The GEL Reference Guide

CHUA Ching Lian       Francis TANG

June 16, 2005

Revision: 1.20 – Date: 2005/06/15 07:46:23  Last Author: francis

Contents

1 Introduction 2
   1.1 Workflows . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 2
      1.1.1 Dependence . . . . . . . . . . . . . . . . . . . . . . . . . 2
   1.2 Workspace . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 3
   1.3 Syntactic matters . . . . . . . . . . . . . . . . . . . . . . . . . 3
   1.4 Variables . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 4
      1.4.1 Definition and dereferencing . . . . . . . . . . . . . . . . 4
      1.4.2 Scoping rules . . . . . . . . . . . . . . . . . . . . . . . . 5
   1.5 Expressions . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 5
      1.5.1 Concatenation . . . . . . . . . . . . . . . . . . . . . . . . 5
      1.5.2 Subtraction (basename) . . . . . . . . . . . . . . . . . . 6
   1.6 precedence . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 6

2 Job declarations 6
   2.1 Declaration . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 6
   2.2 Attributes . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 7
      2.2.1 Boolean jobs . . . . . . . . . . . . . . . . . . . . . . . . . 7

3 Statements 8
   3.1 Job instantiation . . . . . . . . . . . . . . . . . . . . . . . . . . 8
   3.2 Sequential composition . . . . . . . . . . . . . . . . . . . . . . 9
   3.3 Parallel composition . . . . . . . . . . . . . . . . . . . . . . . 9
   3.4 Conditional . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 10
   3.5 While loop . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 10
   3.6 Sequential for loop . . . . . . . . . . . . . . . . . . . . . . . . 10
   3.7 Parallel for loop . . . . . . . . . . . . . . . . . . . . . . . . . 11
   3.8 Parallel iteration over files . . . . . . . . . . . . . . . . . . . . 11
   3.9 Precedence of statement constructs . . . . . . . . . . . . . . . . 12

4 Implementation 12
   4.1 MPI options . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 13
   4.2 Local Executor (includes SMP) . . . . . . . . . . . . . . . . . . 14
   4.3 SGE, LSF and PBS (Torque) executors . . . . . . . . . . . . . . 14
      4.3.1 Specific tips for LSF executor . . . . . . . . . . . . . . . 14
   4.4 Condor executor . . . . . . . . . . . . . . . . . . . . . . . . . . 15
1 Introduction

Grid Execution Language (GEL) is a language for describing workflow compositions of programs. Conceptually, it is similar to scripting languages such as bash, perl, and python. The key feature of GEL is that it has inherently parallel constructs—such as parallel for loops, and parallel composition—which allow for parallellism in workflows to be naturally annotated.

1.1 Workflows

A GEL script is a program which describes a workflow, i.e. how and in which order jobs, i.e. component programs, can be executed. By analogy, a typical bash script describes a sequential workflow, i.e. one where the jobs must execute in sequence. In a GEL script, jobs can be explicitly annotated as independent by using the correct parallel constructs. The script interpreters will execute the independent jobs in parallel whenever possible.

1.1.1 Dependence

A job $B$ is dependent on job $A$ if $A$ must complete before $B$ can start. A typical reason for such a dependency is that $A$ creates/modifies files which $B$ uses. In this example, $B$ can assume that the working directory contains the files created/modified by $A$. We generalise this to workflows as follows: a workflow $Q$ is dependent on $P$ if $P$ must complete before $Q$ can start executing.

Conversely, we say that workflows $P$ and $Q$ are independent if: (i) $P$ and $Q$ do not interfere with each other by creating/modifying files of the same name, nor (ii) does the behaviour of one depend on files created/modified by the other, and vice-versa.

For example, if $P$ and $Q$ both append lines to a file $f$, then $P$ and $Q$ are not independent, since the content of $f$ will depend on in which order $P$ and $Q$ were executed. Explicitly, if $P$ and $Q$ each write 100 lines to $f$, then the order in which the lines appear in $f$ depend on the order in which $P$ and $Q$ are executed.

For another example, suppose $P$ writes lines to a file $f$ and $Q$ reads $f$ and writes the number of lines in $f$ to $g$. In this case $P$ and $Q$ are not independent, since $Q$ depends on $f$ which is modified by $P$.

