SAGA Tutorial Exercise

Ole Weidner, Hartmut Kaiser, Andre Merzky, Shantenu Jha

22 Nov 2010

1 Scope of this Tutorial

The scope of this tutorial is to provide the audience with the required resources and technical knowledge (hands-on experience) to start hacking their own distributed applications with SAGA. We will begin with a quick overview of how to use SAGA – the basic, and then quickly dive into simple yet complete applications written using SAGA. In the latter half we will discuss a couple of "complex" applications, which aren't complex at all, but would have been if they hadn't been written in SAGA.

The source code for all the examples in this tutorials are part of the *saga-core* source tree and can be found at:

`http://svn.cct.lsu.edu/repos/saga/core/trunk/examples/tutorial`

1.1 Prerequisites

This tutorial requires basic knowledge of the C/C++ programming language. Experience using the command line on a Linux/UNIX based operating system and a basic idea of what a compiler, a linker and a Makefile is might come in handy.

Unless this tutorial is going to be preceded by a SAGA installation tutorial, the students are required to have a fully working installation of SAGA on

their laptops/lab machines (preferred), or remote access (e.g. via SSH) to a machine with SAGA installed.

1.2 SAGA Applications

Every application that uses even a single SAGA call is considered a SAGA application, since it triggers the whole (SAGA) stack. SAGA is not a framework, but it provides the building blocks from which to develop frameworks and/or applications. Loosely speaking, SAGA is programming system for distributed applications; it is important to understand that it does not impose a specific programming model. Remember the rough taxonomy that we presented of three ways of developing distributed applications using SAGA: (i) Implement a distributed submission/execution mode for a *legacy* application; (ii) Create a framework that supports a specific application characteristics and/or pattern, or (iii) Compose applications from multiple (distributed) units, which in a way makes it an *a priori* application.

Unit I: Command Line Utilities

The following is a simple example of an application that copies a file. There are more such simple (command-line utilities) that we will discuss at:
`https://svn.cct.lsu.edu/repos/saga/core/trunk/tools/clutils/`

```
#include <saga/saga.hpp>

int main(int argc, char * argv[])
{
  saga::url source("ssh://hostname//etc/passwd");
  saga::url target(".");

  saga::filesystem::file file (source, saga::filesystem::Read);
  file.copy(target);

  return 0;
}
```

Q: Which of the three types of distributed applications would you classify the above copy program into?

Unit II

Compiling and linking. Like with any other C/C++ library, you have to let the compiler and the linker know where to find the header files and the library. To make life easier, SAGA provides a Makefile which you can include in order to build your application:

```
SAGA_SRC          = $(wildcard *.cpp)
SAGA_ADD_BIN_OBJ  = $(SAGA_SRC:%.cpp=%.o)
SAGA_BIN          = my_saga_app

include $(SAGA_LOCATION)/share/saga/make/saga.application.mk

## Other (optional) compiler and linker flags
SAGA_CPPFLAGS    += -I/opt/super/include
SAGA_LDFLAGS     += -L/opt/super/lib -lsuper
```

Of course it is also possible to compile and link a SAGA application manually:

```
g++ -Wall -I$SAGA_LOCATION/include -pthread \
    -L$SAGA_LOCATION/lib \
    -lsaga_engine -lsaga_package_job -lsaga_package_XYZ \
    <FILENAME>.cpp
```

Unit III: Running a SAGA application.

SAGA needs to know where its configuration files are located and where to find its middleware adaptors. This is done via the `SAGA_LOCATION` environment variable, e.g.:

```
export SAGA_LOCATION=/opt/saga-1.5.3-pre/
```

In order to run a SAGA application, you have to make sure that all required libraries can be found by the loader in case SAGA is not installed within the default system path. The easiest way to do that is to set the LD_LIBRARY_PATH (DYLD_LIBRARY_PATH on Mac OS), e.g.:

```
export LD_LIBRARY_PATH=$SAGA_LOCATION/lib:$LD_LIBRARY_PATH
```

Another (optional) environment variable that might come in handy is `SAGA_VERBOSE` in case something goes wrong. If set, SAGA will print detailed debug information to a log-file in the working directory, e.g.:

```
export SAGA_VERBOSE=100
```

Unit IV: SAGA-based Applications:

You have had some initial exposure to the API when going through the command-line utilities. In this section, we will work with three different examples. The aim of these applications is to give you a quick feel for how SAGA is actually utilized to develop *complete, stand-alone* distributed applications. And although these examples are by necessity very simple, they are representative of the way you would use SAGA in actual real-world examples to develop many of the scientific applications.

