Distributed BEAGLE HOWTO

Marc Dubreuil

Laboratoire de Vision et Systèmes Numériques (LVSN),
Département de Génie Électrique et de Génie Informatique,
Université Laval, Québec (QC), Canada, G1K 7P4.
E-mail: dubreuil@gel.ulaval.ca

For Distributed BEAGLE version 0.9

August 26, 2004

Contents

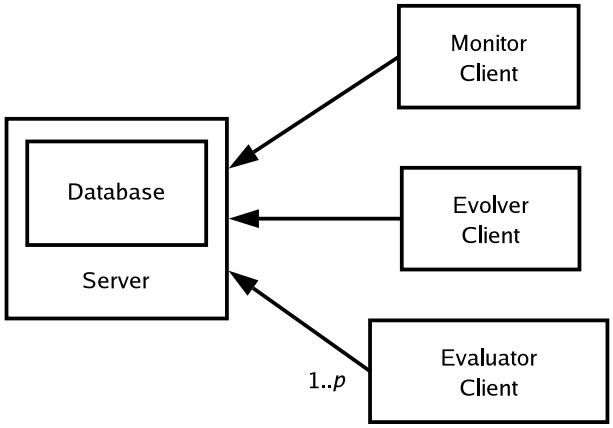

Figure 1: Distributed BEAGLE is a client/server Architecture.

1 Overview

Distributed BEAGLE comprises four main components: the database, the server, one or more evolver clients, and a pool of evaluation clients. Figure 1 illustrates the system's architecture. The system works on data by separating the EC generation concept into two distinct steps: deme evolution and fitness evaluation. Deme evolution is done by evolver clients. It consists in applying several genetic and natural selection operations to evolve the deme through one generation. Once a deme has evolved, the composing individuals need to be evaluated for fitness. Fitness evaluation is done by evaluator clients. When all individuals have been evaluated, the generation is finished and the demes are ready to be evolved again. Since the computational bottleneck in EC is usually the fitness evaluation (at least for hard problems), an evolution with Distributed BEAGLE is usually conducted using a single evolver client and as much evaluator clients as possible (one per available processor).

The *database* guarantees data persistency by storing the demes and the evolution state. This is an important element of robustness for such a software system, where computations may last weeks or even months. Furthermore, the use of a common database separates software elements specific to EC from population storage management. Data are classified into two categories: demes that require evolution, and individuals that need evaluation.

The *server* acts as an interface between the different clients and the database. The primary function of the server is to dispatch the demes to evolver clients, and the individuals to evaluator clients. The number of individuals sent to an evaluator client depends on a load balancing mechanism. The mechanism dynamically adjusts the number of individuals sent to each evaluator node based on its recent performance history. The server is often called DAGS which means *DAGS is an Agile Grid Scheduler*. It is not specific for a given evolutionary algorithm.

An *evolver client* sends requests for a deme to the server, and then applies selection and genetic operations on this deme. These operations are usually specific to the implemented EC flavor. An *evaluator client* sends requests to the server for individuals that need to be evaluated. The number of individuals returned by the server is variable and depends on the (recent) past performance of the client. The evaluator clients are specific to the problem at hand. A *monitor client* sends requests to the server in order to retrieve the current

state of the evolution, allowing users to monitor it. This client does not modify database content.

The *load balancing* policy is to regulate the size of individual sets in order to approximately achieve constant time periods between successive evaluator requests. For fast clients, more individuals are sent in order to lower communication latency. For slow clients, fewer individuals are sent in order to minimize synchronization overheads at the end of an evaluation cycle. This design choice of Distributed BEAGLE allows efficient performances on loosely-coupled multi-processor systems such as Beowulf clusters or LANs of workstations.

When all individuals of a deme have been distributed and after a time out proportional to the load balancing time period, individuals that have been sent to lagging nodes are automatically re-dispatched to other nodes by the server until it receives a fitness response. If duplicate answers are received, only the first one is kept and all others are discarded. This approach both reduces the time needed to complete a generation and assures general fault tolerance for the system.

The critical failure point of Distributed BEAGLE is the server. If it crashes, the whole system comes down. But data persistency is guaranteed by the database. Another interesting aspect of Distributed BEAGLE is the robustness over computer or network failures. If a client (slave) fails or is unreachable over the network, the data it was processing is not permanently lost, it is simply re-assigned to one of its peers.

