Function:Troilkatt/DistributedSpell

From FunctionWiki

(Redirected from Function:Troilkatt/Parallel Spell)

This document describes the code organization of **parallel Spell.**

The main design is described in the Design document. The discussion page contains rejected design alternatives.

Contents

Background

See also: Spell paper [1] (http://bioinformatics.oxfordjournals.org/cgi/content/short/23/20/2692) and yeast search page [2] (http://imperio.princeton.edu:3000/yeast) .

Spell implements a context-sensitive search algorithm for gene expression compendium. A researcher using the system provides a small set of query genes to establish a biological search context. The query genes are sued to weight each dataset's relevance to the context, and within these weighted datasets additional genes are identified that are co-expressed with the query set.

Spell Algorithm

The dataset weight is calculated by:

$$w_d = \left(\frac{2}{|Q|(|Q|-1)} \right) \sum_{i=1}^{|Q|-1} \sum_{j=i+1}^{|Q|} f(z_{q_i,q_j})$$

And the per gene score by:

$$s_x = \frac{1}{|Q| \sum_{d \in D} w_d} \sum_{d \in D} \sum_{q \in Q} w_d f(z_{x,q})$$

Pseudo code to calculate the weight is:

```
// The datasetWeights array is used to return the weight for each dataset
// All elements are initialized to zero
datasetWeights[]
// The numWeighted counts the number of datasets that were weighted
// If the count is below a given number all weights are set to 1.0
numWeighted = 0
// Calculate weights
for (d = 0; d < datasets.length; d++):
    numPairs = 0
    for (i = 0; i < queryGenes.length; i++):
        for (j = i + 1; j < queryGenes.length; j++):
          // expressionValues[d][g] is an array of floating point values for gene
          // g in dataset d
          w = calculateScore(expressionValues[d][i], expressionValues[d][j])
          if (w != NaN):
              datasetWeights[d] += w
              numPairs += 1
    // The function returns the weights and the number of datasets that were weighted
    if ((numPairs > 0) && (weights[d] > 0)) {
        datasetWeights[d] = datasetWeights[d]/ numPairs
        numWeighted += 1
```

There are two filters used for the weight calculation:

1. If the number of query genes is 1, all weights are set to 1.0
2. If the number of weighted datasets (numWeighted) is less than 3, all weights are set to 1.0

After weighting the datasets, the weights are used to calculate the score for the genes:

```
// Gene scores is an array of scores for each of the g genes in the organism
```

```
geneScores[]
for (g = 0; g < allGenes.length; i++):
    // Score calculated for gene g
    score = 0
    // Sum of weights for datasets used to calculate score
    totalWeight = 0
    // Number of datasets used to calculate score
    dsetsUsed = 0
    for (d = 0; d < datasets.length: d++):
        if (weights[d] == 0): // Skip datasets that did not have any query genes
            continue

        // Calculate score for this dataset
        dsetScore = 0
        numPairs = 0
        for (q = 0; q < queryGenes.length; q++):
            s = calculateScore(expressionValues[d][g], expressionValues[d][q])
            if (s != NaN): // score could not be calculated
              dsetScore += s
              numPairs += 1

        if (numPairs > 0):
            score += weights[d] * (dsetScore / numPairs)
            totalWeight += weights[d]
            dsetsUsed += 1

    // The function returns genesScores and dsetsUsed
    if (totalWeight > 0):
        geneScores[g] = score / totalWeight
    else:
        geneScores[g] = 0
```

Before the datasets are returned the genes for which to few datasets were used to calculate the score are filtered out. The formula for the filter is:

```
if (dsetsUsed < min(numWeighted, max((0.1 * datasets.length), 4)))
    geneScores[g] = 0
```

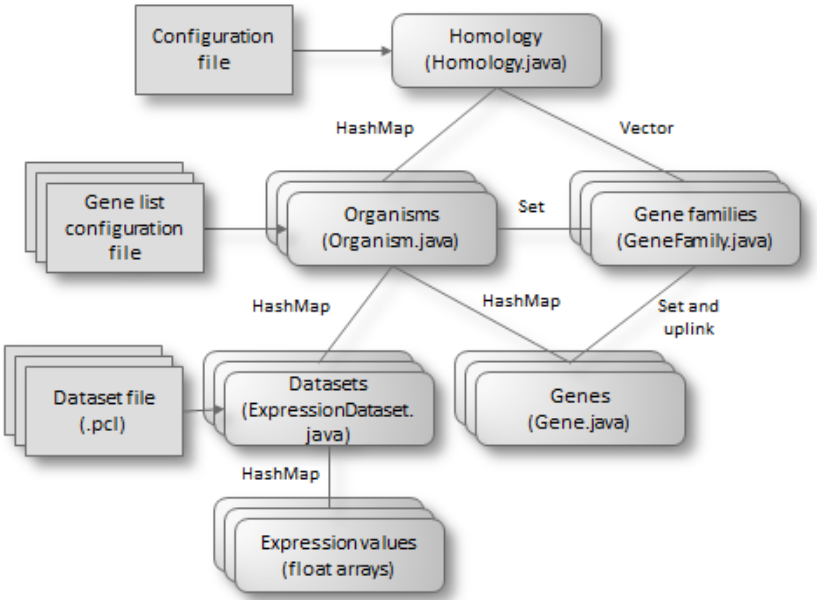
After the search the weights and datasets are sorted and the final calculation is done

```
// The datasetResults is an array of (dataset ID, contribution tuples)
// The contribution is the percentage of the sum of weights
datasetResults[]
// The geneResults is an array of (gene ID, score) tuples
// The score is the square root of the score calculated above
geneResults[]

dsetSum = sum(datasetWeights)
for (d = 0; d < datasets.length; d++):
    datasetResults.add(getDatasetID(d), (100 * datasetWeights[d]) / dsetSum)
// The array is sorted by weights, highest to lowest
datasetResults.sort()

for (g = 0; g < allGenes.length; g++):
    geneResults.add(getGeneID(g), sqrt(geneScores[g]))
// The array is sorted by scores, highest to lowest
geneResults.sort();
```

Sequential Code Overview

The main class in the sequential code is SearchServer.java that implements a server that listens on a specified port and for each connections creates a SearchThread.java object that handles the query (despite the name, it is not a thread). The SearchThread parses the query and calls the querySearch() function in the Searcher.java class. This function implements the search functions described in the previous function.