In the first example, we will introduce a simple "Hello Distributed World!", where the aim will be to submit three simple remote jobs using SAGA. In the second example application ("chaining_jobs.cpp"), we will serialize the launch of three (remote) jobs, so that the second job is launched after the first, and the third job is launched after the second. In the third example application ("depending_jobs.cpp"), we will start an application that once started, is able to re-spawn itself on another machine, and after doing so increments a "global counter". Finally, we will leave you with a programming exercise that will build upon your understanding of application examples 2 and 3.

Example 1: Hello distributed world!

Submit three jobs to three machines. One returns Hello, one returns Distributed and one returns World. They may or may not return in the right order. This should give you an an idea how they could potentially speed up their application using multiple resources. Also, execute the program several times. Do you notice any difference in the outputs? Are they same?

```
// The hello_world example is meant to be a very simple and
// first example to try when it comes to SAGA. It's purpose
// is to spawn 3 (possibly remote) identical jobs (/bin/echo)
// while passing the 3 words "Hello", distributed", and "world!"
// on their command lines.
//
// The result  is that the jobs will print the respective command
// line arguments (it's /bin/echo after all). The master job (this one)
// waits for the 3 child jobs to  finish. It intercepts the generated
// output and prints it to the user. Depending on which child jobs
// finish first the overall printed message might be some combination
// of the 3 arguments we passed. But most of the time you
// will see "Hello distributed world!", which is our way of saying
// hello and welcome to the world of SAGA.

// the routine spawning the SAGA jobs and waiting for their results
void run_a_job(std::string host, std::string argument)
{
    try {
        saga::job::service js (host);
        saga::job::ostream in;
        saga::job::istream out;
        saga::job::istream err;

        // run the job
        saga::job::job j = js.run_job("/bin/echo " + \\
                                      argument, host, in, out, err);

        // wait for the job to finish
        saga::job::state s = j.get_state();
```

```
        while (s != saga::job::Done && s != saga::job::Failed)
            s = j.get_state();

        // if the job finished successfully, print the generated output
        if (s == saga::job::Done) {
            std::string line;
            while (!std::getline(out, line).eof())
                std::cout << line << '\n';
        }
        else {
            std::cerr << "SAGA job: " << j.get_job_id() << " failed (state: "
                      << saga::job::detail::get_state_name(s) << ")\n";
        }
    }

   catch (saga::exception const& e) {
        std::cerr << "saga::exception caught: " << e.what () << std::endl;
    }
    catch (std::exception const& e) {
        std::cerr << "std::exception caught: " << e.what () << std::endl;
    }
    catch (...) {
        std::cerr << "unexpected exception caught" << std::endl;
    }
}

///////////////////////////////////////////////////////////////////////////////
int main(int argc, char* argv[])
{
    // run 3 separate threads executing the saga calls
    boost::thread t1 (run_a_job, HOST1, "Hello");
    boost::thread t2 (run_a_job, HOST2, "distributed");
    boost::thread t3 (run_a_job, HOST3, "world!");

    // wait for all spawned threads to finish
    t1.join();
    t2.join();
    t3.join();
```

```
    return 0;
}
```

Example 2: Multiple Sequential Jobs

Applications The aim of this section is to see how SAGA is used to implement common *higher-level* functionality that is used by Distributed Applications (DA). Specifically, we will look at two commonly occurring functionality required by DA.

Example 1: Here we will demonstrate the ability to checkpoint, use a specified resource, self-migrate and restart on a different computational resource.

