sparse distribution (COO)

If the special `debug` make target is used, an additional binary is produced, which prints trace information for debugging the CSR distribution.

The benchmark prints certain configuration information (specifically, the number of tasks per locale used by the parallel iterators, and the number of locales), and collects timing data for the multiplication loop. With these facilities, data was collected for the CSR and dense block configurations.

5.2 Generation of Matrices and Vectors

A number of matrices and vectors were generated for testing. All generated matrices were square. A matrix was generated for each combination of dimension 2K, 4K, 6K, 8K, and 10K, and sparisty 70%, 80%, and 90%. A single vector was generated for each dimension. No special sparsity patterns were considered; non-zeroes should be distributed roughly uniformly. The code used to generate these input files is included in the submission.

5.3 Verification of Result Vectors

The threaded SPMV implementation from assignment 3 was used to generate correct answers for each matrix. The job submission script runs a verification script after each benchmark execution, and signals a failure if the correct / Chapel result vectors differ beyond some threshold.

5.4 Correctness

Although the multicore CSR distribution allows the correct answer to be generated, it is unlikely that it is yet correct. The benchmark requires a small subset of a fairly large API. It would be interesting to determine There are likely functions left unimplemented, that will cause compilation failures for other usages. At least some of these operations, such as slicing, have not yet been extended to sparse domains.

6 Results

Results are attached, and are also available as `randy/spmv_results.pdf` in the submission. No speedup greater than 1.0 was achieved for multi-core CSR. For all tested inputs, multi-core CSR took over 6 times longer when using sixteen or four tasks, versus when number of tasks was restricted to 1.

This would have been extremely disappointing, had the benchmark not also been tested with a dense, block-distributed matrix. The single-locale dense results are almost identical to the multi-core CSR results. This suggests that some issue exists either in Chapel or the particular Chapel configuration employed on Ranger to collect this data.

7 Acknowledgements

Many thanks are owed to Brad Chamberlain for patiently explaining the current distribution interface, and various performance pitfalls. Since the interface is not yet stable or documented, this would not have been possible without his help.

This work is based on the Chapel 1.1 distribution, released on April 15th, 2010. It is under a BSD license. The included language specification was a helpful reference.

The random vector generator used and included is copyright Benjamin Hiller, and is also under a BSD license.