An important design choice of Open BEAGLE is the use of a population with multiple demes and synchronous migration. This is independent of Distributed BEAGLE. It separates multiple parallel populations from evolution distributed on multiple processors. We strongly believe that there is no need to evolve a separate deme on every available processor. Moreover, the size of the demes does not need to be proportional to processor performance. Adding a degree of liberty by separating these elements enable a finer control over the evolution parameters.

Distributed BEAGLE is cross-platform Unix/Windows library, developed using an SQLite database[1], portable TCP/IP socket and threading classes[2] as the communication protocol and XML (eXtensible Markup Language) for data encoding. The main advantages of using XML is that messages are strongly structured, they are represented using portable character encoding, and there is a variety of XML parsers available to process them. In most cases, it is almost perfectly transparent to the BEAGLE programmer. Typically, only one lines of code must be changed: the one that specifies the evolver object. The standard BEAGLE evolver must be replaced by the Distributed BEAGLE evolver, the application needs to be recompiled and linked to the `dbeagle` library.

This document can be considered as a HOWTO. It explains how to create, compile and execute your own Parallel and Distributed Evolutionnary Computation (PDEC) example. The reader must have tried Open BEAGLE[3] before attempting to use Distributed BEAGLE as many explanations are not given again in this document.

2 Building libraries

Distributed BEAGLE needs to link on three libraries: Open BEAGLE, SQLite and zlib. "SQLite is a small C library that implements a self-contained, embeddable, zero-configuration SQL database engine.[4]". "zlib is

[1] http://www.sqlite.org
[2] http://www.gel.ulaval.ca/~parizeau/PACC
[3] http://www.gel.ulaval.ca/~beagle/
[4] http://www.sqlite.org/

designed to be a free, general-purpose, legally unencumbered – that is, not covered by any patents – lossless data-compression library for use on virtually any computer hardware and operating system.[5]"

Building Open BEAGLE libraries won't be discussed here, as there is already an excellent document that can be found on Open BEAGLE web page.

2.1 Building Libraries on Unix

The first thing you should know is that SQLite won't work over an NFS directory, the disk must be local. Also, Distributed BEAGLE presently work with SQLite version 2.8.x. It was not tested over the newly released version 3.x and it's not compatible for the moment. This version is still considered beta. The version that we recommend is sqlite-2.8.14.tar.gz(955231 bytes) as it is the last stable version released. Distributed BEAGLE will probably start using SQLite version 3.x in the next six months.

Since DAGS is a multithreaded environment, you have to specify -DTHREADSAFE=1 to the CFLAGS option when using SQLite. SQLite should also be compiled in optimization mode. The following steps should be used if you're already lost.

```
1) Download SQLite (sqlite-2.8.14.tar.gz) in temporary directory: ~/tmp/
2) Start a console and go to the directory of sqlite-2.8.14.tar.gz: cd tmp
3) Unpack the file: gunzip sqlite-2.8.14.tar.gz and tar -xvf sqlite-2.8.14.tar
4) Go inside the new directory sqlite: cd sqlite
5) ./configure CFLAGS="-DTHREADSAFE=1 -O3"
6) make
7) make install
```

If you don't have permission to write to the directory */usr*, you have to specify another command to configure: *–prefix=/path/to/install*

zlib is already present on most Unix distribution, so there is no need to compile the zlib library.

2.2 Building libraries on Windows

SQLite can be compiled on Windows as described on the following web page: http://www.sqlite.org/cvstrac/wiki?p=HowToCompileWithVsNet

zlib on Windows is already built and can be downloaded on its web page. Inside the zip file, zlib1.dll can be found which will be needed later..

3 Building DAGS and Distributed BEAGLE

DAGS and Distributed BEAGLE are two different things: DAGS is the server that store the population and send individuals to crunchers to be evaluated while Distributed BEAGLE is a library that act as a bridge between an Open BEAGLE application and DAGS.

[5] http://www.gzip.org/zlib/

Typically, four directories will be compiled: *dags*, the library that can communicate with the server, *dags-server*, the server, *dbeagle*, the library that your application will link to and *monitor*, a simple monitor program to query the server's state.

The other directory, "examples", contains the Open BEAGLE examples modified to be used with Distributed BEAGLE. It should be noted that most of these examples don't work that great in a multi-client environment because the time needed to compute individuals' fitness is a lot lower than the time needed to transmit, parse and store the individuals in a database. The best case scenario is spambase.