Here we will demonstrate this capability using the **Hello distributed world!** example discussed in Unit IV. Instead of launching three jobs on three machines, we will launch one job on one machine, which will then launch itself on another machine, which in turn will do so onto yet another machine.

```
////////////////////////////////////////////////////////////////
// The chaining_jobs example tries to overcome one of the limitations
// of the   hello_world example: it introduces dependencies between 3
// (possibly remotely) spawned childs. In this example the next child
// will be spawned only after the previous one has finished its execution.
// Here we use an "addition" hack to do some calculations,
// where the result of the previous calculation is used as
// the input for the next one.
//
// Try to make more complex calculations if you like/can
////////////////////////////////////////////////////////////////

////////////////////////////////////////////////////////////////////
// the routine spawning the SAGA jobs and waiting for their results
std::string increment(std::string host, std::string argument)
{
    try {
        saga::job::service js (host);
        saga::job::ostream in;
        saga::job::istream out;
        saga::job::istream err;

        // run the job
```

```
        saga::job::job j = js.run_job("/usr/bin/bc -q", host, in, out, err);

        // wait for the job to finish
        saga::job::state s = j.get_state();
        while (s != saga::job::Running && s != saga::job::Failed)
            s = j.get_state();

        // if the job didn't start successfully, print error message
        if (s == saga::job::Failed) {
            std::cerr << "SAGA job: " << j.get_job_id() << " failed (state: "
                      << saga::job::detail::get_state_name(s) << ")\n";
            return argument;
        }

        // feed the remote process some input
        in << "1 + " + argument + "\n";

        // receive result
        std::string line;
        std::getline(out, line);

        // quit remote process
        in << "quit\n";

        return line;
    }
    catch (saga::exception const& e) {
        std::cerr << "saga::exception caught: " << e.what () << std::endl;
    }
    catch (std::exception const& e) {
        std::cerr << "std::exception caught: " << e.what () << std::endl;
    }
    catch (...) {
        std::cerr << "unexpected exception caught" << std::endl;
    }
    return argument;    // by default just return argument
}
```

```
///////////////////////////////////////////////////////////////////
int main(int argc, char* argv[])
{
    // run 3 separate threads executing the saga calls
    std::string result = increment(HOST1, "1");
    result = increment(HOST2, result);
    result = increment(HOST3, result);

    std::cout << "The overall result is: " << result << std::endl;

    return 0;
}
```

Once developed, this capability can be utilized by a wide range of different applications. In other words this capability described/shown above is independent of any specific application logic. Do you know of a (Scientific) application that could utilize this feature?

Example 3: Managing Dependencies between Jobs

In this example, we will introduce the *advert service* as a simple mechanism to provide coordination between different distributed tasks. Specifically, the advert service will be used by a set of jobs to increment a global counter everytime a job is successfully spawned. There are other ways of *coordinating* distributed tasks/jobs, but the idea of a centralized data-store is arguably the simplest, even if not the most robust (fault-tolerant) or tuned for performance. Also, of interest is the `respawn` method.

```
///////////////////////////////////////////////////////////////////////
// Start this example by providing an arbitrary number of hosts on the
// command line. It will re-spawn itself on each of the hosts. Each
// instance will increment a number stored in a central result store.
///////////////////////////////////////////////////////////////////////


///////////////////////////////////////////////////////////////////////
// the routine spawning the SAGA jobs and waiting for their results
void respawn(int argc, char *argv[])
{
    assert(argc > 1);     // we shouldn't end up here without any given hosts
    try {
        saga::job::service js (argv[1]);

        // compose the command line, skip first argument
        std::string commandline (JOB_PATH);
        for (int i = 2; i < argc; ++i) {
            commandline += " ";
            commandline += argv[i];
        }

        // run the job on host given by first argument
        saga::job::job j = js.run_job(commandline, argv[1]);

        // wait for the job to start
        saga::job::state s = j.get_state();
        while (s != saga::job::Running && s != saga::job::Failed)
            s = j.get_state();
```

```
        // if the job didn't start successfully, print error message
        if (s == saga::job::Failed) {
            std::cerr << "SAGA job: " << j.get_job_id() << " failed (state: "
                      << saga::job::detail::get_state_name(s) << ")\n";
        }

        // wait for the job to Finish
        s = j.get_state();
        while (s == saga::job::Running)
            s = j.get_state();
    }

    catch (saga::exception const& e) {
        std::cerr << "saga::exception caught: " << e.what () << std::endl;
    }
    catch (std::exception const& e) {
        std::cerr << "std::exception caught: " << e.what () << std::endl;
    }
    catch (...) {
        std::cerr << "unexpected exception caught" << std::endl;
    }
}
```

