oTN-2009-01 openlab Summer Student Report

LHC physics simulation using CernVM and BOINC

Jarno Rantala (Supervisor: Ben Segal / IT)
21 August 2009
Version 1
Distribution:: **Public**

Abstract 1
Introduction 1
1 Background and previous work 2
2 BOINC wrapper 2
2.1 Original wrapper 2
2.2 Wrapper for virtual machines 3
3 CernVM 5
4 Co-Pilot 5
5 Results 6
5.1 Worker application in a host machine 6
5.2 Worker application in a VM 6
5.3 Co-Pilot jobs 6
5.4 Notes 7
5.5 Acknowledgements 7
6 Appendices 7
6.1 job.xml files 7
6.1.1 The job.xml file used with worker.py computations 7
6.1.2 The job.xml used with Copilot jobs 8
6.2 The code of worker applications 9
6.2.1 Original worker 9
6.2.2 worker.py 11
6.3 Original wrapper code 12
6.4 VMwrapper code 22

Abstract

The basic goal was to set up a prototype cluster running LHC simulation jobs on BOINC hosts. We created a BOINC project server to manage a set of hosts (Windows, Linux or MacOSX) which register as willing to run Virtual Machines (VM's) on behalf of an LHC experiment. These VM's run a CernVM image to deploy the experiment's software environment, and use Co-Pilot agents to interface to the experiment's job production system via an Co-Pilot gateway. ALICE was the first LHC experiment to be demonstrated, with ATLAS, LHCb and CMS to follow as soon as possible.

Introduction

Over the last 3 years, work has been ongoing to support the execution of large physics application programs, developed under the Red Hat Scientific Linux operating system for the LHC experiment collaborations, on platforms volunteered by their owners via the BOINC (http://boinc.berkeley.edu/) distributed computing system. These platforms run Windows, MacOS and various flavours of Linux. To solve issues of cross-platform compatibility and to interface well with the experiments' existing programming and job production systems, it was decided to run these applications in Virtual Machines and to use the recently developed CernVM (http://cern.ch/cernvm/) facility to build and update the virtual images that are run.

This paper describes a major final step in this work, the development and testing of a new BOINC Wrapper

program which allows an unmodified BOINC client to run such virtual applications successfully. This Wrapper is not restricted to running just the desired LHC/CernVM applications but will allow any Virtual Machine based BOINC application to be developed.

1 Background and previous work

➔ First work done in 2006 (Daniel Lombrana Gonzalez):

Published as: **"Customizable Execution Environments with Virtual Desktop Grid Computing".** Presented at the Parallel and Distributed Computing and Systems Conference (PDCS 2007), Cambridge, Massachusetts, USA (November 19–21, 2007).

➔ Next work in 2007 (David Weir):

BOINC and Paravirtualization: what needs to be considered before we can run BOINC in 'black box' virtual machines on a large scale without problems.
(https://twiki.cern.ch/twiki/bin/view/LHCAtHome/BOINCAndParavirtualization)
BOINC and Atlas: setting up a VMWare image that can execute Atlas physics applications, and some comments on using Pythia for progress reporting. Includes scripts and RPMs for creating an Atlas-ready guest OS.
(https://twiki.cern.ch/twiki/bin/view/LHCAtHome/BOINCAndAtlas)
BOINC and Atlas Job transforms: Providing a cookbook-style approach on how to run real physics applications in form of job transformation scripts in the BOINC Atlas environment.
(https://twiki.cern.ch/twiki/bin/view/LHCAtHome/BOINCAndAtlasJobtransform)

➔ Next work done in 2008 (Ben Segal, David Weir, Kevin Reed et al):

General considerations for **"Running apps in virtual machines"**
(http://boinc.berkeley.edu/trac/wiki/VmApps)

➔ Next work done in 2009:

A Python API for BOINC (David Weir):
(http://plato.tp.ph.ic.ac.uk/~djw03/boinc-python/documentation-0.3.1/)
A host <-> guest VM communication system for Virtual Box (David Garcia Quintas):
(http://boinc.berkeley.edu/trac/wiki/VirtualBox)

2 BOINC wrapper

2.1 Original wrapper

The original wrapper application (http://boinc.berkeley.edu/trac/wiki/WrapperApp), which is standard BOINC source code, was written to run applications without doing modifications to the applications. The wrapper communicates with the BOINC core client i.e. suspending, resuming and measuring cpu time of the applications.

The wrapper reads an xml-file called job.xml which contains a sequence of tasks which should be run by the wrapper. Each task contains the name of application to run, the name of files to which connect the stdin, stdout and stderr. One can also define a command line to give to the application and, if the application uses a checkpoint file, the name of the file should also be defined. Each task also has a weight which tells how much time a task would take compared to other tasks. For example, if task X takes twice the time compared to task Y then the weight of X should be 2 and task Y 1.

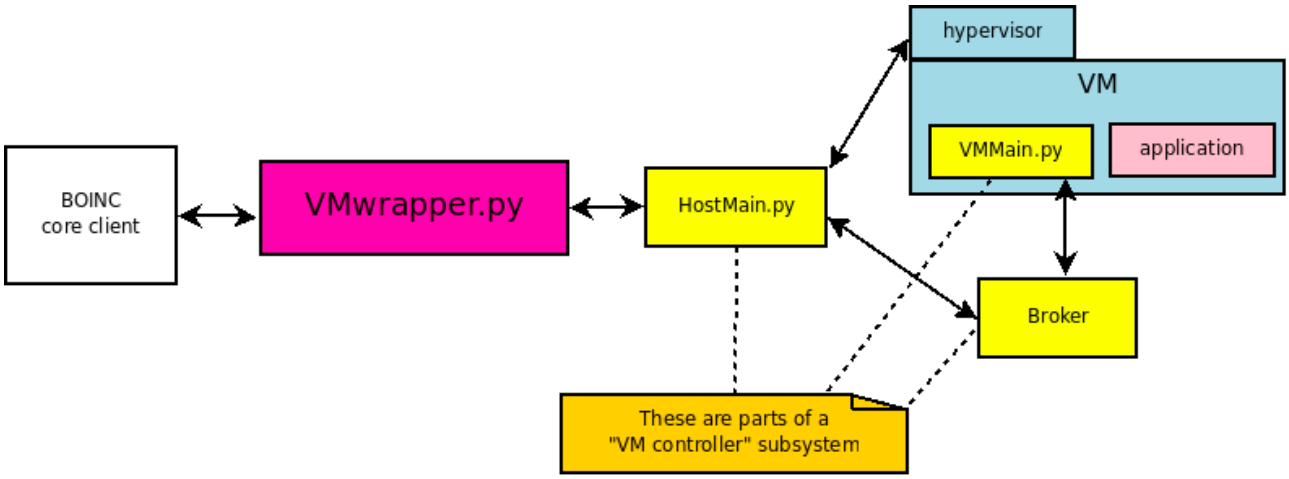
Format of job.xml:

```
<job_desc>
    <task>
        <application>worker</application>
        [ <stdin_filename>stdin_file</stdin_filename> ]
        [ <stdout_filename>stdout_file</stdout_filename> ]
        [ <stderr_filename>stderr_file</stderr_filename> ]
        [ <command_line>--foo bar</command_line> ]
        [ <weight>X</weight> ]
        [ <checkpoint_filename>filename</checkpoint_filename> ]
    </task>
    [ ... ]
</job_desc>
```

2.2 Wrapper for virtual machines

The wrapper for virtual machines called "VMwrapper" (http://boinc.berkeley.edu/trac/wiki/VmApps) works compatibly with the original one. One can give it the same kind of job.xml file and VMwrapper will do the same as the original wrapper. In addition, one can now run applications in virtual machines. VMwrapper suspends, resumes or aborts the VM computations under control of the volunteer via the BOINC core client.

VMwrapper is written in Python using BOINC API Python bindings written by David Weir. VMwrapper uses VM controllers written by David Garcia Quintas to communicate with these guest VMs. The VM controller code is also written in Python. The system architecture is shown here:



VMwrapper can copy input and application files to a VM and run a command there. After that, it can copy VM output files to the volunteer's host machine. There are appropriate extra tags that we have implemented in job.xml files which can be specified to do this, as explained now.