3.1 Building DAGS and Distributed BEAGLE on UNIX

For compiling DAGS and Distributed BEAGLE, if you did not install Open BEAGLE and/or SQLite in the root directory (usually /usr) you must specify where the compiler can find them.

```
./configure --prefix=/where/to/install/DAGS \
--with-beagle=/where/to/find/openbeagle/install \
--with-sqlite=/where/to/find/sqlite/install
```

In some cases where your admin compiled Open BEAGLE and SQLite, they are installed and accessible by every users. So the command would be

```
./configure
```

To build DAGS and Distributed BEAGLE and to install them where you told configure, issue the following command:

```
make; make install
```

3.2 Building DAGS and Distributed BEAGLE on Windows

Note: The only version that we know that can compile DAGS and Distributed BEAGLE is Visual Studio .NET 2003. Other programs may work, but they were not tested.

After downloading the file on sourceforge.net and extracting the zip file, rename the directory to **distributed**. The compiled DLL/LIB of SQLite and zlib should be put in the directory distributed/MSVCPP/zlib and distributed/MSVCPP/sqlite, respectively.

To compile DAGS and Distributed BEAGLE, start Microsoft Visual Studio .NET 2003. Open the solution file **distributed.sln**, located in distributed/MSVCPP/ and build the four project, starting with "dags", "dags-server", "dbeagle" and "monitor".

```
File->Open Solution->distributed.sln
```

By default, these commands assume that the tree of directories is

```
WhereYourSourceAre\beagle\
WhereYourSourceAre\distributed\
WhereYourSourceAre\distributed\MSVCPP\sqlite\
WhereYourSourceAre\distributed\MSVCPP\zlib
```

The sqlite directory includes the sqlite.dll and sqlite.h. The zlib directory includes zlib1.dll and zlib.h.

4 Executing an example

All the examples included in Distributed BEAGLE are parallel version of the Open BEAGLE ones. Let's take a genetic algorithm example: onemax. Fitness evaluation is done by computing the number of bits that are equal to one. The greater the number of ones, the better is the individual.

First thing to do is to compile onemax example. In UNIX, go to the directory distributed/example/GA/onemax. And execute the following commands

```
./configure; make; make install
```

If you installed everything as proposed in the last section, it should compile without a problem. If you did not install in the default place, you can tell "configure" where to find Open BEAGLE and Distributed BEAGLE.

```
./configure --with-beagle=DIR --with-dbeagle=DIR --with-dags=DIR
```

On Windows, the solution file can be found in distributed/example/GA/onemax/MSVCPP.

4.1 Starting programs

dags-server and onemax has now been compiled. The following lines give the workflow of what should be done to run a PDEC.

```
a) start the server
b) start one evolver on the same CPU as the server
c) start one or more crunchers on many CPU
d) wait until the evolver finish
e) stop the crunchers by telling them a SIGINT (ctrl-c) or SIGTERM
f) stop the server by telling it a SIGINT (ctrl-c) or SIGTERM
```

4.1.1 dags-server

The command *dags-server -h* gives you options of the server that can be changed. Normally, the most important ones are "-d filename" to create a default configuration file, "-f filename" to use a specific configuration file and "-v X" to specify the level of verbosity the server will have. By default, the level of verbosity is

2, which means that every entering connections will be logged in the console. A degree higher than 2 will start using a log file. A degree lower than 2 will be a stealth mode where no messages will be displayed. More options can be found in a configuration file. To create a configuration file with default values, issue the command *dags-server -d dagsserver.conf*. The default configuration file is the following:

```
<DAGS>
  <Port value="9123"/>
  <MaxConnections value="25"/>
  <MaxThreads value="6"/>
  <DBFileName value="universe.db"/>
  <EnableLoadBalancing value="false"/>
  <JobsToCruncherPerCycle value="100"/>
  <ExpectedTimeBetweenComms value="60"/>
  <Timeout value="200"/>
  <HistorySize value="5"/>
  <HistoryWeight1 value="0.2"/>
  <HistoryWeight2 value="0.2"/>
  <HistoryWeight3 value="0.2"/>
  <HistoryWeight4 value="0.2"/>
  <HistoryWeight5 value="0.2"/>
  <DBSyncPercent value="100"/>
  <DBSyncGroup value="true"/>
  <MemoryShortMode value="false"/>
  <CompressGroupComms value="0"/>
  <CompressSubgroupComms value="0"/>
  <ErrorsLogFileName value="DAGSserver.err"/>
  <MessagesLogFileName value="DAGSserver.log"/>
</DAGS>
```

The options are explained in the Table 1, Table 2 and Table 3.