The main data structures are the expression values. These are organized as shown in the figure:



The data structure organization is designed to support Homologies where a gene can belong to multiple organisms. The main three datastructures are therefore:

- Organism: which contains a list of genes and a set of dataset.
- Gene familiy: which is a unique identifier for a gene. A gene can be in multiple organisms, but for our experiments we only consider genes that belong to only one organism.
- Expression datasets: which contain the expression values for an experiment.

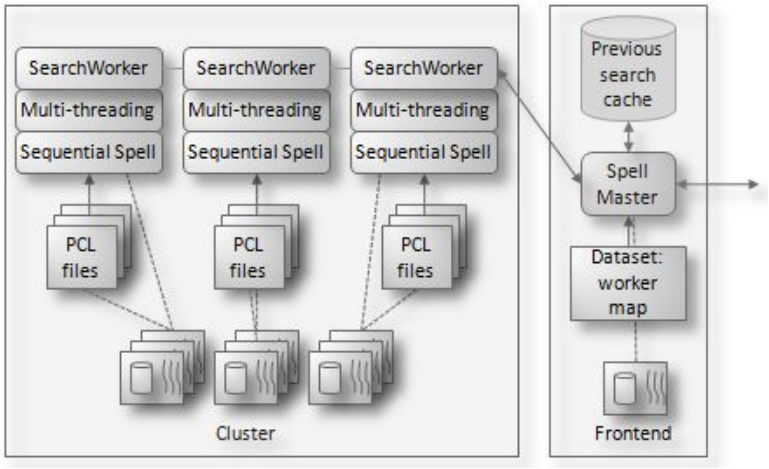
The Homology object is used to map the received dataset IDs to ExpressionDataset objects (by iterating over the organisms), map the received query gene IDs to GeneFamily objects (by iterating over the organisms), and to receive a list of all GeneFamily objects. During the dataset weight and gene score calculating, the GeneFamily objects are used to get the unique gene ID which is used to lookup the expression values for a gene in a dataset.

The code is split among the following files:

- ExpressionDataset: expression dataset data structures and distance metric functions.
- Gene: gene data structure.
- GeneFamily: gene familiy data structure and functions.
- Homology: homology data structure and functions.
- Organism: organism data structure and functions
- PartialScore: comparable (int key, float value) tuples.
- Searcher: the dataset weight calculation and gene score calculation functions.
- SearchResult: result returned by the searcher.
- SearchServer: the main class.
- SearchThread: object called for each new request (not a thread despite the name).

Architecture

The parallel spell is designed to utilize the multiple cores and multiple nodes in a cluster while maintaining backward compatibility with the sequential search algorithm.



Parallel Spell consist of two main parts:

- A SpellMaster that runs on a frontend machine. It receives search queries from clients such as the webserver, distributes the search to workers, orchestrates the result gathering from workers, and sends the final result to the requesting client. The SpellMaster maintains two important data structures. First, a map of the dataset to worker distribution. Second, it caches the results of previous searches. This allows reducing the work for *refined* queries as described below.
- SearchWorker's that runs on the cluster nodes. These receives a subset of a query from the SpellMaster, computes a partial result, and sends it to the SpellMaster. A search worker is multi-threaded, and they use the Sequential Spell algorithm for the search.

Dataset storage

The dataset are persistently stored in either a global filesystem such as NFS, or in HDFS.

A worker reads maintains all of it's datasets in memory. It loads these into memory at startup time.

Parallel Implementation

The parallel search algorithm splits the query processing among mulitple workers by assigning a subset of the query datasets to each worker. The workers then compute the dataset weights for its datasets, and all gene scores for its datasets. Finally, the master gathers the calculated weights, and reduces the gene scores.

The algorithm can be implemented using the MapReduce paradigm:

```
map(datasetID, expressionValues):
    queryGenes[] = getQueryGenes()
    // Dataset weight calculation pseudo code is given in the background section.
    datasetWeight = calculateWeights(queryGenes)
    // Pseudo code for parallel gene score calculation is given below
    geneScores[], geneWeights[], geneDatasetsUsed[] = calculateGeneScore(queryGenes, datasetID)
    // Output dataset weight...
    output("dataset", (datasetID, datasetWeight))
    // ...and all gene scores
    for (g = 0; g < allGenes.length, g++):
        output("gene: " + allGenes[g], (geneScores[g], geneWeights[g], geneDatasetsUsed[g]))
```

```
reduce(key, values):
    // The datasetResults is an array of (dataset ID, contribution tuples)
    // The contribution is the percentage of the sum of weights
    datasetResults[];
    // The geneResults is an array of (gene ID, score) tuples
    // The score is the square root of the score calculated above
    geneResults[];
    // There are two types of keys
    if (isDataset(key)):
        for (w in values):
            datasetID, datasetWeight = w
            // Just output received values
            output("dataset: " + datasetID, w)

    else: // is a gene
        geneID = getGeneID(key)
```

```
        for (w in values):
            geneScore, geneWeight, geneDatasetsUsed = w
            // The genes are reduced by calling a function that adds the geneScore, geneWeight, and geneDatasetsUsed to the prev
            geneResults.update(geneID, geneScore, geneWeight, geneDatasetsUsed)
        // output reduced score
        output("gene: " + geneID, geneResult.getFinalScore());
```

Computation time and communication volume

The computation time for the weight calculation is: $O(D * Q^2/2 * E)$, where D is the number of datasets, $Q^2/2$ is the query-gene-to-query gene calculations, and E is the average number of expressions in the datsets. The number of floating point operations in the last part is about: 5*E + 16 (hence the number of floating point operations is: $G * Q^2/2 * (5E + 16)$)

The computation time for the score calculation is: O(G * D * Q * E), where G is the number of genes in the organism (or homology), D is the number of datasets, Q is the number of query genes, E is the number of floating point operations and the number of floating point operation is the same as above. Since the score calculation depends on the dataset weights the calculations must be done in sequence.