However, if

- $P$ only reads from files $f_1, \ldots, f_j$ and writes to files $g_1, \ldots, g_k$ and
- $Q$ only reads from files $f_1, \ldots, f_j$ and writes to files $h_1, \ldots, h_{\ell}$

then $P$ and $Q$ are independent. One possible implementation can copy files $f_1, \ldots, f_j$ to two different machines, and run $P$ and $Q$ concurrently, one on each machine. When both $P$ and $Q$ have completed, the files $g_1, \ldots, g_k$ from the first and $h_1, \ldots, h_{\ell}$ can be copied back to the same machine, thus providing the illusion that both $P$ and $Q$ had executed on the same machine.

In the case where $P$ and $Q$ are independent, we can (possibility), but are not obliged (non-obligation) to, execute $P$ and $Q$, (i) concurrently, and (ii) in the same working directory. In point (i), possibility allows us to take advantage of multiple processors; non-obligation allows for single threaded implementations.
of an interpreter. In point (ii), possibility allows for optimisations on machines with a shared disk image, e.g. SMP and cluster machines; non-obligation allows Jobs to run on nodes with separate file systems.

Independence is a semantic property of workflows: it is a property dependent on the behaviour of programs. If a GEL workflow declares two workflows as being independent then interpretors can schedule them as independent without checking if they really are.

Note that GEL is only concerned with the temporal dependency between jobs. It does not consider data format dependency. For example, if A writes to a file f and B reads from f, but A writes data in a format not expected by B, then B will likely fail. GEL cannot detect or warn of such potential problems.

1.2 Workspace
A typical GEL project comprises of three types of component directories along with the GEL script. These component directories are used by programs as follows.

1. executable directories: where the executables can be located.
   This is used by the dir attribute in the job declarations.

2. input file directories: where the input data files can be located and copied from.
   There can be as many of these directories as there are jobs. In an extreme case, each job has its own ipdir directory. It is implicitly assumed that the files in these directories are read-only, i.e. no job changes these files.
   This is used by the ipdir attribute in the job declaration.

3. common file directories: where files can be copied from and deposited into.
   These type of directories are like input directories except jobs are allowed to modify the contents. They can be used for gathering of output files. GEL would copy contents into the working directory (explained in section 3.1) before running a job, and then the contents of the working directory back into these directories after running the job/s.
   This is used by the cmdir attribute in the job declaration.

1.3 Syntactic matters
Informally, a GEL program consists of two parts: the first part consists of job declarations; and the second part contains the imperative statements which describe the workflow itself. Figure 1 shows an example program.

Lexically, GEL consists of keywords, identifiers, string and integer literals. The hash (“#”) symbol or double slash (“//”) denotes the start of a comment which ends at the end of the line. White space other than those in string literals are ignored. An identifier must start with a letter or underscore (“_”), followed by zero or more alphanumeric characters or underscore.

A job is a description of how a program should be run, e.g. where to find the binaries, where to find input files, which os/cpu architectures the binaries are for, what command-line arguments to use. A job declaration gives a job a name,
# Here are the job declarations. init and blast are jobs.

\begin{verbatim}
init:={exec="prog1.exe";
    args="";
    dir="bin"}

blast(blast_type, query, db, out) := {exec="blastall";
       args="-p", $blast_type,
            "-m", "9",
            "-i", $query,
            "-d", $db,
            "-o", $out}
\end{verbatim}

# Here is the statement describing the workflow.
# It's actually a compound statement composed of
# smaller statements.

\begin{verbatim}
init;
pforeach db of "*.fsa" do
    blast("blastp","seq.fsa", $db, $db%".fsa".".out")
endpforeach
\end{verbatim}

Figure 1: A GEL program

which must be an identifier. Figure 1 declares a job \texttt{init} and a parameterised
job \texttt{blast}.

A GEL statement describes the temporal dependencies between jobs. GEL
statement operators include: sequential composition (\texttt{;}); parallel composition
\texttt{(\|}); sequential loops (\texttt{while}, \texttt{for}); parallel loops (\texttt{pfor}, \texttt{pforeach}).

Command-line arguments and directories are GEL expressions. Simple expressions are string and integer literals. More complex expressions can also be built up: variable lookup ($\$$); concatenation (\texttt{.}); and string subtraction (\texttt{\%}) — analogous to the GNU \texttt{basename} program.

Variables are names for expressions, i.e. they are immutable. They are introduced in job definitions, and the \texttt{for}, \texttt{pfor} and \texttt{pforeach} constructs. The value of a variable depends on the context in which it is used: this is explained in detail later when we describe the statements constructs.

A precise definition of GEL syntax is given in BNF form in Figure 2.

1.4 Variables

1.4.1 Definition and dereferencing

The lexical definition of a variable is the same as for job name. Examples of valid variables

\texttt{blastall}
\texttt{prog1}
\texttt{run_prog_3}

GEL variables are untyped; they are able to contain either an integer or a string. There are two places to declare/bind a variable in GEL:

1. in a job declaration, in the list of formal parameters; and
2. in the for, pfor, and pforeach statements headers.