Format of job.xml for VMwrapper:

```
<job_desc>
    <unzip_task>
        <application></application>
        .
        .
        <command_line></command_line>
    </unzip_task>
    <VMmanage_task>
```

```
        <application></application>
        .
        .
        <command_line></command_line>
    </VMmanage_task>
    <task>
        <virtualmachine></virtualmachine>
        <image></image>
        <application></application>
        <copy_app_to_VM></copy_app_to_VM>
        <copy_file_to_VM></copy_file_to_VM>
        <copy_file_to_VM></copy_file_to_VM>
        <stdin_filename></stdin_filename>
        <stdout_filename></stdout_filename>
        <stderr_filename></stderr_filename>
        <copy_file_from_VM></copy_file_from_VM>
        <command_line></command_line>
        <weight></weight>
    </task>
</job_desc>
```

There are two special kinds of tasks: VM managing tasks "VMmanage_task" and unpacking tasks "unzip_task". The unpacking tasks are performed before any other tasks. They are used to unpack a packed file from the project-directory to the slot-directory. VM managing tasks are used to control VM's and they are started only if there is a task using a VM. These VM managing tasks have to run in parallel with any tasks using a VM because they do the communication with the VM. We need to run a Python script called "HostMain.py" and a broker for this communication. ActiveMQ (http://activemq.apache.org/) has been used as the broker in our tests.

The descriptor for each task includes:

Tag	Description
virtualmachine	The name of the virtual machine
image	The logical name of the image to be loaded
copy_app_to_VM	Specifies if the application should be copied to the VM. (zero or nonzero)
copy_file_to_VM	The logical name(s) of file(s) which should be copied to the VM. (input files, stdin file name)
copy_file_from_VM	The name(s) of file(s) copied from the VM after computation. (output files, stdout_filename)
application	The logical name of the application
stdin_filename, stdout_filename, stderr_filename	The logical names of the files to which stdin, stdout, and stderr are to be connected (if any).
command_line*	Command line arguments to be passed to the application.
weight	The contribution of each task to the overall fraction done is proportional to its weight (floating-point, default 1).

checkpoint_filename	The name of the checkpoint file used by the app, if any. When this is modified, the wrapper assumes that a checkpoint has been completed and notifies the core client.

- One can also give a command line to "'VMwrapper.py"' and this is passed to the "'task"'-applications appended to the command line in job.xml (command_line in job.xml + " " + command_line for wrapper). If one gives a file name in command_line (recognized by "./") then the boinc_resolve_filename-method is used to resolve the physical name of the file.

Examples of job.xml files for VM applications are given in Appendix 6.1 below.

3 CernVM

As stated in the Introduction, virtualization permits cross-platform execution of applications developed under each specific LHC software environment. But to support a full set of LHC experiment software packages, a very large size of virtual image is required (8-10 GB); furthermore, some component packages also change frequently, requiring a full image reload in principle. Both these factors would be show-stoppers for the BOINC environment.

The CernVM project was launched as a general solution to the problem of virtual image management for physics computing. Instead of loading each running virtual machine with a full image, only a basic "thin appliance" of about 100MB is loaded initially, and further image increments are downloaded from a CernVM image repository as needed by any given application and choice of LHC experiment. In this way a virtual image of the order of 1 GB suffices to run typical LHC physics problems. An image is not only custom-built incrementally for each application but is cached on each execution node and so remains available for succeeding jobs of the same application type, without further need for communication with the repository. Updates to images are made automatically by the CernVM system when required by changes in the package sources. The use of CernVM thus makes it possible to distribute LHC physics jobs to any reasonably well configured BOINC client node, including all important platform types.

4 Co-Pilot

Each LHC physics experiment needs to run hundreds of thousands of jobs on the various computing fabrics available to it. This job flow is managed by “job production” facilities of several types. We choose to interface BOINC volunteer machines as a “best-effort Cloud”, using the same approach as has recently been taken to interface the ALICE experiment’s physics job production system to other Cloud facilities such as Amazon EC2, reported in the paper CHEP179: “Dynamic Virtual AliEn Grid Sites on Nimbus with CernVM”, A. Harutyunyan, P. Buncic, T. Freeman, and K. Keahey; Computing in High Energy and Nuclear Physics (CHEP), Prague, March 2009. The information in the rest of this section is based on a part of this paper.

The approach is based on “pilot jobs” sent out by the job production system to explore cloud resources, and “pilot agents” running on the cloud’s worker nodes which will request and run received jobs. The approach is general, as all the LHC experiments have pilot job systems, even though they are not identical. To exploit this fact, an “Adapter” is introduced between the pilot job system and the clouds to be interfaced: on each cloud worker node (in our case a BOINC client) a standard “Co-Pilot Agent” runs and uses the Jabber/XMPP instant messaging protocol to communicate with the “Co-Pilot Adapter”, which runs outside the cloud and interfaces to the various job production systems. The use of Jabber/XMPP allows scaling of the system in case of a high load on the Adapter node by just adding new Adapter node(s) to the existing messaging network.

The Co-Pilot Agent code is a standard part of CernVM and is thus freely available to BOINC nodes running with VMwrapper. The Co-Pilot Adapter system exists already in a version for the ALICE experiment, in this case interfacing to the AliEn job production system. (Later versions will become available for ATLAS, CMS and LHCb). So testing with ALICE jobs only requires:

1. Ensure the CernVM Co-Pilot Agent code is loaded in the BOINC nodes.
2. Transfer credentials (agent username and password) from AliEn to the newly created nodes.
3. Start the Co-Pilot Agent service on the worker nodes.

Each Agent is preconfigured with the Jabber server hostname and the Jabber ID of the Co-Pilot Adapter, which are the only configuration parameters it needs. The Co-Pilot Agent then contacts the Co-Pilot Adapter service and requests a job to execute. Upon receiving the job request from an Agent, the Adapter contacts the AliEn Job Broker central service which fetches jobs from the ALICE task queue and sends them to the Adapter, which in turn forwards jobs to the Agents. When the job is done, the Agent reports its results to the Adapter which in turn forwards them to the AliEn central services.

5 Results

5.1 Worker application in a host machine

The first thing to do was to test that VMwrapper would be able to run applications on a host machine, compatibly with the original Wrapper program.. This was done by running the "worker" application which is included in the standard BOINC source code. This application reads from stdin and writes the same input to stdout, reads file "in" and writes these contents to file "out", and uses CPU time at least the time specified in the command line in seconds. This was run successfully in a Linux host machine.

(NOTE: this won't work properly at the moment in a Windows host because the methods used to suspend and resume computing do not work in Windows).

5.2 Worker application in a VM

The second step was to compute the same kind of task in a VM. A Python version of the “worker” application ("worker.py") was written for this, and run in a very simple VM in which Python was already installed.
The job.xml file used in these computations is shown in Appendix 6.1.1. In the server side, the used configuration files for a BOINC task i.e. workunit and result templates are the same as running the worker application with original wrapper. VMwrapper was able to start the VM, copy the application file and input files to the VM and run the application with a command line specified in job.xml. After the computation VMwrapper was able to copy the output file from the VM and write the stdout of the application to the file specified in job.xml. After this, VMwrapper killed the VM managing tasks and exited.

5.3 Co-Pilot jobs

A Co-Pilot job was tested after these two steps. There existed already a Co-Pilot gateway for ALICE Co-Pilot jobs, so it was used. The script called "copilotAgent" was already in the image of CernVM which was used. The script was written to ask for jobs from ALICE’s job production system, run these jobs in the VM, and send the results back to the job production system. In this case, VMwrapper had to just run the script. The job.xml file used in these computations is shown in Appendix 6.1.2. It was confirmed both from the log file in the VM and from the ALICE production system log that these jobs were run successfully. The jobs were in fact just "hello world" jobs, not real physics computations, but provided convincing proof that the whole computation chain was working correctly. The CernVM image used in the VM was configured with the full ALICE system environment required to run real ALICE jobs.

5.4 Notes

In these test computations, the image was not sent as part of the BOINC task because there was not an existing image which would run VMMain.py automatically. So the test was done by starting VMMain.py in the VM manually and saving the state of VM after that. Anyway, it has been tested that VMwrapper.py is able to create a new VM properly. There is demo video about that suspending, resuming and aborting BOINC tasks is working under BOINC core client using VMwrapper.py. The demo can be found from /afs/cern.ch/user/r/rantala/public/VMwrapperDemo.

5.5 Acknowledgements

First of all, I want to thank my supervisor Ben Segal for all the advice and support I have had during the summer. It has been an honour and very pleasant to work under his guidance. I have been extremely lucky to work with brilliant people during my stay at CERN. I want to thank all the people who have helped me with my project: David Garcia Quintas, David Weir, Predrag Buncic, Artem Harutyunyan. I am also very grateful to the CERN openlab secretariat for hosting me and to Markus Nordberg and Robert Pichè for financial support. Finally, I want to thank all the other CERN openlab summer students for the great summer.