The communication volume for sending the query requests to the workers is: (D * 4) + (N * G * 4) + (Q * 4), where D is the number of datasets, N is the number of nodes allocated to process the query, G is the number of genes in the organism/homology, and 4 is the size of the integers or floating point values used in the messages. An optimization allows using only a single 4 byte values instead of (N * G * 4)

The communication volume for the reduce part is: (D * 4) + (N * G * 12), where the constants are as above, and 12 is the bytes used for the gene score, gene weight, and datasets used.

Data distribution

At startup time the master distributes the datasets among the workers using a greedy algorithm that takes as input the memory allocated for the expression values at each worker. The master then assigns datasets linearly until the memory at each worker is full. When all datasets have been assigned, replicas are assigned to the following workers. The master estimates the memory usage for each dataset by multiplying the on disk size with 2. Pseudo code for the algorithm is:

```
// Get the size of the allocated memory at each worker
maxRss = getWorkerMemorySize()
// Index in the datasets array
ci = 0
for (i = 0; i < aliveWorkers.length; i++):
    // Size of datasets added to worker
    sum = 0;
    // Number of datasets assigned to the worker
    cnt = 0
    while ((cnt < datasets.length) && (sum < maxRss)):
        assignDataset(worker[i], dataset[ci])
        sum += estimateDatasetSize(dataset[ci])
        cnt += 1
        ci = (ci + 1) % datasets.length
```

The algorithm will distribute datasets with replication k, if (maxRss * aliveWorkers.length) / data.size >= k. It is an error if some dataset is not assigned to at least one worker.

Query distribution

To split the workload for a query the master distributes the datasets to be computed by each worker. The algorithm receives the number of workers (QW) to distribute the data on and then attempts to split the datasets equally among the workers. If the QW selected workers do not have all datasets the remaining will be distributed on the other workers. Pseudo code for the algorithm follows.

```
// The datasets are split equally among the workers
datasetsPerWorker = queryDatasets.length / nQueryWorkers
// This ensures that all datasets are distributed
if (datasets.length % nQueryWorkers > 0):
    datasetsPerWorker += 1

// Assigned datasets is an array with a list of datasets assigned to each worker
assignedDatasets[]
// Datasets to assign is a list with the query datasets to assign
datasetsToAssign[] = getQueryDatasets()
```

```
// Attempt to assign QW datasets to each worker
for (i = 0; i < QW; i++):
    while (datasetsToAssign.length > 0) && (assignedDatasets[w].length < datasetsPerWorker):
        dset = datasetsToAssign.removeFirst()
        // The workerDatasets contains the set of datasets assigned to a worker
        if (workerDatasets[i].contains(dset)):
            assignedDatasets.add(dset)
        else:
            datasetsToAssign.addToFront(dset)
```

At this this stage each worker has a maxium of OW *datasetsPerWorker* datasets. However some datasets may not have been assigned. The remaining datasets are distributed among the remaining workers to ensure that the initial OW workers can finish their processing within the allocated time.

```
while (datasetsToAssign.length > 0):
  for (i = QW; i < workers.length; i++):
      dset = datasetsToAssign.removeFirst()
      if (workerDatasets[i].contains(dset)):
          assignedDatasets.add(dset)
      else:
          datasetsToAssign.addToFront(dset);
```

It is possible that some datasets were only in one of the OW initial workers that already got the maximum number of datasets in the first round of distribution. This final stage distributes the remaining datasets among all workers. It will add as many datasets as possible to a worker, independent of the number of datasets already assigned.

```
while (datasetsToAssign.length > 0):
    for (i = 0; i < QW; i++):
      dset = datasetsToAssign.removeFirst()
      if (workerDatasets[i].contains(dset)):
          assignedDatasets.add(dset)
      else:
          datasetsToAssign.addToFront(dset);
```

Thread level parallelism

The parallel algorithm has a high communication volume. It is therefore important to exploit all available thread level parallelism before distributing the workload among nodes. We use the same strategy for intra-node parallelism as for inter-node parallelism. That is, after mapping a subset of the datasets to a node these are further mapped to helper threads that compute the scores and weights for these datasets. Then the reduce is also done in two phases; first intra-node and then inter-node. Pseudo code for the SearchWorker thread that is started for a new connection follows.

First the datasets are divided into P parts using an algorithm that returns a distribution where the first split has the 1..S first datasets, the second has the S+1...2S splits, and so on. Typically P is the number of cores in the machine.

```
datasetSplits[] = distributeWork(datasets)
```

The master starts P-1 helper threads and does the calculation for the last split itself. This split will be smaller than the first split if D mod P > 1. The master is therefore typically among the first threads to complete the computation, thereby allowing for the data reduce time to be overlapped with the computation time for the slower threads.

```
// Start P-1 helper threads
for (h = 0: h < P-1; h++):
    // Helper threads does the same calculation as the master thread
    helperThreads[h] = startHelperThread(datasetSplits[h])
// The master thread does the computation for the last, and usually smallest, split
datasetWeights = calculateDatasetWeights(datasetSplit[P-1])
geneScores, geneWeights, geneDatastsUsed = calculateGeneScoreParallel(datasetWeights)
for (h = 0: h < P-1; h++):
    // Wait for the helper threads...
    helperThreads[h].wait()
    helperDatasetWeights, helperGeneScores, helperGeneWeights, helperGeneDatastsUsed = helperThreads[h].getResults()
    // ...update dataset weights
    datasetWeights.append(helperDatasetWeights)
    // ...and update the sums for calculated gene values
    geneScores.update(helperGeneScores)
    geneWeights.update(helperGeneWeights)
    geneDatasetsUsed.update(helperGeneDatasetsUsed)
```

```
// The calculated dataset weights and gene values are sent to the Frontend for global reduction
return (datasetWeights, gneScores, geneWeights, geneDatastsUsed)
```