To dereference a variable (obtain its content), the programmer must prefix the variable with the dereference symbol ($).

The value of a variable depends on its definition. Formal parameters in job definitions take a value determined at the point where the job is instantiated (see job instantiation later). The iterator variable in the for and pfor constructs take successive values on each iteration (see later). This is similar to the for loop in a language such as C. The iterator variable in the pforeach construct takes successive file names matching a glob pattern in the working directory (see later).

1.4.2 Scoping rules

As mentioned in section 1.4.1, there are two places in which variables can be declared.

In the first case of a job parameter, the variable exists within the definition of the job only. See section 2.1 for an example of such a variable declaration. In the example, three variables: blast_type, query and db, are declared in the job parameters.

In the second case, a variable exists within the statement’s body. For an example see section 2.1, where the value of the variable named database is passed to a blast job within a pforeach statement. If, however, we come across nested statements declaring variables with the same name, this is a syntax error.

A variable used out of scope is a syntax error.

1.5 Expressions

The simplest expressions are string literals and integer literals. A string literal is enclosed in double quotes. Examples:

"I am a string"
"colour"
"*?/"

Integers in GEL use decimal notation. Examples:

100
-999
4

1.5.1 Concatenation

\[ expression1 . expression2 \]

Expressions are concatenated by the "." operator. Examples:

"input".".txt"
$1.$j
"output".$k
1.5.2 Subtraction (basename)

expression1 % expression2

If expression2 is a suffix of expression1, then this expression is expression1 with the suffix removed. If expression2 is not a suffix of expression1, then the value of the whole expression is simply expression1.\(^1\) Examples:

"input.txt"%".txt"
$i%$j
"outfile"%$k

1.6 precedence

The basename and concatenate operators have equal precedence. Therefore, in the absence of parentheses, the interpreter will evaluate a series of expression operators from left to right.

2 Job declarations

2.1 Declaration

A job is declared in GEL with the following syntax:

identifier := \{attribute-value-pair list\}

or

identifier(identifier list) := \{attribute-value-pair list\}

attribute-value-pair list is a semi-colon delimited list of attribute-value-pairs attribute=values, where attribute is one of the attributes explained in section 2.2, and value list is a comma-delimited list of expressions. identifier list is a comma separated list of formal parameters with no repetitions. The repeated use of the same variable in the list of formal parameters is a syntax error. Examples:

prog1 := \{ exec="prog1.exe";
               args="i"\}

blast(blast_type, query, db) := \{exec="blastall";
                 args="-p", $blast_type,
                   "-m", "9",
                   "-i", $query,
                   "-d", $db\}

Job names have global scope and they have to be declared at the beginning of the GEL script.

The definitions are used by job instantiation statements, which are reminiscent of function/procedure calls (see Sec. 3.1).

\(^1\)This is consistent with the GNU utility basename.
2.2 Attributes

GEL attributes are defined as attribute-value pairs. Its syntax is:

\[ \text{attribute} = \text{value list} \]

where \text{value list} in GEL is a comma separated list of GEL expressions.

Explanation of GEL attributes:

1. \text{exec}: the executable to run. Examples \text{exec}="date" or \text{exec}="eog".

2. \text{exectype}: the executable type. This is an optional attribute. The value "mpi" means that the binary is an MPI program.

3. \text{nproc}: the number of processes (cpus) needed by MPI program. By default, \text{nproc}="1" if \text{exectype}="mpi" and \text{nproc}"" or \text{nproc} is not specified.

4. \text{dir}: the relative directory where the executable could be fetched. It is required only if the executable is not in the user’s PATH environment. Examples \text{dir}="execbin", or \text{dir}="bin/executables". If it is not defined or is set to "", then it is assumed that the binary can be found in the PATH environment.

5. \text{args}: a comma separated list of arguments to the job. Example

\[ \text{args}="-infile" , "input.txt" , "-outfile" , "report.txt" , "-num" , 5 \]

Expressions using variables are very effective when used as arguments, for example

\[ \text{args}="-infile" , $f$ , "-outfile" , $f$ % ".fasta" . ".out" \]

6. \text{ipdir}: the relative path to the data directory where any input file required by the job is fetched. Examples \text{ipdir}="ipdir" or \text{ipdir}="motif/hmmpfam".

7. \text{cmdir}: the relative path to a directory with input files, and where result files can be deposited when the job has finished executing. Examples \text{cmdir}="result" or \text{cmdir}="project/results1".