Not surprisingly the code snippet above is independent of any application specific details and focuses on the assignment of workloads to workers, execution and then retrieval. This specific approach adopted here relies heavily on the use of the Advert Service.

```
////////////////////////////////////////////////////////////////////
int main(int argc, char* argv[])
{
    if (argc == 1) {
    // no more hosts are given, we're done!
        int result = 0;
        if (get_result(result))
            std::cout << "The overall result is: " << result << std::endl;
```

```
    }
    else {
    // otherwise get current value, increment it, and store new value
        int result = 0;
        get_result(result);   // ignore errors, will set result to zero

        // re-spawn this job, increment result
        if (set_result(result + 1))
            respawn(argc, argv);
    }
    return 0;
}
```

Q: Can you think of an a usage-mode that can be supported by the general functionality to respawn a job? Long-running Simulations? What else?

Additional Real World Example

SAGA can be used to implement simple Master-Worker applications. See:

```
https://svn.cct.lsu.edu/repos/sci-comp/public/Module-E/7700-E3.pptx
```

MapReduce and Mandelbrot Set Calculation are bust just two simple examples. In both of these, the fundamental idea is that there is a Master which coordinates the distribution of work to a large number of Workers, and manages the merging of the output of the computation that the Workers produce. In addition to performance, a fundamental challenge is the need to be able to coordinate Master-Workers across a wide range of distributed systems. For more details, check out the code at:

```
https://svn.cct.lsu.edu/repos/saga-projects/applications/
   MapReduce/branches/MapReduce.2009
https://svn.cct.lsu.edu/repos/saga-projects/applications/
   MapReduce/branches/MapReduce.2009/docs
```

A wordcount example that uses SAGA MapReduce can be found at:

```
https://svn.cct.lsu.edu/repos/saga-projects/applications/
   MapReduce/branches/MapReduce.2009/examples/wordcount
```

A SAGA based implementation of the Mandelbrot Set computation can be found at:

```
https://svn.cct.lsu.edu/repos/saga-projects/applications/mandelbrot
```

1.3 Conclusion

Let us recap that there are multiple types of distributed applications. What you have seen here are simple applications that utilize distributed functionality, such as remote job submission to achieve tasks. What you should take away from this tutorial are essentially the following:

- Distributed Applications can be developed much like regular applications. The challenges facing Distributed Applications – development and deployment are different from traditional applications and many of those challenges arise from the distributed infrastructure. It is to precisely meet these "unique" distributed computing challenges that there is a need for simple, standard and pervasive application level interface such as SAGA was conceived.

- We have focused on some of the challenges of developing distributed applications, such as coordinating distributed tasks. We have shown the ability to do so in a simple fashion; however this is not necessarily scalable, and poses challenges for many real-world applications. Some other real-world challenges we have not discussed here are fault-tolerance, recovery, replication etc. SAGA provides APIs to these "Advanced" features as well.

- Interestingly, we have built all the distributed functionality around simple "ssh" adaptors; if you wanted to launch to a Globus or a Condor specific infrastructure, you would just configure SAGA to utilize Globus or Condor specific adaptors.

- Remember the following website `http://saga.cct.lsu.edu` is your source of information for all things SAGA. And this document can be found at:

  ```
  https://svn.cct.lsu.edu/repos/saga-projects/tutorial/
          general_tutorial/saga_tutorial_example.tex
  ```

1.4 Programming Exercise:

1.4.1 Exercise 1:

Recall how when the program hello_world was executed the order of the values returned often varied. Can you use introduce dependencies between the jobs so as to ensure that the output is always in order "Hello Distributed World!"?

1.4.2 Exercise 2:

In earlier examples, we introduced the underlying concepts of submitting jobs and coordination amongst multiple distributed jobs (tasks), where we updated the value of a counter. Can the same approach (i.e. advert) be used to coordinate the submission of multiple jobs? In effect, this is a way of informing a job (that is ready to spawn another job) about which (possible) machines to spawn too. The aim of this exercise is for you to complete the code by implementing some (i) job submission functionality, and (ii) accessing advert entries. (We will post a sample solution to this at the end of the tutorial).

```
<CODE>
```

Think of generalizations to this concept: Say one application is "producing" this information (that is a list of possible resources), and this information is being "consumed" by another application.