6 Appendices

6.1 job.xml files

6.1.1 The job.xml file used with worker.py computations

```
<job_desc>
   <unzip_task>
      <application>tar</application>
      <command_line>-xf ./cctools-2_5_2-i686-linux-2.6.tar</command_line>
      <stdout_filename>stdout_tar</stdout_filename>
      <stderr_filename>stderr_tar</stderr_filename>
   </unzip_task>
   <unzip_task>
      <application>tar</application>
      <command_line>-xf ./apache-activemq-5.2.0.tar</command_line>
      <stdout_filename>stdout_tar</stdout_filename>
      <stderr_filename>stderr_tar</stderr_filename>
   </unzip_task>
   <unzip_task>
      <application>tar</application>
      <command_line>-xf ./boincvm.tar</command_line>
   </unzip_task>
   <VMmanage_task>
      <application>./apache-activemq-5.2.0/bin/activemq</application>
      <stdin_filename></stdin_filename>
      <stdout_filename>stdout_broker</stdout_filename>
      <stderr_filename>stderr_broker</stderr_filename>
      <command_line></command_line>
   </VMmanage_task>
   <VMmanage_task>
      <application>./boincvm/HostMain.py</application>
      <stdin_filename></stdin_filename>
      <stdout_filename>stdout_HostMain</stdout_filename>
      <stderr_filename>stderr_HostMain</stderr_filename>
```

```
      <command_line>./boincvm/HostConfig.cfg</command_line>
   </VMmanage_task>
   <task>
      <virtualmachine>CernVM</virtualmachine>
      <image>/home/jarno/Documents/cernvm-1.2.0-x86/cernvm-1.2.0-x86.vmdk</image>
      <app_pathVM></app_pathVM>
      <application>./worker.py</application>
      <copy_app_to_VM>1</copy_app_to_VM>
      <copy_file_to_VM>stdin</copy_file_to_VM>
      <copy_file_to_VM>in</copy_file_to_VM>
      <stdin_filename>stdin</stdin_filename>
      <stdout_filename>stdout</stdout_filename>
      <stderr_filename></stderr_filename>
      <copy_file_from_VM>out</copy_file_from_VM>
      <command_line>2</command_line>
      <weight>2</weight>
   </task>
</job_desc>
```

6.1.2 The job.xml used with Copilot jobs

```
<job_desc>
   <unzip_task>
      <application>tar</application>
      <command_line>-xf ./cctools-2_5_2-i686-linux-2.6.tar</command_line>
      <stdout_filename>stdout_tar</stdout_filename>
      <stderr_filename>stderr_tar</stderr_filename>
   </unzip_task>
   <unzip_task>
      <application>tar</application>
      <command_line>-xf ./apache-activemq-5.2.0.tar</command_line>
      <stdout_filename>stdout_tar</stdout_filename>
      <stderr_filename>stderr_tar</stderr_filename>
   </unzip_task>
   <unzip_task>
      <application>tar</application>
      <command_line>-xf ./boincvm.tar</command_line>
   </unzip_task>
   <VMmanage_task>
      <application>./apache-activemq-5.2.0/bin/activemq</application>
      <stdin_filename></stdin_filename>
      <stdout_filename>stdout_broker</stdout_filename>
      <stderr_filename>stderr_broker</stderr_filename>
      <command_line></command_line>
   </VMmanage_task>
   <VMmanage_task>
      <application>python</application>
      <stdin_filename></stdin_filename>
      <stdout_filename>stdout_HostMain</stdout_filename>
      <stderr_filename>stderr_HostMain</stderr_filename>
      <command_line>./boincvm/HostMain.py ./boincvm/HostConfig.cfg</command_line>
   </VMmanage_task>
   <task>
```

```
      <virtualmachine>CernVMAgent</virtualmachine>
      <image></image>
      <app_pathVM></app_pathVM>
      <application>./copilotAgent</application>
      <copy_app_to_VM>0</copy_app_to_VM>
      <copy_file_to_VM></copy_file_to_VM>
      <copy_file_to_VM></copy_file_to_VM>
      <stdin_filename></stdin_filename>
      <stdout_filename>stdout_agent</stdout_filename>
      <stderr_filename></stderr_filename>
      <copy_file_from_VM></copy_file_from_VM>
      <command_line></command_line>
      <weight>2</weight>
   </task>
</job_desc>
```

6.2 The code of worker applications

6.2.1 Original worker

```
// This file is part of BOINC.
// http://boinc.berkeley.edu
// Copyright (C) 2008 University of California
//
// BOINC is free software; you can redistribute it and/or modify it
// under the terms of the GNU Lesser General Public License
// as published by the Free Software Foundation,
// either version 3 of the License, or (at your option) any later version.
//
// BOINC is distributed in the hope that it will be useful,
// but WITHOUT ANY WARRANTY; without even the implied warranty of
// MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.
// See the GNU Lesser General Public License for more details.
//
// You should have received a copy of the GNU Lesser General Public License
// along with BOINC.  If not, see <http://www.gnu.org/licenses/>.

// worker - application without BOINC runtime system;
// used for testing wrapper.
// What this does:
//
// copies one line of stdin to stdout
// copies one line of "in" to "out"
// uses 10 sec of CPU time
// (or as specified by a command-line arg)
//
// THIS PROGRAM SHOULDN'T USE ANY BOINC CODE.  That's the whole point.

#include <stdio.h>
#include <stdlib.h>
#include <time.h>

// do a billion floating-point ops
// (note: I needed to add an arg to this;
// otherwise the MS C++ compiler optimizes away
// all but the first call to it!)
```

```
//
static double do_a_giga_flop(int foo) {
    double x = 3.14159*foo;
    int i;
    for (i=0; i<500000000; i++) {
        x += 5.12313123;
        x *= 0.5398394834;
    }
    return x;
}

int main(int argc, char** argv) {
    char buf[256];
    FILE* in, *out;

    fprintf(stderr, "worker starting\n");
    in = fopen("in", "r");
    if (!in) {
        fprintf(stderr, "missing input file\n");
        exit(1);
    }
    out = fopen("out", "w");
    if (!out) {
        fprintf(stderr, "can't open output file\n");
        exit(1);
    }
    fgets(buf, 256, in);
    fputs(buf, out);

    fgets(buf, 256, stdin);
    fputs(buf, stdout);

    int start = time(0);
    int nsec = 10;
    if (argc > 1) nsec = atoi(argv[1]);

    int i=0;
    while (time(0) < start+nsec) {
        do_a_giga_flop(i++);
    }
    fputs("done!\n", stdout);
    return 0;
}

#ifdef _WIN32
#include <windows.h>

// take a string containing some space separated words.
// return an array of pointers to the null-terminated words.
// Modifies the string arg.
// Returns argc
// TODO: use strtok here

#define NOT_IN_TOKEN                0
#define IN_SINGLE_QUOTED_TOKEN      1
#define IN_DOUBLE_QUOTED_TOKEN      2
#define IN_UNQUOTED_TOKEN           3

int parse_command_line(char* p, char** argv) {
    int state = NOT_IN_TOKEN;
    int argc=0;
```

```
    while (*p) {
        switch(state) {
        case NOT_IN_TOKEN:
            if (isspace(*p)) {
            } else if (*p == '\'') {
                p++;
                argv[argc++] = p;
                state = IN_SINGLE_QUOTED_TOKEN;
                break;
            } else if (*p == '\"') {
                p++;
                argv[argc++] = p;
                state = IN_DOUBLE_QUOTED_TOKEN;
                break;
            } else {
                argv[argc++] = p;
                state = IN_UNQUOTED_TOKEN;
            }
            break;
        case IN_SINGLE_QUOTED_TOKEN:
            if (*p == '\'') {
                *p = 0;
                state = NOT_IN_TOKEN;
            }
            break;
        case IN_DOUBLE_QUOTED_TOKEN:
            if (*p == '\"') {
                *p = 0;
                state = NOT_IN_TOKEN;
            }
            break;
        case IN_UNQUOTED_TOKEN:
            if (isspace(*p)) {
                *p = 0;
                state = NOT_IN_TOKEN;
            }
            break;
        }
        p++;
    }
    argv[argc] = 0;
    return argc;
}

int WINAPI WinMain(HINSTANCE hInst, HINSTANCE hPrevInst, LPSTR Args, int
WinMode) {
    LPSTR command_line;
    char* argv[100];
    int argc;

    command_line = GetCommandLine();
    argc = parse_command_line( command_line, argv );
    return main(argc, argv);
}
#endif
```