8. \text{arch}: architecture of the target host. (Currently only available for Condor.)

9. \text{opsys}: operating system of the target host. (Currently only available for Condor.)

2.2.1 Boolean jobs

A boolean job is syntactically the same as a normal job. However semantically, the job must not create or modify any file. After the job has finished executing, if it has written anything to stdout, its result is considered to be \text{false}; if it is has written nothing to stdout, its result is considered to be \text{true}.

Boolean jobs are used in the \text{if} and \text{while} statements.
3 Statements

In this section, we introduce, case-by-case, the different types of GEL statements and explain what it means to execute such a statement.

3.1 Job instantiation

Job instantiation is analogous to function/procedure call.

Syntactically it is written either as

\[ J \]

if \( J \) is defined with no formal parameters, or

\[ J(e_1, \ldots, e_n) \]

where \( e_1 \) to \( e_n \) are expressions for the formal parameters in the definition of \( J \).

An example of job instantiation syntax:

\[
\text{prog1} := \{ \text{exec="prog1.exe"; args="i"}\}
\]

This is the most simple GEL script. The statement is a single job instantiation. Another example:

\[
\text{blast(blast_type, query, db)} := \{ \text{exec="blastall"; args="-p", $blast_type, 
}-m", "9",
"-i", $query,
"-d", $db}\}
\]

\[
pforeach database of ".fsa" do
\]
\[
\text{blast("blastp", "seq.fsa", $database) endpforeach}
\]

The example above instantiates job \texttt{blast}, setting the formal parameters \texttt{blast_type}, \texttt{query} and \texttt{db} respectively to strings "\texttt{blastp}"", "\texttt{seq.fsa}" and the value of variable \texttt{database}. The variable \texttt{$database} contains the name of a file in the working directory matching the regular expression, "\texttt{*.fsa}". Each file matching that name creates an instance of job \texttt{blast}.

Semantically, to execute a job instantiation statement, the interpreter take the following steps.

1. If \texttt{ipdir} is defined, it will copy the contents of that directory into the working directory. Note: files of the same name in the working directory will be overwritten.

2. If \texttt{cmdir} is defined, it will copy the contents of that directory into the working directory. Note: files of the same name in the working directory will be overwritten.
3. It will execute the binary specified by `exec`, which is assumed to be located in the directory given by `dir`, or found in the PATH. If the binary is declared to be an MPI program by the `exectype`, it will be run using the appropriate `mpirun` command requesting the number processors specified by `nproc`. The command-line arguments provided to the program are those specified by the `args` attribute.

4. When the binary has terminated, if `cmdir` is defined, the contents of the working directory, i.e. a snapshot, is copied into that directory.

Notes:

1. The `cmdir` attribute will force the interpreter to copy a snapshot of the working directory. Currently, this copies all files. A typical use pattern of the `cmdir` is to specify in the last job an output directory for results of the workflow.

2. The steps above describe how the interpreter might execute a job instantiation statement. However, the interpreter can do other steps, provided it gives the illusion that this is how the binary is executed. For example, the interpreter may choose to copy the input files to a remote machine, run the program on the remote machine, and then copy the files back to the local machine, taking care which files to overwrite. For independent jobs, the interpreter can run them concurrently because of the definition of independence.

3.2 Sequential composition

The sequential composition of workflows $P$ and $Q$ is written as

$$ P ; Q $$

This can be viewed as an explicit annotation that $Q$ depends on $P$. Semantically this means the interpreter must execute $P$ before $Q$. Examples:

prog1 ; prog2

prog2;
prog3;
prog4

3.3 Parallel composition

Parallel composition is written as

$$ P | Q $$

This can be viewed as an explicit annotation that $P$ and $Q$ are independent (see Sec.1.1.1). Thus, the interpreter may execute these two workflows concurrently if possible. Examples:

prog1 | prog2
This states that prog1 and prog2 can be executed concurrently if possible.

Parallel composition is an associative operation and so there is no ambiguity when parentheses are omitted:

\[
\text{prog2 | prog3 | prog4}
\]

This states that prog1, prog2 and prog3 are independent.

### 3.4 Conditional

\[
\text{if } \text{job then } \text{statement1 else } \text{statement2 endif}
\]

Here job is the name of a boolean job. If job evaluates to true (see Sec. 2.2.1), then statement1 is executed, else statement2 is executed. Examples:

\[
\text{if A then }
\text{ B}
\text{ else }
\text{ C; D}
\text{ endif}
\]

### 3.5 While loop

The while statement provides an iterative loop.

