C_bale

3.0.0

Generated by Doxygen 1.8.13

Contents

1 C_bale: the bale apps written in serial C serial

In one sentence

A textbook, C, implementation of the apps in bale.

The elevator pitch

The bale effort is, first and foremost, a vehicle for discussion for parallel programming productivity. We have included a simple C version of the apps in bale_classic as a concrete description of the apps written in a familiar language. Some of the apps (like histo and ig) are trivial as serial apps. It might be useful to view the serial version of the more complicated apps (like toposort and sssp) before dealing with the parallelism and buffer communication of the bale_classic apps. The C version of the way we implement a compressed row format data structure for a sparse matrix is also simpler than the parallel version. Finally, we also have examples of applications that are efficient as serial codes, but have no known parallel implementations.

This is a self contained directory that does not depend on build process for the rest of bale.

Nitty Gritty

Where does it run?

This is written in generic C, hopefully it will run in any C environment. It does depend on the `argp` library for command line argument parsing, doxygen for documentation and `pytest` for unit testing.

What is included here:

- Makefile - simple explicit makefile for all the apps.
- APPS:
  - histo – creates a large histogram (random stores) (see doxygen: histo.c)
  - ig – a large gather (random loads) (see doxygen: ig.c)
  - randperm – creates a random permutation (see doxygen: randperm.c)
  - transpose_matrix – computes the transpose of a sparse matrix (see doxygen: transpose_matrix.c)
  - permute_matrix – applies row and column permutations to a sparse matrix (see doxygen: permute_matrix.c)
  - triangle – counts the number triangles in a graph (see doxygen: triangle.c)
  - toposort – performs a toposort (matrix) sort of a morally upper triangular matrix (see doxygen: toposort.c)
  - sssp – solves the single source shortest path problem on a graph (see doxygen: sssp.c)
  - unionfind – uses the union-find data structure to find connected components in a graph (see doxygen: unionfind.c)
- Other:
- spmat_utils – the sparse matrix library and some support functions (see doxygen: spmat_utils.h, spmat_utils.c)
- std_options – the command line parsing routines (see doxygen: std_options.h, std_options.c)

Build Instructions

This is meant to be basic C, with a simple Makefile, so hopefully just typing 'make' will work.

```
make   # make the apps
make test # run the unit tests
make doc # make the doxygen documentation
```

However, we do use the argp library from the GNU standard library. This is usually present by default on most linux systems. On Mac, you will probably have to install it by hand (one way to do this is 'brew install argp-standalone') and then mess with your LD_LIBRARY_PATH and LDFLAGS variables.

Testing

For bale 3.0 we have started using pytest for the unit testing. One can run the test specified in file `tests/test_↩ all.py` by hand with the command:

```
pytest -s
```

For more details on how to modify the tests see pytest

Documentation

serial_C is documented using Doxygen. If doxygen is installed on your system, you should be able to type `make doc` and then load `./html/index.html` into a browser.

2 histo

Definition

We form the histogram of a large number of `int64_t`'s into a large table. The loop is as simple as:

```
foreach idx in index[ ]
  counts[idx]++
```

On a serial thread this shows the difference between random stores and streaming stores. On a large parallel machine this is the simplest case of managing the latency and bandwidth of the interconnection network.

Algorithms

We start by filling the index array with random number between 0 and the `table size`.

We have the generic algorithm (as simple as the loop above).

We also have a buffered version where we sort the indices into buffers, based on their high bits. When a buffer gets full we perform all the updates from that buffer's contents all at once. Studying this version with different number of buffers and buffer sizes might reveal properties of the memory hierarchy, like page sizes or tlbs.

A third version first sorts the indices before running the loop. This ought to be closer to the streams benchmark's performance as it is streaming with holes in it.

Discussion

Comparing these random access patterns to the streams benchmark for a particular node could be interesting.

In serial, we don't have the problem of atomic updates to `histo`. The parallel version in bale_classic as to use some form of atomicity to handle multiple threads concurrently update a particular entry in counts[ ].

Running this simple version of histo on a node, with node level threads, might reveal something about atomic updates to memory, without the performance being dominated by the interconnection network.