6.2.2 worker.py

```
#!/usr/bin/env python
import sys, string, time
```

```
def do_a_giga_flop(foo):
   x = 3.14159*foo
   for i in range(1,500000):
      x += 5.12313123;
      x *= 0.5398394834;
   return x

# Main

sys.stderr.write("worker starting\n")
try:
   infile = open("in", "r")
except:
   sys.stderr.write("missing input file\n")
   sys.exit(1)

try:
   outfile = open("out", "w")
except:
   sys.stderr.write("can't opent output file\n")
   sys.exit(1)


outfile.write(infile.read())
sys.stdout.write(sys.stdin.read())

nsec = 10
if len(sys.argv) > 1:
   nsec = int(sys.argv[1])

i = 0
tic = time.time()
toc = time.time()

while toc-tic < nsec:
   do_a_giga_flop(i+1)
   toc = time.time()

sys.stdout.write("done!\n")
sys.exit(0)
```

6.3 Original wrapper code

```
// This file is part of BOINC.
// http://boinc.berkeley.edu
// Copyright (C) 2008 University of California
//
// BOINC is free software; you can redistribute it and/or modify it
// under the terms of the GNU Lesser General Public License
// as published by the Free Software Foundation,
// either version 3 of the License, or (at your option) any later version.
//
// BOINC is distributed in the hope that it will be useful,
// but WITHOUT ANY WARRANTY; without even the implied warranty of
// MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.
// See the GNU Lesser General Public License for more details.
//
// You should have received a copy of the GNU Lesser General Public License
```

```
// along with BOINC.  If not, see <http://www.gnu.org/licenses/>.

// wrapper.C
// wrapper program - lets you use non-BOINC apps with BOINC
//
// Handles:
// - suspend/resume/quit/abort
// - reporting CPU time
// - loss of heartbeat from core client
// - checkpointing
//      (at the level of task; or potentially within task)
//
// See http://boinc.berkeley.edu/wrapper.php for details
// Contributor: Andrew J. Younge (ajy4490@umiacs.umd.edu)

#include <stdio.h>
#include <vector>
#include <string>
#ifdef _WIN32
#include "boinc_win.h"
#include "win_util.h"
#else
#include <sys/wait.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <unistd.h>
#include "procinfo.h"
#endif

#include "boinc_api.h"
#include "diagnostics.h"
#include "filesys.h"
#include "parse.h"
#include "str_util.h"
#include "util.h"
#include "error_numbers.h"

#define JOB_FILENAME "job.xml"
#define CHECKPOINT_FILENAME "checkpoint.txt"

#define POLL_PERIOD 1.0

using std::vector;
using std::string;

struct TASK {
    string application;
    string stdin_filename;
    string stdout_filename;
    string stderr_filename;
    string checkpoint_filename;
        // name of task's checkpoint file, if any
    string command_line;
    double weight;
        // contribution of this task to overall fraction done
    double final_cpu_time;
    double starting_cpu;
        // how much CPU time was used by tasks before this in the job file
    bool suspended;
```

```
    double wall_cpu_time;
        // for estimating CPU time on Win98/ME and Mac
#ifdef _WIN32
    HANDLE pid_handle;
    DWORD pid;
    HANDLE thread_handle;
    struct _stat last_stat;    // mod time of checkpoint file
#else
    int pid;
    struct stat last_stat;
#endif
    bool stat_first;
    int parse(XML_PARSER&);
    bool poll(int& status);
    int run(int argc, char** argv);
    void kill();
    void stop();
    void resume();
        double cpu_time();
    inline bool has_checkpointed() {
        bool changed = false;
        if (checkpoint_filename.size() == 0) return false;
        struct stat new_stat;
        int retval = stat(checkpoint_filename.c_str(), &new_stat);
        if (retval) return false;
        if (!stat_first && new_stat.st_mtime != last_stat.st_mtime) {
            changed = true;
        }
        stat_first = false;
        last_stat.st_mtime = new_stat.st_mtime;
        return changed;
    }
};

vector<TASK> tasks;
APP_INIT_DATA aid;
bool graphics = false;

int TASK::parse(XML_PARSER& xp) {
    char tag[1024], buf[8192], buf2[8192];
    bool is_tag;

    weight = 1;
    final_cpu_time = 0;
    stat_first = true;
    while (!xp.get(tag, sizeof(tag), is_tag)) {
        if (!is_tag) {
            fprintf(stderr, "%s TASK::parse(): unexpected text %s\n",
                boinc_msg_prefix(), tag
            );
            continue;
        }
        if (!strcmp(tag, "/task")) {
            return 0;
        }
        else if (xp.parse_string(tag, "application", application)) continue;
        else if (xp.parse_string(tag, "stdin_filename", stdin_filename))
continue;
```

```
        else if (xp.parse_string(tag, "stdout_filename", stdout_filename))
continue;
        else if (xp.parse_string(tag, "stderr_filename", stderr_filename))
continue;
        else if (xp.parse_str(tag, "command_line", buf, sizeof(buf))) {
            while (1) {
                char* p = strstr(buf, "$PROJECT_DIR");
                if (!p) break;
                strcpy(buf2, p+strlen("$PROJECT_DIR"));
                strcpy(p, aid.project_dir);
                strcat(p, buf2);
            }
            command_line = buf;
            continue;
        }
        else if (xp.parse_string(tag, "checkpoint_filename",
checkpoint_filename)) continue;
        else if (xp.parse_double(tag, "weight", weight)) continue;
    }
    return ERR_XML_PARSE;
}

int parse_job_file() {
    MIOFILE mf;
    char tag[1024], buf[256];
    bool is_tag;

    boinc_resolve_filename(JOB_FILENAME, buf, 1024);
    FILE* f = boinc_fopen(buf, "r");
    if (!f) {
        fprintf(stderr, "%s can't open job file %s\n", boinc_msg_prefix(), buf);
        return ERR_FOPEN;
    }
    mf.init_file(f);
    XML_PARSER xp(&mf);

    if (!xp.parse_start("job_desc")) return ERR_XML_PARSE;
    while (!xp.get(tag, sizeof(tag), is_tag)) {
        if (!is_tag) {
            fprintf(stderr, "%s SCHED_CONFIG::parse(): unexpected text %s\n",
                boinc_msg_prefix(), tag
            );
            continue;
        }
        if (!strcmp(tag, "/job_desc")) {
            fclose(f);
            return 0;
        }
        if (!strcmp(tag, "task")) {
            TASK task;
            int retval = task.parse(xp);
            if (!retval) {
                tasks.push_back(task);
            }
        }
    }
    fclose(f);
    return ERR_XML_PARSE;
}
```

```
#ifdef _WIN32
// CreateProcess() takes HANDLEs for the stdin/stdout.
// We need to use CreateFile() to get them.  Ugh.
//
HANDLE win_fopen(const char* path, const char* mode) {
        SECURITY_ATTRIBUTES sa;
        memset(&sa, 0, sizeof(sa));
        sa.nLength = sizeof(sa);
        sa.bInheritHandle = TRUE;

        if (!strcmp(mode, "r")) {
                return CreateFile(
                        path,
                        GENERIC_READ,
                        FILE_SHARE_READ,
                        &sa,
                        OPEN_EXISTING,
                        0, 0
                );
        } else if (!strcmp(mode, "w")) {
                return CreateFile(
                        path,
                        GENERIC_WRITE,
                        FILE_SHARE_WRITE,
                        &sa,
                        OPEN_ALWAYS,
                        0, 0
                );
        } else if (!strcmp(mode, "a")) {
                HANDLE hAppend = CreateFile(
                        path,
                        GENERIC_WRITE,
                        FILE_SHARE_WRITE,
                        &sa,
                        OPEN_ALWAYS,
                        0, 0
                );
        SetFilePointer(hAppend, 0, NULL, FILE_END);
        return hAppend;
        } else {
                return 0;
        }
}
#endif

void slash_to_backslash(char* p) {
    while (1) {
        char* q = strchr(p, '/');
        if (!q) break;
        *q = '\\';
    }
}

int TASK::run(int argct, char** argvt) {
    string stdout_path, stdin_path, stderr_path;
    char app_path[1024], buf[256];

    strcpy(buf, application.c_str());
```

```
    char* p = strstr(buf, "$PROJECT_DIR");
    if (p) {
        p += strlen("$PROJECT_DIR");
        sprintf(app_path, "%s%s", aid.project_dir, p);
    } else {
        boinc_resolve_filename(buf, app_path, sizeof(app_path));
    }

    // Append wrapper's command-line arguments to those in the job file.
    //
    for (int i=1; i<argct; i++){
        command_line += string(" ");
        command_line += argvt[i];
    }

    fprintf(stderr, "%s wrapper: running %s (%s)\n",
        boinc_msg_prefix(), app_path, command_line.c_str()
    );

#ifdef _WIN32
    PROCESS_INFORMATION process_info;
    STARTUPINFO startup_info;
    string command;

    slash_to_backslash(app_path);
    memset(&process_info, 0, sizeof(process_info));
    memset(&startup_info, 0, sizeof(startup_info));