Syntax:

\[
\text{while } \text{job do}
\text{ statement}
\text{ endwhile}
\]

Its evaluation condition is similar to the if. If job evaluates to false, the loop exits. If job evaluates to true, then statement, the body of the loop, is executed and the test job is repeated. Examples:

\[
\text{while A do}
\text{ B; }
\text{ (C | D); }
\text{ E}
\text{ endwhile}
\]

### 3.6 Sequential for loop

The for statement allows for an sequential loop. Syntax:

\[
\text{for } \text{identifier = integer1 to integer2 do}
\text{ statement}
\text{ endfor}
\]

The identifier is assigned the value of integer1 initially. If the value of integer1 is smaller than integer2, statement is executed. After each iteration, the identifier is incremented one unit and compared to integer2. If the value of
identifier is larger than integer2, the loop exits.

Examples:

```plaintext
for i = 1 to 7 do
  B
endfor
Execute B seven times in succession.

for var = 45 to 100 do
  A($var); B
endfor
Execute pipeline of A and B 56 times in sequence.
```

### 3.7 Parallel for loop

The `pfor` statement executes parallel instances of a statement. Syntax:

```plaintext
pfor identifier = integer1 to integer2 do
  statement
endpfor
```

By assigning `identifier` the value of `integer1`, the `identifier` is incremented, in unit of one, to the value of `integer2`. If `integer2` is greater than `integer1`, the number of times that `identifier` is incremented is the number of times the `statement` body is parallelised. However if `integer2` is the same as or is smaller than `integer1`, `statement` is never executed.

Examples:

```plaintext
pfor i = 1 to 7 do
  B
endpfor
Execute B seven times (assuming the seven instances of B do not modify the same file system).

pfor var = 45 to 100 do
  A($var); B
endpfor
Execute the 56 pipelines of A and B in parallel.
```

Similar to the parallel composition operator, the programmer must ensure that the statement instances are independent of each other.

### 3.8 Parallel iteration over files

The `pforeach` statement allows for parallel instances of a statement based on files in the workspace matching a glob pattern. Syntax:

```plaintext
pforeach identifier of “glob regular expression” do
  statement
endpforeach
```

The `identifier` is assigned values that the `glob regular expression` matches in
the current workspace. All files in the working directory are matched against *glob pattern*. For each matching file, an instance of *statement* is executed where *identifier* has assigned the file name. Examples:

```gel
pforeach file of "*.txt" do
    prog2;
endpforeach
```

Executes prog2 as many times in parallel as the number of matches found in the regular expression "*.txt".

```gel
pforeach query of "input???.dat" do
    blast($query, "sw.database")
endpforeach
```

Blast takes in the variable query as input and query contains the matches to the expression "input???.dat". Blast is executed as many times in parallel as the number of matches found.

### 3.9 Precedence of statement constructs

Generally, operators in GEL are evaluated from left to right but precedence is higher for the sequential than the parallel operator. Statement example:

```gel
A; B | C; D
A; (B | C); D
```

The above two statements can evaluate differently. In first case, it is A in sequence with B are executed in parallel with C which is in sequence with D. In the second case, A is first executed, followed by B and C in parallel. Finally D is executed after B and C finish. A, B, C and D could be *if*, *while*, *for*, *pfor*, and *pforeach* statements and the precedence will still apply.

### 4 Implementation

There are GEL interpreters supporting execution on the same host, on clusters with LSF, PBS and SGE schedulers, and on Condor-based Grids. All interpreters are packaged together as one Java application.

The interpreter can be invoked with the command `gel`. If invoked without command-line options, `gel` will show the interpreter version number, and display a help summary.

When executing a script, the interpreters will create a new directory of the form `Jtmpnnnnnnnnn`, where `nnnnnnnnn` is a randomly generated decimal number, for use as the *working directory*. Executables and binaries will be copied into this directory as the script is processed. The interpreters will also create two directories named *stdout* and *stderr* in which the captured output to stdout and stderr are stored in separate files. The file names vary depending on the interpreter.

The GEL script is specified using the `-f` option, for example:

```
% gel -f myscript.gel
```
By default, the interpreter will try to execute the jobs on the same host, i.e. use the local executor. Different executors can be selected using the following options:

<table>
<thead>
<tr>
<th>long form</th>
<th>short form</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>--local</td>
<td>-l</td>
<td>use local executor</td>
</tr>
<tr>
<td>--sge</td>
<td>-q</td>
<td>use SGE executor</td>
</tr>
<tr>
<td>--lsf</td>
<td>-b</td>
<td>use LSF executor</td>
</tr>
<tr>
<td>--pbs</td>
<td>-p</td>
<td>use PBS executor</td>
</tr>
<tr>
<td>--condor</td>
<td>-r</td>
<td>use Condor executor</td>
</tr>
</tbody>
</table>

Table 1: Attribute to Execution Engine Table. Legend: ⋆:required. •:optional. ◦:not applicable. This table illustrate the various job attributes and their applicability to the different execution environments. When an attribute shown as not applicable, it will be ignored. MPI programs are currently not supported in the Local and Condor executors.