To calculate the dataset weights the sequential function is used unmodified. But, the parallel gene score calculation requires some modifications, since the gene score, weight and datasets used are sent to the frontend.

```
// Array with scores calculated for each of the G threads ordered by gene ID.
geneScores[]
// Array with the sum of weights for datasets used to calculate the gene score
geneWeights[]
// Array with the number of datasets used to calculate score
geneDatasetsUsed[]

for (g = 0; g < allGenes.length; i++):
    for (d = 0; d < datasets.length: d++):
      // ** This part is unchanged **
      if (weights[d] == 0):
        continue
      dsetScore = 0;
      numPairs = 0;
      for (q = 0; q < queryGenes.length; q++):
          s = calculateScore(expressionValues[d][g], expressionValues[d][q])
          if (s != NaN):
              dsetScore += s
              numPairs += 1

      // Instead of local variables the results are stored in the arrays returned by the function
      if (numPairs > 0)
          geneScore[g] += weights[d] * (dsetScore / numPairs)
          geneWeights[g] += weights[d]
          geneDatasetsUsed[g] += 1

  // The geneScore is divided by the totalWeight after the final reduction
  // Gene filtering is also done after the final reduction

  // The gene arrays are returned for redcution
  return geneScores, geneWeights, geneDatasetsUsed
```

Distributed computation

The SpellMaster creates a new "scheduler" thread for each request it received. The scheduler thread allocates the number of nodes (K) needed to complete the query within the required time. K may be smaller than the total number of nodes (N). It then splits the data among these K nodes, send a partial query request to each worker, and finally reduces the values returned from the workers. Pseudo code follows.

```
// Split the datasets among K workers using the algorithm in the Query distribution section.
datasetSplits = distributeDatasets(K)
// Send dataset and query gene information to each worker
for (k = 0; k < K; k++):
    sendSplit(worker[k], datasetSplits[k])
```

The reduce is implemented by the master using a select call on all the worker sockets. This call returns once data is returned on a socket, thereby allowing the data to be processed in the order the results arrives from the workers. It also allows hiding the communication and result reduction time in load imbalance, since the master can receive and reduce the data while waiting for the slow node(s) to complete.

```
for (k = 0; k < K; k++) {
    r = selectOnAllSockets();
    datasetWeights, geneScores, geneWeights, geneDatasetsUsed = workers[r].recvResults()
```

For the parallel code the number of weighted datasets is not known before reeving results from all workers. If there are too few datasets the query must be re-run with a parameter specifying that datasets should not be weighted. If enough datasets were weighted or the query was re-run, the weights are calculated as described for the sequential code. The gene score result calculation is changed since the sum, and number, of dataset weights used to calculate the score is not known before the reduce.

```
// The dataset result structure is created using the same function as for the sequential code
datasetResult = getDatasetResults(datasets, datasetWeights)

// The gene results differs from the sequential version
for (g = 0; g < allGenes.length; g++):
    if (geneFilter()):
```

```
        geneResults.add(getGeneID(g), 0.0);
    else:
        geneResults.add(getGeneID(g), sqrt(geneScores[g] / geneWeights[g]));
geneResults.sort();
```

Fault-tolerance

The current code implements a simple fault-tolerance scheme: if the results is not received within N seconds the search is assumed to have failed and is re-executed.

A crashed SpellMaster can recover it's state from the workers.

A new or restarted SearchWorker cannot be added to the workers list.

Dataset redistribution

Dataset redistribution is not currently implemented.

Refined search

Refined search is not implemented.

Query result caching

All query results are cached at the SpellMaster. The cache is used to detect duplicates, but since refined search is not implemented it is of limited use.

Parameter setup

There are three key parameters that influence the performance and resource usage of Distributed Spell:

- The Java heap space on each worker (JVM argument for SearchWorker).
- The memory allocated at worker, which also implicitly set the replication factor (SpellMaster command line argument -m).
- The number of workers used for a query (SpellMaster command line argument -w).
- The number of threads per worker started for a query (SearchWorker command line argument).

Recommended setting for these values are:

- The Java heap space should be set such that there is abotu 10-20% free space after loading all the workers datasets to memory, since there is a significant performance degradation when the heap becomes full.
- The memory allocated at each worker should be such that the number of replicas is at least 2. This allows re-running a query for fault-tolerance. TODO: performance experiments.
- The number of workers per query should be set to ??? TODO: performance experiments,
- The number of threads should be equal to the number of processors on the node. TODO: performance experiments.

Known limitation and issues

Known limitations and issues:

- The query workload distribution does not take into account the workload of each datasets. This will cause a severe load imbalance for the Human datasets.
- The number of workers allocated to a query is set statically.
- The fault-tolerance scheme only detects a fault but does not handle it.

DistSpell Protocol

The DistSpell protocol implements a public interface between SpellMaster and clients, and an internal interface between SpellMaster and workers.

Connections

Clients open a new TCP/IP connection to SpellMaster for each request. On the SpellMaster each request is handled by a separate thread.

There are no handshake messages.

For each query SpellMaster creates a new TCP/IP connection to the workers that computes a partial result. A worker forks off a new thread for each received request. There are no handshake messages.

State

The protocol is designed to work even if the SpellMaster does not need to maintain any state for the clients. However, to optimise refined search queries the server caches search results for a limited amount of time. But a refined search will give the correct result even if the previous result has been evicted from the cache.

SpellMaster must maintain state about which SearchWorker's are running, and which datasets each worker is handling. In case of a crash the SpellMaster can rebuild this state by requesting a list of loaded datasets from each worker.