    command = string("\"") + app_path + string("\" ") + command_line;

    // pass std handles to app
    //
    startup_info.dwFlags = STARTF_USESTDHANDLES;
        if (stdout_filename != "") {
                boinc_resolve_filename_s(stdout_filename.c_str(), stdout_path);
                startup_info.hStdOutput = win_fopen(stdout_path.c_str(), "a");
        }
        if (stdin_filename != "") {
                boinc_resolve_filename_s(stdin_filename.c_str(), stdin_path);
                startup_info.hStdInput = win_fopen(stdin_path.c_str(), "r");
        }
    if (stderr_filename != "") {
        boinc_resolve_filename_s(stderr_filename.c_str(), stderr_path);
        startup_info.hStdError = win_fopen(stderr_path.c_str(), "a");
    } else {
        startup_info.hStdError = win_fopen(STDERR_FILE, "a");
    }

    if (!CreateProcess(
        app_path,
        (LPSTR)command.c_str(),
        NULL,
        NULL,
        TRUE,           // bInheritHandles
        CREATE_NO_WINDOW|IDLE_PRIORITY_CLASS,
        NULL,
        NULL,
        &startup_info,
        &process_info
    )) {
```

```
        return ERR_EXEC;
    }
    pid_handle = process_info.hProcess;
    pid = process_info.dwProcessId;
    thread_handle = process_info.hThread;
    SetThreadPriority(thread_handle, THREAD_PRIORITY_IDLE);
#else
    int retval, argc;
    char progname[256];
    char* argv[256];
    char arglist[4096];
        FILE* stdout_file;
        FILE* stdin_file;
        FILE* stderr_file;

    pid = fork();
    if (pid == -1) {
        boinc_finish(ERR_FORK);
    }
    if (pid == 0) {
                // we're in the child process here
                //
                // open stdout, stdin if file names are given
                // NOTE: if the application is restartable,
                // we should deal with atomicity somehow
                //
                if (stdout_filename != "") {
                        boinc_resolve_filename_s(stdout_filename.c_str(),
stdout_path);
                        stdout_file = freopen(stdout_path.c_str(), "a", stdout);
                        if (!stdout_file) return ERR_FOPEN;
                }
                if (stdin_filename != "") {
                        boinc_resolve_filename_s(stdin_filename.c_str(),
stdin_path);
                        stdin_file = freopen(stdin_path.c_str(), "r", stdin);
                        if (!stdin_file) return ERR_FOPEN;
                }
        if (stderr_filename != "") {
            boinc_resolve_filename_s(stderr_filename.c_str(), stderr_path);
            stderr_file = freopen(stderr_path.c_str(), "a", stderr);
            if (!stderr_file) return ERR_FOPEN;
        }

                // construct argv
        // TODO: use malloc instead of stack var
        //
        argv[0] = app_path;
        strlcpy(arglist, command_line.c_str(), sizeof(arglist));
        argc = parse_command_line(arglist, argv+1);
        setpriority(PRIO_PROCESS, 0, PROCESS_IDLE_PRIORITY);
        retval = execv(app_path, argv);
        exit(ERR_EXEC);
    }
#endif
    wall_cpu_time = 0;
    suspended = false;
    return 0;
}
```

```
bool TASK::poll(int& status) {
    if (!suspended) wall_cpu_time += POLL_PERIOD;
#ifdef _WIN32
    unsigned long exit_code;
    if (GetExitCodeProcess(pid_handle, &exit_code)) {
        if (exit_code != STILL_ACTIVE) {
            status = exit_code;
            final_cpu_time = cpu_time();
            return true;
        }
    }
#else
    int wpid, stat;
    struct rusage ru;

    wpid = wait4(pid, &status, WNOHANG, &ru);
    if (wpid) {
        final_cpu_time = (float)ru.ru_utime.tv_sec +
((float)ru.ru_utime.tv_usec)/1e+6;
        return true;
    }
#endif
    return false;
}

void TASK::kill() {
#ifdef _WIN32
    TerminateProcess(pid_handle, -1);
#else
    ::kill(pid, SIGKILL);
#endif
}

void TASK::stop() {
#ifdef _WIN32
    suspend_or_resume_threads(pid, false);
#else
    ::kill(pid, SIGSTOP);
#endif
    suspended = true;
}

void TASK::resume() {
#ifdef _WIN32
    suspend_or_resume_threads(pid, true);
#else
    ::kill(pid, SIGCONT);
#endif
    suspended = false;
}

void poll_boinc_messages(TASK& task) {
    BOINC_STATUS status;
    boinc_get_status(&status);
    if (status.no_heartbeat) {
        task.kill();
        exit(0);
    }
```

```
    if (status.quit_request) {
        task.kill();
        exit(0);
    }
    if (status.abort_request) {
        task.kill();
        exit(0);
    }
    if (status.suspended) {
        if (!task.suspended) {
            task.stop();
        }
    } else {
        if (task.suspended) {
            task.resume();
        }
    }
}

double TASK::cpu_time() {
#ifdef _WIN32
    double x;
    int retval = boinc_process_cpu_time(pid_handle, x);
    if (retval) return wall_cpu_time;
    return x;
#elif defined(__APPLE__)
    // There's no easy way to get another process's CPU time in Mac OS X
    //
    return wall_cpu_time;
#else
    return linux_cpu_time(pid);
#endif
}

void send_status_message(
    TASK& task, double frac_done, double checkpoint_cpu_time
) {
    double current_cpu_time = task.starting_cpu + task.cpu_time();
    boinc_report_app_status(
        current_cpu_time,
        checkpoint_cpu_time,
        frac_done
    );
}

// Support for multiple tasks.
// We keep a checkpoint file that says how many tasks we've completed
// and how much CPU time has been used so far
//
void write_checkpoint(int ntasks, double cpu) {
    FILE* f = fopen(CHECKPOINT_FILENAME, "w");
    if (!f) return;
    fprintf(f, "%d %f\n", ntasks, cpu);
    fclose(f);
}

void read_checkpoint(int& ntasks, double& cpu) {
    int nt;
    double c;
```

```
    ntasks = 0;
    cpu = 0;
    FILE* f = fopen(CHECKPOINT_FILENAME, "r");
    if (!f) return;
    int n = fscanf(f, "%d %lf", &nt, &c);
    fclose(f);
    if (n != 2) return;
    ntasks = nt;
    cpu = c;
}

int main(int argc, char** argv) {
    BOINC_OPTIONS options;
    int retval, ntasks;
    unsigned int i;
    double total_weight=0, w=0;
    double checkpoint_cpu_time;
        // overall CPU time at last checkpoint

    for (i=1; i<(unsigned int)argc; i++) {
        if (!strcmp(argv[i], "--graphics")) {
            graphics = true;
        }
    }

    memset(&options, 0, sizeof(options));
    options.main_program = true;
    options.check_heartbeat = true;
    options.handle_process_control = true;
    if (graphics) {
        options.backwards_compatible_graphics = true;
    }

    boinc_init_options(&options);
    fprintf(stderr, "wrapper: starting\n");

    boinc_get_init_data(aid);

    retval = parse_job_file();
    if (retval) {
        fprintf(stderr, "can't parse job file: %d\n", retval);
        boinc_finish(retval);
    }

    read_checkpoint(ntasks, checkpoint_cpu_time);
    if (ntasks > (int)tasks.size()) {
        fprintf(stderr, "Checkpoint file: ntasks %d too large\n", ntasks);
        boinc_finish(1);
    }
    for (i=0; i<tasks.size(); i++) {
        total_weight += tasks[i].weight;
    }
    for (i=0; i<tasks.size(); i++) {
        TASK& task = tasks[i];
        w += task.weight;
        if ((int)i<ntasks) continue;
        double frac_done = w/total_weight;
        task.starting_cpu = checkpoint_cpu_time;
```

```
        retval = task.run(argc, argv);
        if (retval) {
            fprintf(stderr, "can't run app: %d\n", retval);
            boinc_finish(retval);
        }
        while(1) {
            int status;
            if (task.poll(status)) {
                if (status) {
                    fprintf(stderr, "app exit status: 0x%x\n", status);
                    // On Unix, if the app is non-executable,
                    // the child status will be 0x6c00.
                    // If we return this the client will treat it
                    // as recoverable, and restart us.
                    // We don't want this, so return an 8-bit error code.
                    //
                    boinc_finish(EXIT_CHILD_FAILED);
                }
                break;
            }
            poll_boinc_messages(task);
            send_status_message(task, frac_done, checkpoint_cpu_time);
            if (task.has_checkpointed()) {
                checkpoint_cpu_time = task.starting_cpu + task.cpu_time();
                write_checkpoint(i, checkpoint_cpu_time);
            }
            boinc_sleep(POLL_PERIOD);
        }
        checkpoint_cpu_time = task.starting_cpu + task.final_cpu_time;
        write_checkpoint(i+1, checkpoint_cpu_time);
    }
    boinc_finish(0);
}
#ifdef _WIN32

int WINAPI WinMain(HINSTANCE hInst, HINSTANCE hPrevInst, LPSTR Args, int
WinMode) {
    LPSTR command_line;
    char* argv[100];
    int argc;

    command_line = GetCommandLine();
    argc = parse_command_line(command_line, argv);
    return main(argc, argv);
}
#endif
```