The following sections describe the MPI-specific options and the features of the different executors. Table 1 displays the attributes supported by the different executors.

4.1 MPI options

Different target hosts have different MPI implementations, e.g. MPI on shared memory machines, MPI over IP (mpich-p4), MPI over Quadrics (QsNet) and MPI over Myrinet (mpich-gm). To allow easy porting of GEL scripts, the interpreter provides the following options to specify which MPI implementation is required:

<table>
<thead>
<tr>
<th>Option</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>--mpi, -k</td>
<td>Specify MPI implementation, must be one of mpichp4, mpichgm or mpiqsnet</td>
</tr>
<tr>
<td>--mpipath, -m</td>
<td>Path to “mpirun” script</td>
</tr>
</tbody>
</table>

For example, suppose we would like to use MPI over Myrinet (mpich-gm), and the mpirun script is in /usr/local/mpich-gm/bin/:

```bash
% gel --sge
   --mpi=mpichgm --mpipath=/usr/local/mpich-gm/bin/mpirun
   -f myscript.gel
```
If no --mpipath option is provided, then GEL will use the default depending on the value of the --mpi option, based on the following table:

<table>
<thead>
<tr>
<th>MPI</th>
<th>default mpirun script</th>
</tr>
</thead>
<tbody>
<tr>
<td>mpichp4, mpichgm</td>
<td>mpirun as found in PATH</td>
</tr>
<tr>
<td>mpiqsnet</td>
<td>prun as found in PATH</td>
</tr>
</tbody>
</table>

If --mpi is not used, and there is at least one MPI job defined in the GEL script, then the interpreter will terminate with an error message.

The selection of MPI implementation to use is currently a workflow-global option. I.e., it is not possible to specify an MPI implementation on a job-by-job manner.

4.2 Local Executor (includes SMP)

The local executor will try to run the jobs on the same host. It is the user’s responsibility to ensure that the binaries are suitable for the host machine. For example, if GEL is running on an i386 Linux platform, then all the binaries referenced in the gel script must be compiled for the i386 Linux platform.

This executor allows independent jobs to be executed concurrently. This allows the executor to use more than one processor on a multi-processor machine. To prevent GEL from running arbitrarily many jobs concurrently, the user can specify a limit using the --nproc option. For example,

\[
% \text{gel} \ --\text{local} \ --\text{nproc}=8 \ -f \ \text{myscript.gel}
\]

will execute myscript.gel, running jobs on the same machine, with at most 8 concurrent jobs.

Since GEL polls for when jobs are complete\(^2\), for scripts with several small concurrent jobs, the makespan (i.e. total execution time) can be reduced by specifying a limit larger than the number of physical processors.

4.3 SGE, LSF and PBS (Torque) executors

These executors will submit jobs to a cluster’s scheduler. They assume that the cluster is configured with a shared file system, and that the compute nodes are homogeneous, i.e. are of the same architecture and have the same software installed.

Since the jobs are submitted the scheduler, the jobs will run on the compute nodes of the cluster.

4.3.1 Specific tips for LSF executor

The LSF executor by default will submit jobs to the default queue. The executor can be configured to submit to a different queue by setting the \text{LSB_DEFAULTQUEUE} environment variable. For example

\[
% \text{export} \ \text{LSB_DEFAULTQUEUE}="long"
% \text{gel} \ --\text{lsf} \ -f \ \text{myscript.gel}
\]

will make the LSF executor submit jobs to queue \text{long}.

\(^2\)The polling is delayed to prevent the interpreter from competing for CPU resources from the jobs themselves.
4.4 Condor executor

The Condor executor will extract parts of the gel script and submit them as DAGMan scripts (DAGs). For simple GEL scripts, it might be possible to submit the whole script as one DAG.

The executor assumes that the Condor cluster does not have a shared file system. It relies on Condor to copy the binaries and related files to the remote host for execution. Since two jobs might run on separate systems with distinct file systems, if both jobs modify the same file, then the result of one job will overwrite the other.