All SearchWorkers are stateless.

New Client-SpellMaster Protocol

The binary protocol between the Web-server (or another client) and the SpellMaster is as follows:

Binary request messages

Message	Op-code	Arguments	Data types
Binary search	**4**	**nDatasets, [dataset ID's], nQueryGenes. [query gene ID's], orgID.length, orgID, maxDsets, maxGenes**	**int, [int], int, [int], int, byte[], int, int**
Binary refined search	**5**	**nDatasets, [dataset ID's], nQueryGenes. [query gene ID's], orgID.length, orgID, maxDsets, maxGenes, query-ID**	**int, [int], int, [int], int, byte[], int, int, int**
MySpell search	**6**	**nDatasets, [dataset ID's], nQueryGenes. [query gene ID's], orgID.length, orgID**	**int, [int], int, [int], int, byte[]**
Dataset list	**7**	**orgID.length, orgID**	**int, byte[]**
Heartbeat	**9**		

All requests are sent as *binary* messages. The arguments for the *binary search* request message are: the number of datasets, dataset ID's, number of query genes, query gene ID's, organism ID length, organism ID, maximum number of datasets weights to return, and maximum number of gene scores to return. All except *orgID* are Java integers, *orgID* is a string encoded in US-ASCII. *Binary refined search* has in addition a query-ID (integer) returned in a previous search.

Note that to reduce the number of bytes sent over the network, it is possible to set nDatasets=1, and dsetID[0]=-1. The SpellMaster will then insert a list of all dataset IDs to the message before it is processed. Also, the maximum number of scores and weights to return can both be set to -1 to return all scores and/or weights.

Binary result message

Request	Op-code	Response	Data types
Search	**104**	**nWeights, numWeighted, [(dataset ID, weight),], nScores, [(gene ID, score),], query-ID**	**int, int, [int, float,...], int, [int, float...], int**
Refined search	**104**	**nWeights, numWeighted, [(dataset ID, weight),], nScores, [(gene ID, score),], query-ID**	**int, int, [int, float,...], int, [int, float...], int**
MySpell search	**101**	**nWeights, [weigths], nScores, [(score, gene weight, dataset count),]**	**int, [float], int, [float, float, int...]**
Dataset list	**107**	**nDsets, [dataset ID's]**	**int, [int]**
Heartbeat ACK	**109**		
Error	**100**	**errorMsg.length, errorMsg**	**int, byte[]**

For the binary *search* and *refined search* messages the return messages consists of: the number of datasets for which a weight was calculated, the number of datasets weights returned, dataset ID and dataset score tuples, number of scores returned, gene ID and gene score tuples, and a session ID. The dataset weight tuples and gene score tuples are sorted in descending order with respect to the weight/score. The length of these lists depends on the maximum weights and maximum scores arguments send in the request.

The *MySpell* response returns the weight and score for all datasets and all genes. The format is: the number of datasets weights returned, the dataset weights (only), number of genes scores returned, and (gene score, gene weight, dataset count) tuples for all genes. The dataset count is the number of datasets used to calculate the gene score. The scores and weight are floating point values, while all other values are

integers. Note that neither the dataset ID's nor the gene ID's are returned. Instead both lists are sorted in ascending order with respect to respectively the dataset ID's and gene ID's.

The *dataset list* message comprise the nubmer of dataset ID's returned followed by a list of datasets.

SpellMaster-SearchWorker Protocol

The internal interface is defined by the messages sent between SpellMaster and the workers. Most methods that both SpellMaster and SearchWorker used for sending and receiving messages are in DistributedSpell.java.

SpellMaster to SearchWorker mesasges

Message	Op-code	Arguments	Types
Partial search	**1**	**nDatasets, [dataset ID's], nQueryGenes, [query gene ID]**	**int, [int], int, [int]**
Load datasets	**3**	**nDatasets, [dataset ID]**	**int, [int]**
Get Datasets	**2**		
Heartbeat	**9**		
Get stats	**10**		
Exit	**0**		

All messages starts with an integer opcode that defines the message type and content.

The *partial search* request specifies the partial query handled by the worker receiving the message. The *load dataset* message specifies a list of datasets that the worker should read from a filesystem and load into memory. The number of elements is specified by a preceding integer, then each element in the list consist of an organism ID, datasetID, and filename. The *get datasets* message is used by the SpellMaster to get a list of datasets handled by each worker. It can be used if the SpellMaster is restarted for example due to a crash. Finally, the *exit* message shuts down a worker.

The *partial search* message comprise the dataset ID's to search and the query gene ID's (all encoded as integers). The *load dataset* message has a list of datasets encoded as an US-ASCII String that specifies the organism, dataset ID, and filename for each dataset to load. The list is terminated by an empty string. The filename can either be a on a local, NFS, or HDFS filesystem (using the "hdfs://" prefix). The *get datasets*, *heartbeat*, *get stats*, and *exit* messages do not have any arguments.

SearchWorker to SpellMaster messages

Request	Opcode	Response	Types
Partial search	**101**	**nWeights, [weigths], nScores, [score, gene weight, dataset count]**	**int, [float], int, [float, float, int]**
Load datasets	**103**		
Get datasets	**102**	**nOrganisms, [orgID.length, orgID, nDatasets, [dataset ID]]**	**int, [int, byte[], int, [int]]**
Heartbeat	**109**		
Error	**100**	**errorMsg.length, errorMsg**	**int, byte[]**
Get stats	**110**	**freeMemory, totalMemory, workerThreads, CPUs**	**long, long, int, int**
Exit	**<no reply>**		

The return messages start with an opcode (integer) that defines the message format.

The format of the *partial search* return message is: the number of datasets weights returned, the dataset weights (only), number of genes scores returned, and (gene score, gene weight, datasets used) tuples for all genes. The datasets used is the number of datasets used to calculate the gene score. The scores and weight are floating point values, while all other values are integers. Note that neither the dataset ID's nor the gene ID's are returned. Instead both lists are sorted in ascending order with respect to respectively the dataset ID's and gene ID's.

