Project acronym: EDGeS

Project full title: Enabling Desktop Grids for e-Science

Grant agreement no.:RI 211727

Tutorial:

Application Porting with the DC-API

Client side

Author: Gabor Szmetanko/ Attila Csaba Marosi

Date: 03/07/2009

1 Preparation

Please use the same virtual machine today you used yesterday.

We will first create the client (worker) application. All of the files you will need is in **vm1XX.term:~boinc-sample/edges-tut-files/uppercase_dg/.** A sequential *uppercase.cpp* file was renamed to *client.cpp*. We will extend this skeleton with new code to create the BOINC client application for the Uppercase application.

If not found, you can fetch the application source tarball using these commands (replace **sample** with the project name you used when you created the BOINC project):

```
ssh root@vm1XX.terem
sudo su – boinc-sample
wget http://www.lpds.sztaki.hu/~atisu/edges-tut-files.tar.gz
tar -xzf  edges-tut-files.tar.gz
logout
```

The **edges-tut-files/uppercase_dg/** directory also contains the skeleton of the master application (master.cpp), a header file, a make file, a sample input file, and some other files which are not important at the moment.

.

1.1 The skeleton

To make the application porting process easier for you, we have already prepared the skeleton of the client. The skeleton implements all the functionality not related to the DC-API.

The skeleton has the following functions:

- **resolveFileName**: resolves a file in the DG environment
- **readCheckpointFile**: restores the current character position from the checkpoint file
- **initFiles**: initialises the input and output files
- **doCheckpoint**: stores the current character position into the checkpoint file
- **checkClientEvents**: checks and handles any incoming client events

- **doWork**: the real work
- **main**: connects them all together

It also contains some basic include statements, constants, and global variables.

Some of the functions are already implemented, while others should be completed by you. Places in the source code, where you should insert new code, are marked with a comment similar to the following:

```
/* TODO: Declare the global variables for the householding of the WUs */
```

1.2 Installation of the DC-API packages

To use the DC-API, we have to install the DC-API packages first.

Login to the assigned virtual machine (password is 'kanuka')

```
ssh root@vm1XX.terem
```

Install the DC-API libraries and G++, which we need to develop applications

```
sudo apt-get install libdcapi-boinc-dev g++ make
```

Go to the directory of the tutorial files

```
sudo su – boinc-sample
cd␣./edges-tut-files/uppercase_dg
```

2 Initialisation of the DC-API

2.1 Open the client.cpp source file in your preferred editor

2.2 Insert the new include statements at the top of the file

```
#ifdef _WIN32
#include <windows.h>
#include <dc_win32.h>
#else
#include <unistd.h>
#endif

#include <stdlib.h>
#include <stdio.h>
#include <ctype.h>
#include <dc_client.h>
```

New lines are coloured **green** in the text. Leave everything else unchanged! In the following sections, new code blocks will always be coloured **green**. Code that should be removed will be coloured **red**.

2.3 Locate the main function and insert a call to the DC-API initialisation function at the very beginning of the function

```
int main(int argc, char *argv[]) {
      int retval;
      retval = DC_initClient();
      if (retval != DC_OK) {
            fprintf(stderr, "Failed to initialise the DC-API."
                  "DC-API Return value was: %d\n", retval);
            DC_log(LOG_CRIT, "Failed to initialise the DC-API. "
                  "Return value was: %d", retval);
            DC_finishClient(DC_ERR_CONFIG);
      }
      DC_log(LOG_INFO, "DC-API was initialized successfully.");

      initFiles();
      …
}
```

3 Identification of input/output files

The mapping in the DC-API is done by the `DC_resolveFileName` function. It must be invoked to get the real file path of a logical file name.

The function has the following parameters:

- File type (DC_FILE_IN, DC_FILE_OUT, DC_FILE_TMP)
- The logical name

The function returns the physical file path belonging to the specified logical file name.

Both the master and the client will use the names of the input and output files (in.txt, out.txt). To achieve greater maintainability, we extract the two file names into a separate header file, so that it can be used by both applications. It is a good practice having a separate file for the common things used by both applications.

3.1 Open common.h and have a look at its contents

3.2 We do not need the define statements in client.cpp any more, replace them (in client.cpp) with an include statement

```
#include <stdlib.h>
#include <stdio.h>
#include <ctype.h>
#include <dc_client.h>

#include "common.h"

#define INPUT_LABEL    "in.txt"
#define OUTPUT_LABEL   "out.txt"
```

New lines are coloured **green**. Lines that should be removed are coloured **red**.