References

https://www.cs.virginia.edu/stream

3 ig

Definition

We do an index_gather of a large number of entries from a large table. The loop is simply:

```
for i in 0,...n-1
  target[i] = source[ index[i] ]
```

This is complement of histo.

Algorithms

We have the generic implementation and a buffered implementation.

In the buffered version, we collect the index[i] values into buffers based on their high bits. When a buffer is full we do all the gathers for the indices in that buffer before continuing.

Discussion

This is surprising complicated in the parallel case.

This exercises a streaming load of index, then random loads from table and a streaming store to tgt. As with the histogram example, playing with the number and sizes of buffers might reveal properties of a single thread memory hierarchy.

References

4 randperm

Definition

We fill an array of int64_t's with a flat random permutation.

Algorithms

#### the Fisher-Yates algorithm

```
fill the array, rand[ ], with indices 0 thru n-1.
for l=n-1, n-2, ... ,1
  swap rand[l] with a random entry in 0,1,...,l
```

By definition this picks "n balls from an urn without replacement". This is a standard serial algorithm that is in fact a serial algorithm. You have to process the entries from right to left one at a time.

the "dart throwing" algorithm

We pick a dart board (an array) that is bigger than the desired permutation, say twice as big and fill the entries with -1. Then randomly throw darts (numbers from 0 to `len-1`) at the dart board, re-throwing any dart that hits an entry that is already occupied (!= -1). Then we squeeze out the holes.

We picked the dartboard to be twice the size of the array so that even the last dart has a 50/50 chance of hitting an open entry.

the sorting algorithm

We form an array of (index, key) pairs. Then we randomly fill the keys and sort the array on the keys. Then we read the permutation from the indices.

NB. Repeated key are bad, but tolerated. They would be ok if ties were broken randomly or if doubles were real numbers.

Discussion

The Fisher-Yates algorithm must do one thing at a time, so it doesn't parallelize.

The dart throwing algorithm is here because it shadows the algorithm in bale_classic. It is in bale_classic because it is fun. And because the AGP version is essentially the same as this serial version.

The sorting version seems like a reasonable parallel algorithm, but it not in bale_classic because its not that interesting or fun.

References

`https://en.wikipedia.org/wiki/Fisher-Yates_shuffle`

5 permute_matrix

Definition

We apply given row and column permutations to a sparse matrix.

Algorithm

We produce a new sparse matrix data structure by copying the nonzeros and value entries of the original matrix to their new positions in the permuted matrix. To permute the rows, we compute the new offsets, based on the new order for the original rows. As we are copying the `nonzero[ ]` and `value[ ]` entries from the original matrix to their new position in the permuted matrix, we replace the nonzeros (the column indices) with the new column indices given by the column permutation.

Discussion

This app is really just a wrapper that calls the routine in the sparse matrix library.

This is C_bale to shadow the app in bale_classic. It is an app in bale_classic because the communication pattern is interesting.

References

6 transpose_matrix

Definition

Compute the transpose of a given sparse matrix.

Algorithm

We start by computing columns counts. These become row counts in the transpose. With these we can allocate the memory and set the row offsets for the transpose. Then we go through the `nonzero[ ]` and `value[ ]` arrays one row at a time. We write the given row and value to the location in the transpose matrix given by the of the nonzero (column number) of the original matrix

Discussion

This apps is a timer wrapper for the routine in the sparse matrix library.

This is C_bale to shadow the app in bale_classic. It is an app in bale_classic because the communication pattern is interesting.

References

7 toposort

Definition

We are given a matrix that is a random row and column permutation of an upper triangular matrix (with ones on the diagonal). Such a matrix has been called a morally upper triangular matrix. This algorithm finds a row and column permutation that, when applied, returns it to an upper triangular form.

Algorithms

We generate the morally upper triangular matrix by randomly permuting the rows and columns of a triangular matrix.

To find row and column permutations that would return the matrix to upper triangular form, we recursively find pivot position. A pivot is a nonzero (really a (row,col) pair) that is the single nonzero in a row. If we permute this row and column to the last row and column of our new matrix and delete the row and column from the original matrix, we can recursively construct the new matrix from the bottom right corner to the top left corner.