Get datasets returns the number of organisms handled by a worker, and then for each organism the organism ID, number of datasets, and dataset ID's . The dataset IDs are sorted with smallest ID first. The worker sends an acknowledgement message when the datasets in *load datasets* are loaded. For the *exit* message the socket is closed.

The statistics returned by *get stats* are: the free memory in the JVM, the total memory in the JVM, number of worker threads used for a query, and the number of processors available for the JVM. All JVM statistics are from the Runtime class.

In case of an error, the 'error' message is sent with an error message encoded as a byte array preceded by a four byte integer specifying the length in bytes.

Old Client-SpellMaster Protocol

DistributedSpell is backward compatible with the old ASCII protocol between the Web-server (or another client). The *query* and *refined query* messages allows sending queries in a human readable format. The *more genes* and *more datasets* request are used by clients that cannot receive all data in one response (such as the Ruby-on-rail webserver). Each message is sent as a tab delimited string with a newline at the end. In the query messages the gene ID's and dataset ID's are comma separated. Query-ID is an integer returned for each query. Note that the query-ID may be changed even if doing a refined search. The fields are as follows:

Request messages

Message	gene-ID's	Datasets	Op-code	Arguments
Old query	gene1,gene2,...,geneQ	dataset1,dataset2,...,datasetD		
Query	gene1,gene2,...,geneQ	dataset1,dataset2,...,datasetD	query	
Refined query	gene1,gene2,...,geneQ	dataset1,dataset2,...,datasetD	refined	query-ID
More genes			more genes	query-ID,from-gene-rank,to-gene-rank
More datasets			more datasets	query-ID,from-dataset-rank,to-dataset-rank
Binary protocol			binary	

Non-binary sesponse messages (all as Strings)

Message	gene-ID's	Datasets	Arguments
Old query	gene1:score1,gene2:score2,...,geneG:scoreG	dataset1:weight1,dataset2:weight2,...,datasetD:weightD	
Query	gene1:score1,gene2:score2,...,geneN:scoreN	dataset1:weight1,dataset2:weight2,...,datasetM:weightM	query-ID
Refined query	gene1:score1,gene2:score2,...,geneN:scoreN	dataset1:weight1,dataset2:weight2,...,datasetM:weightM	query-ID
More genes	geneF:scoreF,geneF+1:scoreF+1,...,geneT:scoreT		
More datasets		datasetF:weightF,datasetF+1:weightF+1,...,datasetT:weightT	

The data is sent as a newline-delimited string. N and M are implementation dependent and specify the maximum number of genes and datasets returned at a time. For all return messages the genes and datasets are sorted highest-values first.

Programming Interface

See also Javadoc for API specification: [link]

DistributedSpell is in one package, but the files can be split into five parts: Spell master, search workers, distributed Spell protocols, sequential Spell, and utility functions.

SpellMaster

The SpellMaster does the following:

1. Receive search requests, and send responses, from clients including the Spell webserver
2. Distribute datasets among the worker nodes
3. Distribute query workload among the workers, and send queries for execution.
4. Detect failed queries and re-execute these (not yet implemented)
5. Detect failed workers and re-distribute their datasets (not yet implemented)
6. Receive new datasets to be added to the compendia and distributed among the workers (not yet implemented)
7. Cache results from previously executed queries to provide fast "refined searches" (not yet implemented)

The code is split among the following files:

- DatasetResult: comparable class that allows to store a (int key, float value) tuple and sort these based on the value. It is used to sort the final dataset weights, and gene scores.
- DistSpellProtocolRequestHolder: class used to hold the arguments for a search request.
- QuerySplit: class to hold the dataset distribution and other meta-data for a query.
- SchedulerThread: thread forked of for each request received from a client. This thread implements most of the SpellMaster functionality.
- SearchException: search exception thrown for search errors such as invalid dataset or gene IDs.
- SearchResult2: simplified search result holder.
- SearchSession: data structure used to cache search parameters and results.
- SpellMaster: main class that implements the server, and does the initial data distribution.

- SpellMasterBC: backwards compatible SpellMaster that allows clients to use the old ASCII protocol.
- SpellMasterTroilkatt: SpellMaster that reads datasets from HDFS.

In addition the SpellMaster uses the sequential code classes to do the search and to implement data structures.

SearchWorker

The search worker does the following:

1. Receive list of datasets from master, and load these into memory/local filesystem
2. Do the Spell search on it's datasets

The code is split among the following classes:

- PartialSearchResult: holder for the result returned by each worker.
- SearchWorker: main class.
- SearchWorkerThread: thread forked of for each new request. It implements the SearchWorker functionality.
- SearchWorkerTroilkatt: SerchWorker that reads datasets from HDFS.

DistSpell protocol

Functions for sending and receiving the messages described in the DistSpell protocol are in the DistSpellProtocol class.

Utility

Utility functions and tools are implemented by the following classes:

- ConfigFileParser: contains functions to parse the different configuration files.
- GetWorkerStats: client program that queries workers and returns statistics about loaded datasets, memory usage, and CPU usage.
- WorkerStats: various usage statistics for a worker.

Deployment

This section describes how to compile, configure and run the distributed Spell code.

Distributed Spell consists of a set of workers typically run on a cluster and a master run on a front-end machine.

Quick Setup

To quickly setup and run Distributed Spell: In summary the steps to download, compile, and run SpellWeb2 are:

1. Download and install Eclipse Java IDE.
2. Download the DistributedSPELL module from the CVS server at cvs.cs.princeton.edu.
3. Create a new Java project in Eclipse with the downloaded code.
4. Include the gnu-getopt.jar library in the build path.
5. Compile the code (done automatically by Eclipse).
6. Specify the configuration files for the compendia to use, or reuse existing files (DistributedSPELL/data has samples for yeast).
7. Create a hostfile with the hostnames and ports on which the workers should be run.
8. Configure DistributedSpell by modifying scripts/configDistributedSpell.py
9. Start the workers and master by executing the scripts/startDistributedSpell.py file.