3.3 Update the resolveFileName function with the file resolution code

The function we used in the sequential version to initialise the files was named `initFiles`. We will extend this function to resolve the DG file names. Two files (+1 for checkpointing) should be resolved before we can use them. We will create a general file name resolver function and use that in the `initFiles` function

```
char *resolveFileName(DC_FileType type, char *logicalName) {
      char *resolvedName = DC_resolveFileName(type,
            logicalName);

      if (!resolvedName) {
            fprintf(stderr, "Could not resolve the file: %s\n",
                  logicalName);
            DC_log(LOG_CRIT, "Could not resolve the file: %s\n",
                  logicalName);
            DC_finishClient(1);
      }

      DC_log(LOG_INFO, "%s has been resolved as %s",
            logicalName, resolvedName);

      return resolvedName;
}
```

3.4 Update the initFiles function to resolve the files

```
void initFiles(void) {
      char *fileName;
      fileName = resolveFileName(DC_FILE_IN, INPUT_LABEL);
      inFile = fopen(fileName, "rb");
      if (!inFile) {
            fprintf(stderr, "Could not open the input file.\n");
            DC_log(LOG_CRIT, "Could not open the input file.");
            DC_finishClient(1);
      }
      free(fileName);
      …
      fileName = resolveFileName(DC_FILE_OUT, OUTPUT_LABEL);
      outFile = fopen(fileName, "wb");
      if (!outFile) {
            fprintf(stderr, "Could not create"
                  " the output file.\n");
            DC_log(LOG_CRIT, "Could not create"
                  " the output file.");
            DC_finishClient(1);
      }
      free(fileName);
}
```

4 Implementation of the concrete computation

The DG version does the same computation as the sequential version. The difference is that it works on different files (chunk of the original file). The original sequential version of the application has a function called `doWork`, which is in charge of making the characters uppercase in the input file. The files have been already resolved; the `doWork` function can work in the same way. The only thing we have to do is to replace some function calls with their DC-API counterparts.

Real applications do real computation that takes time; here we just emulate it by adding a sleep call in the main loop.

4.1 Modify the doWork function

```
void doWork(void) {
      int c;
      while ((c = fgetc(inFile)) != EOF) {
            c = toupper(c);
            if (fputc(c, outFile) == EOF) {
                  fprintf(stderr, "fputc error");
                  DC_log(LOG_CRIT, "fputc error");
                  DC_finishClient(1);
            }
            fflush(outFile);
            sleep(1);
      }

      if (fclose(inFile)) {
            fprintf(stderr, "Failed to close the input "
                  "file.\n");
            DC_log(LOG_CRIT, "Failed to close the input file.");
            DC_finishClient(1);
      }
      if (fclose(outFile)) {
            fprintf(stderr, "Failed to close the output "
                  "file.\n");
            DC_log(LOG_CRIT, "Failed to close the output "
                  "file.");
            DC_finishClient(1);
      }
}
```

5 Reporting the fraction of the work completed

The client should periodically report the fraction of work done back to the core client. The client's supervisor process (BOINC core client) can show the progress of the application to the user. The fraction can be reported with the `DC_fractionDone` function. This function accepts a value between 0 and 1, which is the fraction of work completed so far, given that 0 means that no work has been done and 1 means that the computation is ready.

5.1 Store the current character position, declare a global variable in client.cpp

```
#include "common.h"

FILE *inFile;
FILE *outFile;

int currentPosition = 0;
```

5.2 To validate the contents of the checkpoint file, we need to store the size of the input file in a variable; declare a global variable for this purpose

```
#include "common.h"

FILE *inFile;
FILE *outFile;

int currentPosition = 0;
int totalSize;
```

5.3 Insert a call to DC_fractionDone into and increment the current position in the main loop of the doWork function

```
while ((c = fgetc(inFile)) != EOF) {
      c = toupper(c);
      if (fputc(c, outFile) == EOF) {
            fprintf(stderr, "fputc error");
            DC_log(LOG_CRIT, "fputc error");
            DC_finishClient(1);
      }
      currentPosition++;
      sleep(1);
      DC_fractionDone(currentPosition / (double)totalSize);
}
```

6 Notifying the core client of the completion

The client should call the `DC_finishClient` function on completion. This function can be used to report successful or failed computations. We have already used it for reporting errors, now we will use it to report successful computation.

6.1 Insert a call to DC_finishClient at the end of the main function

```
initFiles();
doWork();
printf("Work finished.\n");

DC_finishClient(0);
return 0;
```

7 Compiling the client

You will find a ready to use make file in the tutorial directory. In order to compile DC-API client applications, you have to pass some additional libraries to the linker. You can obtain these library flags with the following command:

pkg-config -libs dcapi-boinc-client

You do not have to change the make file, it has been prepared for you.

7.1 Compile the client with the supplied make file

```
make client
```

7.2 If you have done everything right, the client will compile and a binary with the same name will come into existence

8 Running the client in standalone mode

DC-API client applications can run in standalone mode. In standalone mode DC-API specific features will be ignored and the application will run as a normal sequential application. In this mode the application expects input files and produces output files in the same directory with their logical names (file resolution is ignored), and checkpointing will be disabled.

8.1 Run the client application in standalone mode

```
./client &
```

8.2 The application should produce out.txt, check it!

```
tail –f out.txt
```

9 Cheating

The completed *client.cpp* can be found in the same directory named *client_final.cpp*.

```
mv client.cpp client_my.cpp
cp client_final.cpp client.cpp
```