A more detailed description of the algorithm is given in bale_classic toposort documentation.

enqueuing pivots

In this version when the pivots are found they are placed in a queue. The algorithm runs until the queue is empty.

loop to find pivots

In the loop version we simply continue to loop over the rows until we have found all the pivots. This simplifies the flow of the algorithm but does redundant checking of rows which have already been processed.

Discussion

References

8 triangle

Definition

Find the number of triangles in a given simple unweighted graph.

A triangle is a set of three vertices {u,w,v} where edges {u,w}, {w,v} and {u,v} are in the graph.

Algorithm

This uses matrix algebra approach to counting triangles in a graph. Given, L, the (strictly) lower triangular matrix that holds the undirected graph, we compute {L .& (L $*$ L)} and {L .& (L $*$ U)}. Where U is the upper triangular matrix from the full adjacency matrix. Recall that U = L transpose.

Discussion

This is here to shadow the algorithms in bale_classic. The amount of work done in these two products depends on the matrix. It is important to us that communication pattern in the parallel app differs between these product (and still depends on the row densities in the matrix).

References

See the book, "Graph Algorithms in the Language of Linear Algebra", edited by Gilbert, and Kepner for more details on our approach to this problem.

9 sssp Single Source Shortest Path

Definition

We are given the adjacency matrix for a graph with non-negative edge weights *c(v,w)* and a given source vertex, *v_0*. We wish to find the lightest weighted path from *v_0* to all other vertices, where the weight of a path is the sum of the weights of the edges in the path. Note, if the graph is undirected we are given the full (symmetric) adjacency matrix.

Algorithms

We consider three algorithms: Dijsktra's, Delta-Stepping, and Bellman-Ford. Dijsktra's algorithm is not in bale_classic because it is a serial algorithm.

Delta-Stepping and Bellman-Ford are here as shadows of the parallel versions. The algorithms here are surprisingly similar to those in bale_classic. These may be slightly easier to read because we don't have the communication layer.

Discussion

The priority queue version of Dijsktra's algorithm is a favorite example of the use of data structures in irregular algorithms. We discuss this issue in the bale_classic app and in the serial unionfind app.

References

"Delta-stepping: a parallelizable shortest path algorithm" by U. Meyer and P. Sanders.

10 unionfind

Definition

Use the unionfind data structure to implement the union of disjoint sets approach to finding the connect components in a simple graph.

Algorithms

This algorithm relies on the notion of disjoint subsets. Given a collection of disjoint subsets that covers a space, the union of any of the members of the collection will result in another collection of disjoint subsets that covers the space. The algorithm starts with each vertex in its own subset. Then it looks at each edge in the graph and forms the union of the subsets that contain the incident vertices. The resulting subsets are the connected components.

The key to algorithm is a data structure that forms a tree for each subset. The union of the subsets is formed by connecting the root of one tree to the other tree.

We have implemented two versions: the first joins the trees by connecting the root of the tree corresponding to one vertex of the edge to the node corresponding to the other vertex of the connecting edge. The most efficient version connects the roots of the two trees according to the rank of the trees, where the rank is the length of the longest branch in the tree.

Discussion

This algorithm is in the C cousin of bale because we don't have a parallel version.

Like the use of the priority queue in Dijsktra's algorithm, this algorithm is a favorite example of how important data structures are. In these algorithms, the data structure is more than a place to storage the state. Manipulating the data structure *is* the computation.

In the most efficient version of the algorithm are manipulating structures that hold pointers that encode the tree and the rank of the tree. To do this in an AGP model, the whole operation must be done atomically. Doing this in a lock-free way is currently beyond our capability.

Unlike Dijsktra's or the Fisher-Yates algorithm, the use of this forest of trees is not necessarily serial. There is plenty of opportunity for asynchronous parallelism, but keeping the data structure consistent while making parallel changes to it seems overwhelming. There are other algorithms to find the connected components in a graph. We present this algorithm because we are interested in a discussion about parallel data structures.

References

11 spmat_utils

Definition

The definitions and a few routines sparse matrices we use in these apps.