4.1.2 clients (evolver + crunchers)

With the examples (like onemax, parity, etc.), you already have the configuration file. When you create your own program, you must pass the following arguments to create the configuration file for the evolver and for the crunchers. It should be noted that in Distributed BEAGLE, the evolver and the crunchers are the same program, they just behave differently depending of the used configuration file.

As an example, let's use onemax. The program name is onemax-client. To create a default configuration file:

```
onemax-client -OBec.conf.dump=onemax-evolver.conf
```

file for the evolver. We will edit it next, but first, we will create a configuration file for the crunchers. You should NEVER use the same name as onemax-client.conf since it will also be used by any cruncher/evolver clients and will crash the crunchers.

Options name	*Units*	*Default*	*Descriptions*
Port	integer	9123	DAGS server's will listen to this given port number for incoming connections from clients. You should never use a port lower than 1024, since they are reserved for privilege uses. Before using any number, you should verify if no other users or programs are currently using it.
MaxConnections	integer	25	The maximum number of connections before refusing connections from clients. It should be set to the number of expected crunchers. If you put less than that number, a delay will be added for every connections.
MaxThreads	integer	6	The number of threads that will execute concurrently. You should never use a number too high because it can slow down the whole process and can take lots of resources (CPU and RAM). With current CPU, this option should be set to more than 1, but less than 10.

Table 1: DAGS server's connection settings

```
onemax-client -OBdb.cruncher.dumpconf=onemax-cruncher.conf
```

will create the configuration file for the crunchers.

evolver configuration file Note: With Distributed Beagle and DAGS, nothing can stop you from using multiple evolvers.

```
[...]
  <Register>
    <Entry key="db.app.name">dbeagle-app</Entry>
    <Entry key="db.evolver.compression">0</Entry>
    <Entry key="db.restart">0</Entry>
    <Entry key="db.server.ip">127.0.0.1</Entry>
    <Entry key="db.server.port">9123</Entry>
[...]
```

The table 4 explains the options specific to Distributed BEAGLE for the evolver.

For a complete list and description of other Open BEAGLE options, just issue the standard help command of Open BEAGLE.

cruncher configuration file

Options name	*Units*	*Default*	*Descriptions*
DBFileName	string	universe.db	SQLite database's filename that will store the population. If this option is not present (not in the configuration file), DAGS will NOT use any database.
CompressGroupComms	integer	0	Compression level used for communications between the server and the evolver. At level 0, no compression is used while the level 10 is the maximum compression possible. The greater the level of compression used, the greater the time it will take to perform the compression.
CompressSubgroupComms	integer	0	Compression level used for communications between the server and the crunchers. At level 0, no compression is used while the level 10 is the maximum compression possible. The greater the level of compression used, the greater the time it will take to perform the compression.
ErrorsLogFileName	string	DAGSserver.err	Filename of the errors detected by the server.
MessagesLogFileName	string	DAGSserver.log	Logs are written in the file when using a verbose level greater than 2. At verbose 3, only times of thread are logged. At level 4, lots of information are logged for every connections. By default, the server is running at verbose level 2, which means that every connections are only displayed in the console. To change the level of verbosity, *dags-server -v x* where x represents the degree of verbosity.

Table 2: DAGS server's communication, database and logs settings

Options name	*Units*	*Default*	*Descriptions*
EnableLoadBalancing	bool	false	If a load balancing mecanism should be used or not.
JobsToCruncherPerCycle	integer	100	Number of individuals (also called jobs) that will be sent to every clients if no load balancing is used or before any load balancing is performed (first generation).
ExpectedTimeBetweenComms	integer	60	The cruncher will receive the number of individuals that it can process in the given time in seconds.
Timeout	percent	200	The percent that it will wait before a timeout. After that time, the server will send the individuals to another clients if no answer is received. A value of 100 (100% of ExpectedTimeBetweenComms) can be a little too close. A lower value should never be used. A value of 0 means that NO redispatch will be done (use at your own risk).
HistorySize	integer	5	The number of statistics (history) used for the loadBalancing.
HistoryWeightX	double	0.2	For every stats, a specific weight can be used. The first weight (i.e. HistoryWeight1) is computed by the last received individuals while the last (i.e. HistoryWeight5) is the oldest statistic. The sum of all weights must be equal to 1. As an example, if the value of HistoryWeight1 is 0.7, then the sum of the other weights (HistoryWeight2 to HistoryWeight5) must be equal to 0.3.
DBSyncPercent	percent	100	The percent the server will wait before syncing with the database the fitness it received from the clients. If 50% is used, it means that when the server receives 50% or more fitness evaluation, it will sync these individual with the database.
DBSyncGroup	bool	true	If the server also sync individuals genome received from the evolver. If false, the server will sync only individuals with their fitness.
MemoryShortMode	bool	false	If the memory short mode should be used. Slow things down a lot. This option is still experimental.