Here is an example of a script which will likely give unexpected results with the Condor executor. Suppose $a$ and $b$ are programs that each append 100 lines to a file $f.txt$. Then the script

\[
\begin{align*}
A & := \{ \text{exec = "a"} \} \\
B & := \{ \text{exec = "b"} \} \\
A & \mid B
\end{align*}
\]

If this script is run with the local executor, then after execution $f.txt$ will contain a total of 200 lines, 100 of which are written by $a$ and another 100 by $b$, possibly interleaved with each other. Similar results would be expected with the SGE, LSF and PBS executors. However, with the Condor executor, one would expect one of four results:

1. $f.txt$ contains 100 lines, all written by $a$;
2. similarly for $b$;
3. $f.txt$ contains 200 lines, 100 written by $a$ followed by 100 written by $b$; and
4. vice-versa.

The first two cases result from $a$ and $b$ running concurrently. The last two cases result from $b$ running after $a$ has finished running, and vice-versa.

For specification purposes, this GEL script is considered to be incorrect: it syntactically states that $a$ and $b$ are independent but semantically they are not.

4.4.1 Platform heterogeneity in Condor

The Condor executor allows for heterogeneous workflows; i.e. workflows whose component programs have require different platforms for execution. The attributes `arch` and `opsys` can be used to control this. For example:

\[
\begin{align*}
\text{progw}(f) & := \{ \\
& \quad \text{exec = "winapp.exe"} \\
& \quad \text{dir = "bin"} \\
& \quad \text{arch = "INTEL"} \\
& \quad \text{opsys = "WINNT51"} \\
& \quad \text{args = $f$}
\}
\end{align*}
\]

\[
\begin{align*}
\text{progl}(f) & := \{
\end{align*}
\]
exec = "linapp.exe";
dir = "bin";
arch = "INTEL";
opsys = "LINUX";
args = $f
}

declares a job progw which must run on the INTEL architecture and WINNT51 operating system (Windows XP), an a job progl which must run on the LINUX operating system. Currently, the Condor executor passes the value of arch and opsys straight to Condor and so the meaning and behaviour of specific strings (e.g. WINNT51 and LINUX) are determined by Condor itself. In particular, if a non-existent arch or opsys is provided, then the job will be queued indefinitely.

5 Tips for creating portable GEL workflows

GEL workflows can use MPI programs for component jobs on SGE, PBS and LSF target platforms. The currently supported MPI implementations are MPICH-p4 (IP), MPICH-GM (Myrinet) and QsNet (Quadrics). Portability between clusters, whose architecture, operating system, MPI implementation and scheduler differ, can be achieved for GEL workflows so long as there are no binaries compiled for platform-specific targets in the project.

Ideally, when creating a GEL workflow with portability in mind, try to follow the following guidelines.

1. Install the binaries on the target platform, e.g. let them form part of a standard configuration, so that they are no longer an integral part of the workflow. This solution is good if the binaries meet the following criteria:

   (a) the programs are hard to compile from sources because they require too many dependencies;

   (b) the source code rarely changes, i.e. this is part of a stable application; for example, EMBOSS and BLAST can be considered to be stable applications which should be available on the target platform.

Related to the point 1b, in-house developed code which evolves as quickly as the workflow should avoid this solution. The next point would better serve the needs of such code.

2. Configure a make (or other) script which can rebuild all the binaries from source code. This way, the binaries can be recompiled for each target platform. This applies also to MPI programs, which should be compilable using the default/standard mpicc command.

3. Use platform-neutral interpreted scripts where appropriate. For example, Bourne Shell, Perl and Python scripts are generally supported on most UNIX-style platforms, provided the requisite interpreter is present.
6 Tips for creating platform-heterogeneous GEL workflows

The previous section gave tips for creating portable GEL workflows. However, there are occasions where this is just infeasible. For example, a workflow may use binaries for which the source code is not available. In this case, GEL can solve the problem using the heterogeneous platform features of GEL on Condor. (Future versions of GEL will support heterogeneity in the other executors too.)

In this case, it is recommended that the files in the project be arranged in scheme similar to

myworkflow/ Parent directory
  myworkflow.gel GEL script
  bin/intel-winnt51/ Directory for WinXP binaries
  bin/intel-linux/ Directory for Linux x86 binaries
  data/ Directory for data
  results/ Directory for storing the results

Given this directory scheme, the following might be an example GEL script.

progw := {
  exec = "prog1" ;
  dir = "bin/intel-winnt51" ;
  arch = "INTEL" ;
  opsys = "WINNT51"
}
progl := {
  exec = "prog2";
  dir = "bin/intel-linux"
  arch = "INTEL" ;
  opsys = "LINUX"
}
progw ; progl

This example first runs a Windows XP program prog1 and then a Linux program prog2.

