CppSs – a C++ Library for Efficient Task Parallelism

Steffen Brinkmann and José Gracia
High Performance Computing Centre Stuttgart (HLRS)
University of Stuttgart
70550 Stuttgart, Germany
E-mail: {brinkmann,gracia}@hlrs.de

***Abstract*—We present the C++ library CppSs (C++ superscalar), which provides efficient task-parallelism without the need for special compilers or other software. Any C++ compiler that supports C++11 is sufficient. CppSs features different directionality clauses for defining data dependencies. While the variable argument lists of the taskified functions are evaluated at compile time, the resulting task dependencies are fixed by the runtime value of the arguments and are thus analysed at runtime. With CppSs, we provide task-parallelism using merely native C++.**

Keywords–high-performance computing; task parallelism; parallel libraries.

I. INTRODUCTION

Programming models implementing task-parallelism play a major role when preparing code for modern architectures with many cores per node and thousands of nodes per cluster. In high performance computing, a common approach for achieving the best parallel performance is to apply the message passing interface (MPI) [1] for inter-node communication and a shared-memory programming model for intra-node parallelisation. This way, the communication overhead of pure MPI applications can be overcome.

Shared memory models are also crucial when using single node computers as there are systems consisting of hundreds or even thousands of processing units accessing the same memory address space. These systems offer great parallelism to the developer. But utilising the processing units evenly, so that they can run efficiently, is a non-trivial task.

Many scientific applications are based on processing large amounts of data. Usually, the processing of this data can be split up and some of these chunks have to be executed in a well defined order while others are independent. This is the level on which task based programming models are employed. We will call the chunks of work to be processed tasks, while the appearances in the code (e.g., if they are implemented as functions, methods or subroutines) are going to be called task instances.

The dependencies between tasks can be stated explicitly by the programmer or inferred automatically by some kind of preprocessing of the code. In the case of fork-join-models (e.g. OpenMP [2]), all tasks after a "fork" are (potentially) parallel while code after the "join" and all consecutive forks depend on them. For example, in figure 1a), tasks 2, 3 and 4 can run in parallel, if sufficient processing units are available. Task 5 cannot be executed before all other tasks have finished. In programming models which support nesting (e.g. Cilk [3]), the dependencies can sometimes be derived from the placement of the calls (see Figure 1b)).

In many implementations of task based programming models, the data dependencies are specified explicitly by the programmer (e.g. SMPSs [4], OMPSs [5], StarPU [6] and XKAAPI [7]). This allows for more complex dependency graphs and therefore more possibilities to adjust the parallelisation to the code, the amount of data and the architecture. However, these implementations suffer from a number of disadvantages:

- The tasks and/or task instances and their dependencies have to be marked by special directives, usually within a `#pragma` in C or using special comments in Fortran. These use keywords and syntax which is not part of the actual language and which the programmer needs to learn.
- In order to compile the instrumented code, the programmer needs a special compiler or preprocessor. She depends on this additional software to be available on the desired platform, which is not generally the case.
- The need for special compilers also poses additional work to system administrators who will be asked by the programmer to install the specific compiler used in the application.
- The code of the programming model implementation itself becomes more difficult to maintain and usually at least one additional compile step is introduced when compiling the user code.

In order to avoid these inconveniences, we developed a pure C/C++ library, which allows functions to be marked as tasks and to execute them asynchronously. The programmer still needs to prepare the code looking for the parts feasible for parallelisation and separate them into functions. Also, it is still necessary to instrument the code with the CppSs API. But contrary to the implementations mentioned above, this is achieved using standard C++11 syntax instead of an "imposed" pragma language.

To execute the application serially, e.g. for debugging, the programmer can define the macro `NO_CPPSS`, which bypasses the creation of additional threads and converts the tasks instances into normal function calls.

In the following, we will illustrate the usage (Section II) and present the basic implementation of the library CppSs (Section III). Lastly, we will sum up our conclusions in Section IV.