Source Code and IDE Setup

To develop the code we recommend using the Eclipse Integrated Development Environment (IDE) or NetBeans. DistributedSpell has been developed in Linux and Windows 7, and the code runs in both operating systems. Altough, all scripts, unit and integration tests are for Linux.

The code is in the spell repository on the server cvs.cs.princeton.edu. Currently Lars Ailo (larsab@cs.uit.no), Wenli (wenliz@princeton.edu) and Qian (qzhu@Princeton.EDU) all have administrator rights for the repository and can therefore grant access to other users. To download the code the IDE or a CVS client can be used. In Eclipse the code is downloaded by:

1. Create a new account at "cvs.cs.princeton.edu" and join the spell repository.
2. Start the New Project Wizard.

3. Select "Projects from CVS".
4. Create a new repository location, using host "cvs.cs.princeton.edu", repository path "/cvs", and connection type "extssh".
5. Select to use an existing module, such that you can browse the repository.
6. Select the DistributedSPELL module.
7. Select "Check out project as a project configured New Project Wizard".
8. Select HEAD.
9. Select Java Project.

Build Path Configuration and Required Libraries

The Java port of GNU getopt (gnu-getopt.jar) must be included in the build path. This file is typically included with most Linux distributions.

In addition to compile with hadoop HDFS support gnu-getopt.jar, hadoop-<version-number>-core.jar, hadoop-<version-number>-tools.jar must also be added to the build path. But note that DistributedSPELL can compiled and used without hadoop by excluding SpellMasterTroilkatt.java and SearchWorkerTroilkatt.java from the build path.

The server side code should run on all JVM versions 1.6 or later. We have mainly tested SpellWeb2 on Linux but the code should be portable to other platforms (except the startup scripts).

Compiling

Eclipse compiles the code automatically and outputs the class files to DistributedSPELL/bin.

Configuration Files

To configure SpellMaster the following files must be created:

- A configuration file where each line contain the following: the organism ID, <tab>, gene list filename, <tab> dataset list filename, <tab>, dataset directory.
- A gene list file where each line has a geneID<tab>systematic-name mapping.
- A dataset list file where each lines has a datasetID<tab>filename mapping. The files can either be in a NFS directory or in HDFS. If on a local filesystem, only the base filename should be used. Otherwise, if HDFS is used the filenames should be "hdfs:/<absolute path>/filename"
- A hostfile, with one hostname<tab>port<newline> per worker that should be started.

Startup Scripts

DistributedSpell can be started using the startDistributedSpell.py script and stopped using the stopDistributedSpell.py script. Before using these scripts the variables in configDistributedSpell.py must be set.

Command Line Startup

To start DistributedSpell run the following command, in the SpellMaster/bin directory:

java <vm arguments> -classpath .:<libraries> <main class> <arguments>

The *main class* is either edu.princeton.function.distspell.SpellMaster, or edu.princeton.function.distspell.SpellMasterTroilkatt. The latter should be used if the dataset files are in HDFS.

It is usually necessary to specify the heap size in *vm arguments*, for example -Xmx4096m to use 4GB of virtual memory for SpellMaster.

The *libraries' in the classpath are: gnu-getopt.jar for SpellMaster. For SpellMasterTroilkatt these are hadoop-<version-number>-core.jar, hadoop-<version-number>-tools.jar, gnu-getopt.jar, and cs-logging.jar.*

For a list of *arguments* run SpellMaster[Troilkatt] with the -h argument.

Examples

To start DistributedSpell with dataset files on a local filesystem, run: java -Xmx2048m -classpath .:<java-lib>/gnu-getopt.jar edu/princeton/function/distspell/SpellMaster -c <configuration file> -p 7008 -n <hostfile>

To start DistributedSpell with dataset files in HDFS, run: java -Xmx2048m -classpath .:<hadoop lib>/hadoop-0.20.0-core.jar:<hadoop

lib>/hadoop-0.20.0-tools.jar:<haddop conf>:<java lib>/gnu-getopt.jar:<java lib>/cs-logging.jar edu/princeton/function/distspell/SpellMasterTroilkatt -c <HDFS configuration file> -p 9008 -n <hostfile>

Both will start SpellMaster, which in turn starts SearchWorkers on each worker node (and port) specified in the hostfile.

Recovery

If SpellMaster crashes it can be restarted without restarting SearchWorkers by using the "-r" argument: java -Xmx2048m -classpath .:<java-lib>/gnu-getopt.jar edu/princeton/function/distspell/SpellMaster -c <configuration file> -p 7008 -n <hostfile> -r

Debugging

To manually start a SearchWorker it can be run as follows:

java <vm arguments> -classpath .:<libraries> <main class> <configuration file> <port>

The *main class* is either edu.princeton.function.distspell.SearchWorker, or edu.princeton.function.distspell.SearchWorkerTroilkatt. The latter should be used if the dataset files are in HDFS. *vm arguments*, *libraries*, and the *configuration file* are the same as for SpellMaster. *port* should match a port in the hostfile.

Example To start SearchWorker's manullay:

1. For each (host, port) in the hostfile start a SearchWorker:

java -Xmx2048m -classpath .:<java-lib>/gnu-getopt.jar edu/princeton/function/distspell/SearchWorker <configuration file> <port>

1. Start SpellMaster with the "-d: argument:

java -Xmx2048m -classpath .:<java-lib>/gnu-getopt.jar edu/princeton/function/distspell/SpellMaster -c <configuration file> -p 7008 -n <hostfile> -d

Clients

SpellMaster clients should connect to the port specified in the "-p" argument.

Testing

See also buglist

Status

- All unit tests pass (including SearchTest).
- Integration tests being implemented.