6.4 VMwrapper code

The source code can be found from https://svnweb.cern.ch/world/wsvn/vmwrapper/.

```
#!/usr/bin/env python

# VMwrpapper.py
# VMwrapper program - lets you use non-BOINC apps with BOINC in volunteers machines or in
virtual machines.
#
# Handles:
```

```
# - suspend/resume/quit/abort
# - reporting CPU time
# - loss of heartbeat from core client
# - checkpointing (at least at the level of task)
#     * volunteers machine: checkpoint filename of application has to be specified
#     * VM: Takes periodic snapshots of the virtual machine
# - (supposed to also handle trickle messaging in future)
#
# See http://boinc.berkeley.edu/trac/wiki/VmApps for details
# Contributor: Jarno Rantala (jarno.rantala@gmail.com)
#
# The original code was made under the CERN openlab summer student program July-August
2009
import sys, string, xmlrpclib, time, subprocess, signal, os, traceback
from boinc import *
from xml.dom import minidom

JOB_FILENAME = "job.xml"
CHECKPOINT_FILENAME  = "checkpoint.txt"
SERVER_PROXY = 'http://localhost:8080/RPC2'
POLL_PERIOD = 1.0
CHECKPOINT_PERIOD = 15*60
TRICKLE_UP_PERIOD = 120
MAX_WAIT_TIME = 60
TASK_TAGS = ['virtualmachine',
          'image',
          'application',
          'copy_app_to_VM',
          'copy_file_to_VM',
          'stdin_filename',
          'stdout_filename',
          'stderr_filename',
          'copy_file_from_VM',
          'checkpoint_filename',
          'command_line',
          'weight']

#-----------------------------------------------------------------------
# Definition of TASK class
#-----------------------------------------------------------------------
class TASK:
   virtualmachine = ""
   image = ""
   application = ""
   copy_app_to_VM = 0
   copy_files_to_VM = []
   copy_stdin_to_VM = 0
   stdin_filename = ""
   stdout_filename = ""
   stderr_filename = ""
   copy_files_from_VM = []
   checkpoint_filename = ""
   command_line = ""
   weight = 1.0
   CmdId = ""
```

```
CmdResults = None
app_process = None  # instance of Popen class of subprocess Module
suspended = 0
starting_cpu = 0.0
final_cpu_time = 0.0
time_checkpointed = 0
# contribution of this task to overall fraction done
final_cpu_time = 0;
starting_cpu = 0;
ready = 0;
exitCode = None

# how much CPU time was used by tasks before this in the job file
suspended = 0 # zero or nonzero (false or true)
wall_cpu_time = 0


def readTag(self, tag, data):
  if tag == "virtualmachine":
    self.virtualmachine = data

  elif tag == "image":
    self.image = data

  elif tag == "copy_file_to_VM":
    self.copy_files_to_VM.append(data)

  elif tag == "application":
    self.application = data

  elif tag == "copy_app_to_VM":
    self.copy_app_to_VM = int(data)

  elif tag == "copy_stdin_to_VM":
    self.copy_stdin_to_VM = int(data)

  elif tag == "stdin_filename":
    self.stdin_filename = data

  elif tag == "stdout_filename":
    self.stdout_filename = data

  elif tag == "stderr_filename":
    self.stderr_filename = data

  elif tag == "copy_file_from_VM":
    self.copy_files_from_VM.append(data)

  elif tag == "command_line":
    self.command_line = data

  elif tag == "checkpoint_filename":
    self.checkpoint_filename = int(data)

  elif tag == "weight":
```

```
      self.weight = int(data)

    else:
      sys.stderr.write("Unknown tag: " + tag + "\n")

# Replace file names in command line with the physical names
# resolved by boinc_resolve_filename-method. Every word which
# starts "./" is recognised as file name.
def resolve_commandline(self):
  newline = ""
  for word in self.command_line.split():
    if word[0:2] == "./":
      newline = newline + " " + boinc_resolve_filename(word)
    else:
      newline = newline + " " + word
  self.command_line = newline

def kill(self, VMmanage):
  if self.virtualmachine != "":
    VMmanage.saveState(self.virtualmachine) # saves and power off the VM
  else:
    if not self.ready:
      self.app_process.kill()

def stop(self, VMmanage):
  self.suspended = 1
  if self.virtualmachine != "":
    VMmanage.pause(self.virtualmachine)
  else:
    self.app_process.send_signal(signal.SIGSTOP)

def resume(self, VMmanage):
  self.suspended = 0
  if self.virtualmachine != "":
    VMmanage.unpause(self.virtualmachine)
  else:
    self.app_process.send_signal(signal.SIGCONT)

def has_checkpointed(self, VMmanage, checkpoint_period):

  if self.virtualmachine != "":
    if time.time()-self.time_checkpointed > checkpoint_period:
      # time to checkpoint
      VMmanage.saveSnapshot(self.virtualmachine, self.checkpoint_filename)
      sys.stderr.write("snapshot at time: "+str(time.time())+"\n")
      self.time_checkpointed = time.time()
      return 1
    else:
      return 0
  else:
    changed = 0
    if self.checkpoint_filename == "":
      return 0
    # is the file changed  ??
```

```
  def cpu_time(self, VMmanage):
    if self.virtualmachine != "":
      # linux VM assumed!!!!
      if self.suspended: # we cannot send a cmd to VM which is paused
        return 0

      cmdid = VMmanage.runCmd(self.virtualmachine, "cat", ["/proc/uptime"])

      wait = 1
      tic = time.time()
      while wait:
        for cmd in VMmanage.listFinishedCmds():
          if cmd == cmdid:
            wait = 0
            break
        # sys.stderr.write("Wait cpu_time cmd: "+str(wait)+"\n")
        time.sleep(1)
        if time.time() - tic > 10:
          sys.stderr.write("It took too long to get cpu time! \n")
          wait = 0

      res = VMmanage.getCmdResults(cmdid)['out']
      return reduce( lambda x,y: x-y,  map(float, res.split()) )

    else:
      # cpu of subprocess
      # sys.stderr.write(str(os.times()[2]) +"\n")
      return os.times()[2]

  def poll(self, VMmanage = ""):

    if self.virtualmachine != "":
      for Cmd in VMmanage.listFinishedCmds():
        if Cmd == self.CmdId:
          self.ready = 1
      if self.ready:
        self.CmdResults = VMmanage.getCmdResults(self.CmdId)
        self.exitCode = self.CmdResults['exitCodeOrSignal']
        self.final_cpu_time = self.CmdResults['resources']['ru_stime'] +
self.CmdResults['resources']['ru_utime']

    else:
      self.exitCode = self.app_process.poll()
      if self.exitCode != None:
        self.ready = 1
        self.final_cpu_time = self.cpu_time(VMmanage)

  def VMrunning(self, VMmanage):
    running = 0
    for VM in VMmanage.listRunningVMs():
      if VM == self.virtualmachine:
        running = 1
    return running

  def runVM(self, VMmanage, commandline = "", max_wait_time = float('inf')):
```

```
app_path = boinc_resolve_filename(self.application)
image_path = boinc_resolve_filename(self.image)
vm_path = boinc_resolve_filename(self.virtualmachine)
input_path = boinc_resolve_filename(self.stdin_filename)

# Append wrapper's command-line arguments to those in the job file.
self.command_line = self.command_line + " " + commandline#+ " < "+ self.stdin_filename

# Check if the virtual machine is already on client
doCreateVM = 1

for VM in VMmanage.listAvailableVMs():
  if VM == self.virtualmachine:
    doCreateVM = 0

if doCreateVM:
  # we assume that base directory of createVM is home_of_boinc/.VirtualBox
  # and that VM is in project directory home_of_boinc/projects/URL_of_project
  # in image_path we have the path of image relative to boinc/slot/n/ directory
  # that's why we have to remove first "../" and then it should be OK.
  image_path = image_path[3:len(image_path)]
  try:
    VMmanage.createVM(self.virtualmachine, image_path)
  except Exception as e:
    sys.stderr.write("Creation of VM failed! \n")
    sys.stderr.write(str(e) + "\n")
    raise Exception(3)

# restore snapshot if there is one
if VMmanage.getState(self.virtualmachine) != "Saved":
  try:
    VMmanage.restoreSnapshot(self.virtualmachine)