Discussion

Compared to bale_classic this is a trivial library. This is not a general library; we only include routines need to support the apps in this directory.

We have combined some of the timing and convenience routines from the libgetput library in bale_classic into this file.

The main struct is the sparse matrix struct sparsmat_t

```
typedef struct sparsemat_t{
  int64_t numrows;       // the total number of rows in the matrix
  int64_t numcols;       // the nonzeros have values between 0 and numcols
  int64_t nnz;           // total number of nonzeros in the matrix
  int64_t *offset;       // the row offsets into the array of nonzeros
  int64_t *nonzero;      // the global array of column indices for nonzeros
  double *value;         // the global array of nonzero values (optional)
}sparsemat_t;
```

We also have a struct to hold an array of doubles. It is trivial in the serial case, but more useful in the parallel case. We include it here to make the codes look similar.

```
typedef struct d_array_t {
  int64_t num;             // the total number of entries in the array
  double * entry;          // the array of doubles
} d_array_t;
```

The routines involving sparse matrices are (check the doxygen documentation for details):

```
void             clear_matrix(sparsemat_t * mat);
int64_t          compare_matrix(sparsemat_t *lmat, sparsemat_t *rmat);
sparsemat_t *    copy_matrix(sparsemat_t *srcmat);

sparsemat_t *    init_matrix(int64_t numrows, int64_t numcols, int64_t nnz, int values);

sparsemat_t *    random_graph(int64_t n, graph_model model, edge_type edge_type, self_loops loops, double
       edge_density, int64_t seed);
sparsemat_t *    erdos_renyi_random_graph(int64_t n, double p, edge_type edge_type, self_loops loops,
       int64_t seed);
sparsemat_t *    erdos_renyi_random_graph_naive(int64_t n, double p, edge_type edge_type, self_loops loops,
       int64_t seed);
sparsemat_t *    geometric_random_graph(int64_t n, double r, edge_type edge_type, self_loops loops,
       uint64_t seed);
sparsemat_t *    generate_kronecker_graph_from_spec(int mode, int * spec, int num, int weighted);
int64_t          tri_count_kron_graph(int kron_mode, int * kron_spec, int kron_num);

sparsemat_t *    permute_matrix(sparsemat_t *A, int64_t *rperminv, int64_t *cperminv);
sparsemat_t *    transpose_matrix(sparsemat_t *A);

int64_t          tril(sparsemat_t * A, int64_t k);
int64_t          triu(sparsemat_t * A, int64_t k);

int64_t          is_upper_triangular(sparsemat_t *A, int64_t ondiag);
```

```
int64_t         is_lower_triangular(sparsemat_t *A, int64_t ondiag);

int64_t         is_perm(int64_t * perm, int64_t N);
int64_t *       rand_perm(int64_t N, int64_t seed);


sparsemat_t *   read_matrix_mm(char * name);
int64_t         sort_nonzeros( sparsemat_t *mat);
void            spmat_stats(sparsemat_t *mat);


sparsemat_t *   make_symmetric_from_lower(sparsemat_t * L);
int64_t         write_matrix_mm(sparsemat_t * A, char * name);
int64_t         dump_array(int64_t *A, int64_t len, int64_t maxdisp, char * name);
int64_t         dump_matrix(sparsemat_t * A, int64_t maxrows, char * name);
```

Routines to handle the arrays are:

- **init_d_array** allocate and initialize an array
- **read_d_array** read an array from a file
- **write_d_array** write an array to a file
- **set_d_array** set all entry equal a value
- **clear_d_array** free the space used by the array
- **copy_d_array** copy one array to another
- **replace_d_array** overwrite one array by another

The service rountines are:

- **rand_seed(int64_t seed)** seed the random number generator
- **rand_double()** return a flat random double in (0,1]
- **rand_int64(int64_t N)** return a flat int64_t in 0,1,...,N-1
- **wall_seconds()** the seconds since epoch as a double

  References

12 std_options

Definition

The use of the argp library (standard on most version of unix) to provide a unified command line option and parsing of the options.

Algorithms(s)

Discussion

References