Table 3: DAGS server's load balancing and interaction settings

Options name	*Units*	*Default*	*Descriptions*
db.app.name	string	dbeagle-app	The application's name that the server will use to recognize if the received data is valid.
db.evolver.compression	integer	0	The degree of compression used between communications with the server.
db.restart	bool	0	If the client was restarted and that it should regenerate the population. If the server already has a population, this new population will be added.
db.server.ip	string	127.0.0.1	The IP address of the dags-server.
db.server.port	integer	9123	The dags-server's port number.

Table 4: Evolver settings

```
[...]
  <Register>
    <Entry key="db.app.name">dbeagle-app</Entry>
    <Entry key="db.server.ip">127.0.0.1</Entry>
    <Entry key="db.server.port">9123</Entry>
  </Register>
[...]
```

The table 5 explains the options specific to Distributed BEAGLE for the crunchers.

Options name	*Units*	*Default*	*Descriptions*
db.app.name	string	dbeagle-app	The application's name that the server will use to recognize if the received data is valid.
db.server.ip	string	127.0.0.1	The IP address of the dags-server.
db.server.port	integer	9123	The dags-server's port number.

Table 5: Cruncher settings

4.1.3 Monitor

"./dags-monitor -h" gives you the option of the monitor.

```
-f filename : Path and name of monitor's configuration file.
-c filename : Create a default monitor's configuration file.
-D          : Stats about all clients.
-d clientID : Stats about the client with id.
-E          : All environments of every groups.
-e groupID  : Environment of the groupID.
-g groupID  : Get the entire groupID.
-s          : Get DAGS state.
-m filename : Write a milestone and do nothing else.
```

The default configuration file is the following:

```
<SMON>
  <ApplicationName value="dbeagle-app"/>
  <ServerIP value="127.0.0.1"/>
  <ServerPort value="9123"/>
</SMON>
```

The table 6 explains the options specific to the monitor.

Options name	*Units*	*Default*	*Descriptions*
ApplicationName	string	dbeagle-app	The application's name that the server will use to recognize if the received data is valid.
ServerIP	string	127.0.0.1	The IP address of the dags-server.
ServerPort	integer	9123	The dags-server's port number.

Table 6: Monitor settings

5 Creating your own PDEC

The first step is to create your example using Open BEAGLE. It will help you debugging your application. If you are not satisfied with the time it needs to complete the evolutionary process, namely the time needed to compute individuals' fitness, Distributed BEAGLE can then be useful.

For most applications, you can easily convert OpenBEAGLE program to Distributed BEAGLE in 3 easy steps. Let's take a look at the onemax example.

1) add the following lines to OneMaxMain.cpp

```
#include "dbeagle/EvolverBitString.hpp"
using namespace DBeagle;
```

2) Modify the lEvolver variable

```
DBeagle::EvolverBitString::Handle lEvolver =
  new DBeagle::EvolverBitString(lEvalOp, lEncoding);
```

3) Link your application with libraries in this order: dbeagle, dags, zlib, Ws2_32.lib (Windows only).

Note: It should be noted that if your crunchers needs to use some shared "data" to perform a fitness computation, you have to define a class to read and write the data. For an example using this, see the GP examples symbreg-denv, *denv* for distributed environment. It means that the crunchers will get the environment which can includes information to perform fitness evaluation with the same parameters on every CPU. For the symbreg-denv example, the 20 random samples of the symbolic regression are shared by the crunchers.

As you can see, one of best quality of Distributed BEAGLE is that if you find that your application under Open BEAGLE takes too long to compute individuals' fitness, you can convert it to Distributed BEAGLE with little work!

If you have any questions, you can contact me on the Open BEAGLE mailing list.

Marc Dubreuil
openbeagle@yahoogroups.com
dubreuil@gel.ulaval.ca