Current implementations of GEL do not support jobs with alternative binaries. When GEL supports jobs with alternative binaries, it would be possible, for example, for a job to be declared to have two binaries which compute the same result but are compiled for two different platforms.

7 Grammar

7.1 keywords

Table 2 shows the reserved keywords in GEL.

7.2 comments

A comment in GEL is preceded by the # symbol or a double forward slash, //.

Example:
# This is an example which illustrates how to run a workflow.

**Job Declaration**

```gelf
init:={exec="prog1.exe";
         args="";
         dir="bin"}

blast(blast_type, query, db, out) := {exec="blastall";
         args="-p", $blast_type,
                 
         
         "-m", "9",
         
         "-i", $query,
         
         "-d", $db,
         
         "-o", $out}
```

init; //Also a comment

//Here the databases found matching "*.fsa"
//is blasted with the query file seq.fsa.
//Furthermore, notice the operation done
//on the variable db in the last parameter
//of the job:blast.
//db is dereferenced and its value is operated
//on by the basename operator which strips out
//the .fsa suffix and then concatenates with the
//.out suffix. This ensures that the parallel
//blast outputs will be presented in different
//files and also illustrates the call-by-value
//semantics used in GEL for passing variables
//to jobs.
pforeach db of "*.fsa" do
    blast("blastp","seq.fsa", $db, $db%".fsa".".out")
endpforeach

### 7.3 Grammar in Backus-Naur Form

The grammar for GEL is provided in Figure 2 for the reader. Words in double quotes are to be taken as literal tokens.
A Execution Profile Plot

The GEL distribution includes a script called `formatdata.sh` in `utils/plotprofile` which can make plots of the a script’s execution profile. You will need `bash`, `awk` and the GNU utility `seq` to run this script, and also `gnuplot` to generate the plot.

When you run GEL, it creates a directory `JtmpXXX`, where `XXX` is a 10-digit number, containing a series of files with the `.profile` extension. These files contain the profile information, i.e. when a job instance started and finished executing. The script `formatdata.sh` will format the data in these files into a form suitable for plotting in `gnuplot`.

To plot the profile:

1. change into the directory with the many `.profile` files
2. run `formatdata.sh` and redirect its output to file
3. run `gnuplot` to plot the output generated by `formatdata.sh`. 

Figure 2: Grammar for GEL.
For example, I ran the example tissue (as distributed with GEL), and it created a directory profile/Jtmp2126652642. GEL is installed in $HOME/gel-2.0. This is an example session.

% cd profile/Jtmp2126652642
% ls
INSTANCE0.profile  INSTANCE16.profile  INSTANCE22.profile  INSTANCE3.profile
INSTANCE10.profile  INSTANCE17.profile  INSTANCE23.profile  INSTANCE4.profile
INSTANCE11.profile  INSTANCE18.profile  INSTANCE24.profile  INSTANCE5.profile
INSTANCE12.profile  INSTANCE19.profile  INSTANCE25.profile  INSTANCE6.profile
INSTANCE13.profile  INSTANCE20.profile  INSTANCE26.profile  INSTANCE7.profile
INSTANCE14.profile  INSTANCE21.profile  INSTANCE27.profile  INSTANCE8.profile
INSTANCE15.profile  INSTANCE22.profile  INSTANCE28.profile  INSTANCE9.profile
% /home/francis/gel-2.0/utils/plotprofile/formatdata.sh > profile
% gnuplot

GNUPLOT
Version 3.7 patchlevel 3
last modified Thu Dec 12 13:00:00 GMT 2002
System: Linux 2.6.10-1.771_FC2

Copyright(C) 1986 - 1993, 1998 - 2002
Thomas Williams, Colin Kelley and many others

Type 'help' to access the on-line reference manual
The gnuplot FAQ is available from
http://www.gnuplot.info/gnuplot-faq.html

Send comments and requests for help to <info-gnuplot@dartmouth.edu>
Send bugs, suggestions and mods to <bug-gnuplot@dartmouth.edu>

Terminal type set to 'x11'
gnuplot> set term png
Terminal type set to 'png'
Options are ' small color'
gnuplot> set output "profile.png"
gnuplot> plot "profile" with financebars

After quitting gnuplot, there is the file profile.png (see Fig. 3). In the profile plot, the vertical axis is time relative to the start of the first job, and the horizontal axis is job instance number. For each job instance, a line represents its execution span: the line ends denote when the job started and finished.
Figure 3: Example profile plot.