Unit Testing

We use EclEMMA [3] (http://www.eclemma.org/) (EMMA [4] (http://emma.sourceforge.net/userguide_single/userguide.html) for Eclipse) for code coverage testing, and the JUnit [5] (http://www.junit.org/) [6] (http://junit.sourceforge.net/doc/cookbook/cookbook.htm) framework for unit testing (JUnit is built in Eclipse). For additional details about GWT application testing, refer to the GWT Testing guide [7] (http://code.google.com/webtoolkit/doc/latest/DevGuideTesting.html) .

The unit test code is in DistributedSpell/test. The following is *not* tested by the unit tests:

- Correctness of correlation functions (stubs for these tests are in ExpressionDatasetTests.java).
- Functions for cross-evaluating search results in SearchResults.java.
- Homology with more than one organism (this code is not currently in use).
- Homology with an orthology file (this code is not currently in use).
- Correctness of search results (part of integration tests).
- SpellMaster-SearchWorker interface, including all methods with only networking code (part of integration tests).
- SpellMaster-client interface, including all methods with only networking code (part of integration tests).
- Sequential search code (in SearchThread.java).

Integration Testing

Methodology

The correctness of Distributed Spell is compared against the result of the old sequential Spell implementation. Note that the result of this version differ from the public Spell server (http://imperio.princeton.edu:3000/yeast). A correct result is defined to be a result where the gene weights and datasets weights are maximum 0.05 different than in the sequential version. Exact match is usually not possible due to floating point inaccuracies.

For the test we use two compendia:

- A yeast compendium with 115 datasets. This allows us to test the correctness against previously published results.
- A human compendium with 724 datasets.

Three types of queries are used:

1. *Good* queries chosen from expert curated gold standards. These return results with highly correlated genes.
2. *Bad* queries chosen from expert curated gold standards. These return results with very low gene correlation.
3. Random queries.

The gold standards, queries, and reference results are all in the DistributedSpell/test/data directory. The format of each file is described in a comment at the beginning of each file.

To run the tests the SearchTest.java Junit module, SearchWorkerClient.java Java application, and SpellMasterClient.java application are used. The latter two implements a client that send queries, receives results, and compares these against a reference result obtained by using the sequential version of the code (generated using the SearchServerReference class). The queries are sent using three different approaches: by function call, directly to a search worker, and to the SpellMaster.

Search result correctness

1. Run the SearchTest Junit tests which will verify that the sequential and parallel search functions returns similar results.
2. Run the runIntegrationTests.py script in the DistributedSpell/scripts/ directory. This test will run the following tests with the yeast compendium, and then with the human compendium:
    1. Run SearchWorkerClient with one single threaded worker to verify that the SearchWorker does not introduce any errors. This will test the following Search worker requests:
        1. Load dataset: verify that the request does not fail.
        2. Get datasets: verify that the request does not fail and that the received list is identical to the list sent in the previous request. This test is repeated after the duplicate load and after doing the queries.
        3. Duplicate load dataset: verify that duplicate loads return an error
        4. Partial query: verify that a search with valid parameters does not return an error and that the results match a reference result.
        5. (Handling of invalid parameters is tested in the MasterWorkerProtocolTest unit test).
    2. Run SearchWorkerClient with one worker with 1, 2, and 8 threads. This will test if multi-threading introduces any bugs.
    3. Run SpellMasterClient with one single threaded worker to verify that the SpellMaster does not introduce any errors. That is test:
        1. Dataset load
        2. Search with default parameters (return all genes and all datasets)
        3. Search where only the top 50 gene results are returned.
        4. Search where only the top 25 datasets are returned (or 4 for the micro compendium)
        5. Search where the number of genes and datasets to return are set to the number of genes and datasets in the compendium.
        6. Search where the top 100 and top 75 datasets are returned.
        7. MySpell search for public results.
        8. (Handling of invalid parameters is tested in the ClientMasterProtocolTest unit test).
    4. Run SpellMasterClient2 with one single threaded worker to verify that refined search does not introduce any error. The SpellMasterClient2 will test:
        1. Refined search with default parameters.
        2. Refined search where only the top 50 gene results are returned.
        3. Refined search where only the top 25 datasets are returned (or 4 for the micro compendium)
        4. Refined search where the number of genes and datasets to return are set to the number of genes and datasets in the compendium.
        5. Refined search where the top 100 and top 75 datasets are returned.
    5. Run SpellMasterClient with 1, 2, a#2, 4, 8 and 16 workers relative to a single worker (all with 8 threads).nd 8 workers each with 1, 2, and 8 threads.
    6. Run SpellMasterClient with 8 workers with 4 threads where the data replication is 1, 2, 4 and 8.

7. Run SpellMasterClient with 8 workers with 4 threads with 8x data replication, but use 1, 2, 4, and 8 search workers for the queries.

Invalid Parameters

The SearchWorker interface is tested with invalid parameters in the MasterWorkerProtocolTest unit test.

The SpellMaster interface is tested with invalid parameters in the ClientMasterProtocolTest unit test.

Performance Tests

Verify that runtime improvement is close to linear for:

1. Single worker with 2, 4, and 8 threads relative to sequential version.
2. 2, 4, 8 and 16 workers relative to a single worker (all with 8 threads).
3. When setting per query workers to 2, 4, 8 and 16 workers relative to a single worker (all with 8 threads).

Verify that performance is close to identical for:

1. When using 1, 2, 3, 4, and 8, replicas relative to N replicas (with N = 16).

System Testing

Tests that combine DistributedSpell and SpellWeb2 are described in Function:Troilkatt/SpellWeb2#System_Testing

Tests that combine DistributedSpell and MySpell are described in Function:Troilkatt/MySpell#System_Testing

Regression Testing

Eclipse can be configured to run the Unit tests at compile time (invoke JUnit in the build process).

The runIntergrationTests.py can be used to run all integration tests.

Performance Testing

Stress Tests