  except:
    sys.stderr.write("Restoring snapshot failed. \n")

# start VM
self.time_checkpointed = time.time()
if not self.VMrunning(VMmanage):
  try:
    VMmanage.start(self.virtualmachine)
  except Exception as e:
    sys.stderr.write("Can't start VM: " + self.virtualmachine + "\n")
    sys.stderr.write(str(e) + "\n")
    raise Exception(3)

# wait until VM is running
tic = time.time()
while not self.VMrunning(VMmanage):
  sys.stderr.write("Wait VM '" + self.virtualmachine + "' to run.\n")

  if time.time() - tic > max_wait_time:
    sys.stderr.write("It took too long to VM to run. \n")
    raise Exception(5)

  time.sleep(1)
```

```
    # wait until VM has connected to broker successfully
    tic = time.time()
    while 1:
      try:
        VMmanage.ping(self.virtualmachine)
      except Exception as e:
        sys.stderr.write("waiting for VM to connect to the broker \n")

        if time.time() - tic > max_wait_time:
          sys.stderr.write(str(e)+"\n")
          sys.stderr.write("It took too long to VM to connect to the broker. \n")
          raise Exception(5)

        time.sleep(10)
        continue
      break

    # copy app file to VM
    if self.copy_app_to_VM:
      sys.stderr.write("copy file '"+app_path+"' to VM\n")

      [out, err, status] = VMmanage.cpFileToVM(self.virtualmachine, app_path, self.application)
      if status:
        sys.stderr.write("Can't copy app to VM \n")
        sys.stderr.write(err)
        raise Exception(4)

    # copy files file to VM
    for fileName in self.copy_files_to_VM:
      file_path = boinc_resolve_filename(fileName)
      sys.stderr.write("copy file '"+file_path+"' to VM\n")

      [out, err, status] = VMmanage.cpFileToVM(self.virtualmachine, file_path, fileName)
      if status:
        sys.stderr.write("Can't copy files to VM \n")
        sys.stderr.write(err)
        raise Exception(4)


    if self.application != "" or self.command_line != "":  # we put " " to command_line couple of lines
above
      sys.stderr.write("run command on VM: "+self.application+" "+self.command_line+"\n")
      tic = time.time()
      while 1:
        try:
          self.CmdId = VMmanage.runCmd(self.virtualmachine, self.application,
[self.command_line], '{}',                                    'None', input_path)
        except Exception as e:
          sys.stderr.write("command didn't succeed.. try again.. \n")

          if time.time() - tic > max_wait_time:
            sys.stderr.write(str(e)+"\n")
            sys.stderr.write("Running command in VM didn't succeed. \n")
            raise Exception(5)
```

```
            time.sleep(10)
            continue
          break


    def run(self, commandline = ""):
        app_path = boinc_resolve_filename(self.application)

        stdout_file = None
        stdin_file = None
        stderr_file = None

        if self.stdout_filename != "":
            output_path = boinc_resolve_filename(self.stdout_filename)
            sys.stderr.write("stdout file: "+output_path+"\n")
            stdout_file = open(output_path, "a")

        if self.stdin_filename != "":
            input_path = boinc_resolve_filename(self.stdin_filename)
            sys.stderr.write("stdin file: "+input_path+"\n")
            stdin_file = open(input_path, "r")

        if self.stderr_filename != "":
            err_path = boinc_resolve_filename(self.stderr_filename)
            sys.stderr.write("stderr file: "+err_path+"\n")
            stderr_file = open(err_path, "a")

        # Append wrapper's command-line arguments to those in the job file.
        self.command_line = self.command_line + " " + commandline

        # resolve file names in command line
        self.resolve_commandline()

        sys.stderr.write("wrapper: running "+app_path+" "+"("+self.command_line+") \n")

        # runs application on host machine
        self.app_process = subprocess.Popen((app_path+" "+self.command_line).split(), 0, None,
                                            stdin_file, stdout_file, stderr_file)

#-------------------------------------------------------------------------------
# Definitions of used methods
#-------------------------------------------------------------------------------
def read_job_file(filename):
    input_path = boinc_resolve_filename(filename)

    # open the job file
    try:
        infile = boinc_fopen(input_path, 'r')
    except boinc.error:
        sys.stderr.write("Can't open job file: " + input_path)
        raise Exception(1)

    jobxml = minidom.parse(infile)
    infile.close()
```

```
# read the context of job file
try:
  xmltasks = jobxml.getElementsByTagName("job_desc")[0]
except IndexError:
  sys.stderr.write("Can't read job file: no 'job_desc' tag \n")

tasks = []
# read the attributes of tasks
for xmltask in xmltasks.getElementsByTagName("task"):
  task = TASK()

  for tag in TASK_TAGS:
    try:
      taglist = xmltask.getElementsByTagName(tag)
      for tagxml in taglist:
        data = tagxml.childNodes[0].data
        task.readTag(tag, data)
    except IndexError:
      sys.stderr.write("Task has no "+tag+" \n")

  tasks.append(task)

# read attributes of VMmanage tasks
VMmanageTasks = []
for xmltask in xmltasks.getElementsByTagName("VMmanage_task"):
  task = TASK()

  for tag in TASK_TAGS:
    try:
      taglist = xmltask.getElementsByTagName(tag)
      for tagxml in taglist:
        data = tagxml.childNodes[0].data
        task.readTag(tag, data)
    except IndexError:
      sys.stderr.write("Task has no "+tag+" \n")

  VMmanageTasks.append(task)

# read attributes of unzip tasks
unzipTasks = []
for xmltask in xmltasks.getElementsByTagName("unzip_task"):
  task = TASK()

  for tag in TASK_TAGS:
    try:
      taglist = xmltask.getElementsByTagName(tag)
      for tagxml in taglist:
        data = tagxml.childNodes[0].data
        task.readTag(tag, data)
    except IndexError:
      sys.stderr.write("Task has no "+tag+" \n")

  unzipTasks.append(task)
```

```
    jobxml.unlink()
    return [tasks, VMmanageTasks, unzipTasks]

def read_checkpoint(filename):
    ntasks = 0
    cpu = 0

    try:
        f = open(filename, "r");
    except:
        return [ntasks, cpu]

    data = f.readline().split()
    f.close()

    try:
        ntasks = int(data[0])
        cpu = float(data[1])
    except:
        sys.stderr.write("Can't read checkpoint file \n")
        return [0, 0]

    return [ntasks, cpu]

def poll_boinc_messages(task, VMmanage):
    status = boinc_get_status()
    exit = 0 # if nonzero then VMwrapper should exit
    #sys.stderr.write("status suspended: "+str(status['suspended'])+" \n")
    #sys.stderr.write("status no heartbeat: "+str(status['no_heartbeat'])+" \n")
    #sys.stderr.write("status quit request: "+str(status['quit_request'])+" \n")
    #sys.stderr.write("status abort request: "+str(status['abort_request'])+" \n")

    if status['no_heartbeat'] or status['quit_request'] or status['abort_request']:
        task.kill(VMmanage)
        exit = 1

    if status['suspended']:
        if not task.suspended:
            task.stop(VMmanage)
    else:
        if task.suspended:
            task.resume(VMmanage)

    return exit

def send_status_message(task, VMmanage, frac_done, checkpoint_cpu_time):
    current_cpu_time = task.starting_cpu + task.cpu_time(VMmanage)
    boinc_report_app_status(current_cpu_time, checkpoint_cpu_time, frac_done)

# Support for multiple tasks.
# We keep a checkpoint file that says how many tasks we've completed
# and how much CPU time has been used so far
def write_checkpoint(filename, ntasks, cpu):
    try:
        f = open(filename, "w")
```

```
  except IOError:
    sys.stderr.write("Writing checkpoint file failed. \n")

  f.write(str(ntasks)+" "+str(cpu)+"\n")
  f.close()

#def wait(n):
#   tic = time.time()
#   toc = time.time()
#   while toc - tic < n:
#      time.sleep(1)
#      toc = time.time()


#--------------------------------------------------------------------------
# MAIN
#--------------------------------------------------------------------------

# BOINC OPTIONS (bools) :
main_program = 1
check_heartbeat = 1
handle_process_control = 1
send_status_msgs = 1
handle_trickle_ups = 0
handle_trickle_downs = 0

retval = boinc_init_options(main_program, check_heartbeat, handle_process_control,
                        send_status_msgs, handle_trickle_ups, handle_trickle_downs)

if (retval):
       sys.exit(retval)

# read command line
commandline = ""
for arg in sys.argv[1:len(sys.argv)]:
  commandline = commandline + arg + " "

# read job file
[tasks, VMmanageTasks, unzip_tasks] = read_job_file(JOB_FILENAME)

# read checkpoint file
ntasks = 0
[ntasks, checkpoint_cpu_time] = read_checkpoint(CHECKPOINT_FILENAME)
if ntasks > len(tasks):
  sys.stderr.write("Checkpoint file: ntasks "+str(ntasks)+" too large\n")
  boinc_finish(1)
if ntasks == len(tasks):
  sys.stderr.write("Workunit is already computed.\n")
  boinc_finish(0)


# calculate the total weight
total_weight = 0
for task in tasks:
  total_weight = total_weight + task.weight
```

```
sys.stderr.write('ntasks: '+str(ntasks)+" len tasks: "+str(len(tasks))+"\n")

# check if there is a task which uses VM
isVM = 0
for i in range(ntasks, len(tasks)):
  if tasks[i].virtualmachine != "":
    isVM = 1
    break

# unzip the archives first
for task in unzip_tasks:
  task.run()

for task in unzip_tasks:
  task.poll()
  tic = time.time()
  while not task.ready:
    if time.time() - tic > MAX_WAIT_TIME:
      sys.stderr.write("Unzip task "+task.application + " "+task.command_line+" take too long. \n")
      task.kill()
      boinc_finish(5)

    sys.stderr.write("Wait for unzip tasks. \n")
    time.sleep(2)
    task.poll()

# try-finally structure makes sure that we kill the VMmanage tasks also when
# there is an error.
exitStatus = 0
try:

  # If there is a task which uses VM we start VMmanageTasks
  if isVM:
    for task in VMmanageTasks:
      task.run()

    VM_MANAGE = xmlrpclib.ServerProxy(SERVER_PROXY)

    # wait until VMmanageTasks started succesfully
    tic = time.time()
    time.sleep(5)
    VMmanageRunning = 0
    while 1:

      # check that tasks are still running
      for task in VMmanageTasks:
        task.poll()
        if task.ready:
          sys.stderr.write("VM manage tasks "+task.application+" not running. Exit status:
"+str(task.exitCode)+"\n")
          raise Exception(5)

      try:
        VM_MANAGE.listAvailableVMs()
```

```
      except Exception as e:
        sys.stderr.write("Wait VM manage tasks to start. \n")
        traceback.print_exc()
        if time.time() - tic > MAX_WAIT_TIME:
          sys.stderr.write(str(e)+"\n")
          sys.stderr.write("It took too long to VM manage tasks to start. \n")
          raise Exception(5)
        time.sleep(5)
        continue

      sys.stderr.write("VM manage tasks started successfully \n")
      VMmanageRunning = 1
      break
else:
  VM_MANAGE = []

weight_done = 0
for i in range(ntasks, len(tasks)):
    weight_done = weight_done + tasks[i].weight
    frac_done = weight_done / total_weight
    tasks[i].starting_time = checkpoint_cpu_time

    sys.stderr.write('task number: '+str(i)+"\n")

    if tasks[i].virtualmachine != "":
      # check VMmanageTasks process
      #for task in VMmanageTasks:
      #   [ready, status] = task.poll()
      #   if ready:
      #      sys.stderr.write("App '"+task.application+"' is not running. Exit status: "+str(status)+"\n")
      #      boinc_finish(195); # EXIT_CHILD_FAILED

      # run app in VM (JOB_FILENAME used as a connection test file)
      sys.stderr.write('run virtual machine: '"+tasks[i].virtualmachine+"'\n')
      tasks[i].runVM(VM_MANAGE, commandline, MAX_WAIT_TIME)
    else:
      # run app in host
      sys.stderr.write('run application on Host: '"+tasks[i].application+"'\n')
      tasks[i].run(commandline)

    # wait for the task to accomplish
    while 1:
      tasks[i].poll(VM_MANAGE)

      if tasks[i].ready:
        if tasks[i].exitCode:
          sys.stderr.write("App exit code "+str(task.exitCode)+" \n")
          sys.stderr.write("App stderr: \n")
          sys.stderr.write(tasks[i].CmdResults['err']+"\n")
          raise Exception(195) # EXIT_CHILD_FAILED
        break

      if poll_boinc_messages(tasks[i], VM_MANAGE):
        # VMwrapper should exit cleanly
        raise Exception(0)
```

```
        if task.has_checkpointed(VM_MANAGE, CHECKPOINT_PERIOD):
          checkpoint_cpu_time = tasks[i].starting_cpu + tasks[i].cpu_time(VM_MANAGE)
          write_checkpoint(CHECKPOINT_FILENAME, i, checkpoint_cpu_time)

        send_status_message(tasks[i], VM_MANAGE, frac_done, checkpoint_cpu_time)
        time.sleep(POLL_PERIOD)

      if tasks[i].virtualmachine != "":

        # write stdout and stderr
        if tasks[i].stdout_filename != "":
          fpath = boinc_resolve_filename(tasks[i].stdout_filename)
          fout = open(fpath, "w")
          fout.write(tasks[i].CmdResults['out'])

        if tasks[i].stderr_filename != "":
          fpath = boinc_resolve_filename(tasks[i].stderr_filename)
          ferr = open(fpath, "w")
          ferr.write(tasks[i].CmdResults['err'])
        else:
          sys.stderr.write(tasks[i].CmdResults['err'])

        # if we are going to copy files to VM we need to try that VM has connected to
        # broker successfully
        if len(tasks[i].copy_files_from_VM) > 0:
          tic = time.time()
          while 1:
            try:
              VM_MANAGE.ping(tasks[i].virtualmachine)
            except Exception as e:
              sys.stderr.write("waiting for VM to connect to the broker \n")
              sys.stderr.write(str(e)+"\n")
              if time.time() - tic > MAX_WAIT_TIME:
                sys.stderr.write(str(e)+"\n")
                sys.stderr.write("It took too long to VM to connect to the broker. \n")
                raise Exception(5)

              time.sleep(10)
              continue
            break

        # copy files from VM
        for fileName in tasks[i].copy_files_from_VM:
          file_path = fileName # is it always true?
          sys.stderr.write("Copy file from VM: "+file_path+"\n")
          [out, err, status] = VM_MANAGE.cpFileFromVM(tasks[i].virtualmachine, file_path,
fileName)
          if status:
            sys.stderr.write("Can't copy file from VM \n")
            sys.stderr.write(err)
            raise Exception(4)

        # save state of VM and kill it (VMMain.py is running on VM!)
        VM_MANAGE.saveState(tasks[i].virtualmachine) # or do we want to power off??
```

```
      checkpoint_cpu_time = tasks[i].starting_cpu + tasks[i].final_cpu_time
      write_checkpoint(CHECKPOINT_FILENAME, i+1, checkpoint_cpu_time)

# read the exit status from raised exception
except Exception as e:
  sys.stderr.write(str(e)+"\n")
  traceback.print_exc()
  if type(e.args[0]) == int:
    exitStatus = e.args[0]

finally:
  if isVM:
    # save state of running VMs
    if VMmanageRunning: # otherwise listRunningVMs() never exits
      for task in tasks:
        if task.virtualmachine != "":
          try:
            for VM in VM_MANAGE.listRunningVMs():
              if VM == task.virtualmachine:
                VM_MANAGE.saveState(task.virtualmachine)
          except:
            sys.stderr.write("Couldn't save state of "+task.virtualmachine+". \n")
            traceback.print_exc()

    # kill VM manage tasks
    for task in VMmanageTasks:
      sys.stderr.write("Kill the VM manage task: "+task.application+" \n")
      task.kill(VM_MANAGE)

  boinc_finish(exitStatus)
```