 ISBN: 978-1-61208-310-0

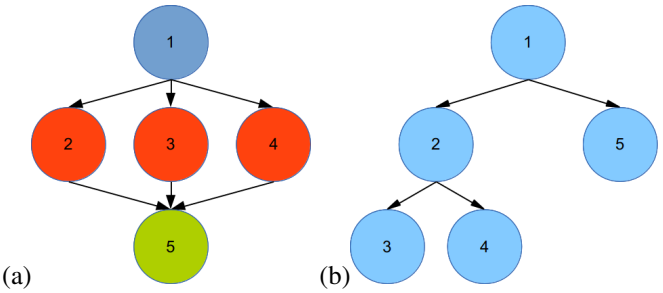

Fig. 1. (a) Example of fork-join-parallelism. After task 1 the execution thread is forked. (b) Example of nested parallelism. Task 1 spawns tasks 2 and 5. Before task 5 is created, task 3 and 4 are spawned, hence the numbering.

```
void func1(int *a1, double *a2, double *b)
{
        //...
}
auto func1_task = CppSs::MakeTask(func1,
                                  {INOUT,IN,OUT},
                                  "func1");
```

Fig. 2. Defining and taskifying a function. The return value is a functor, i.e., an object which overloads the parenthesis operator. Hence it can be "called" like a function.

II. CppSs - usage

CppSs is a library which compiles on any system with a working C++ compiler. The C++11 features necessary for CppSs are provided by the GNU compiler of version 4.6 or higher and the Intel compiler of version 13 or higher.

In order to use CppSs, the programmer only needs to include the header `CppSs.h` and link against the library `libcppss.so`. All of CppSs' application programming interface (API) functions are declared in the namespace `CppSs` to avoid overlap with other libraries' functions. In the following, the CppSs API is introduced presenting the declaration of tasks (Section II-A), the initialisation and finishing of the parallel execution (Section II-B) and setting barriers (Section II-C). Finally, we will give a minimal example putting everything together in Section II-D.

A. Declaring Tasks

Parallelisation with CppSs relies on functions with well defined directionality of their parameters. Loop parallelisation and anonymous code blocks are not supported.

To convert a function into a task, the programmer has to call the API function `MakeTask`, which takes the following parameters (see listing in Figure 2):

- a pointer to the function,
- an initialiser list containing directionality specifiers for each function parameter,
- (optional) a string with the function name for debugging purposes and
- (optional) a priority level, which is ignored in the present version. Future versions will provide one or more priority queues.

```
auto func_task = CppSs::MakeTask(func,
                                 {INOUT,IN,OUT},
                                 "func");
auto func_task = CPPSS_TASK(func,
                            {INOUT,IN,OUT});
CPPSS_TASKIFY(func,{INOUT,IN,OUT})
```

Fig. 3. Convenience macros for task declaration. These three lines translate into the same binary code. Hence only one of them should be used.

It is required that the arguments of the taskified function which are intended to cause dependencies are pointers. These can be used to access arrays, built-in types or any other data structure. However, potential overlap with other data structures is not detected. The directionality specifier must be one of `IN`, `OUT`, `INOUT`, `REDUCTION` or `PARAMETER`. The latter is used for arguments which are not to be interpreted as a potential dependency and must be of a built-in numerical type. The effect of each of the directionality specifiers are described in the following:

a) IN: The task treats this argument as input. It will not be executed until all task instantiations which were called before the function and which write to this argument (i.e. have an `OUT`, `INOUT` or `REDUCTION` specifier for the same argument value) have finished.

b) OUT: The task treats this argument as output. The content of the variable or array pointed to is (possibly) overwritten. This affects functions with an `IN` or `INOUT` specifier for the same argument value.

c) INOUT: The task intends to read from and write to this argument value. It will be dependent on the last task writing to this memory address. The following tasks reading from this memory address will be dependent on this task.

d) REDUCTION: Similar to `INOUT`. The task intends to read from and write to this argument value. In contrast to `INOUT`, the tasks with a `REDUCTION` clause will depend on other tasks with a `REDUCTION` clause on the same argument value.

e) PARAMETER: The argument is treated as a parameter. It will be ignored for the dependency analysis.

The return type of `MakeTask` is an internal template type, which includes the argument types of the taskified function, thus we recommend to use the C++11 keyword `auto`.

For convenience two macros were defined that wrap the call to `MakeTask`. The three calls in Figure 3 are equivalent.

B. Init and Finish

The next instrumentation to be inserted in the application code is calls to `Init` and `Finish`. These calls must be called before and after each task, respectively. While `Finish` takes no arguments, `Init` takes two optional arguments, namely

- the number of threads and
- the reporting level.

The number of threads must be any positive integer. If none is given, the default is 2. The reporting level must be

 ISBN: 978-1-61208-310-0

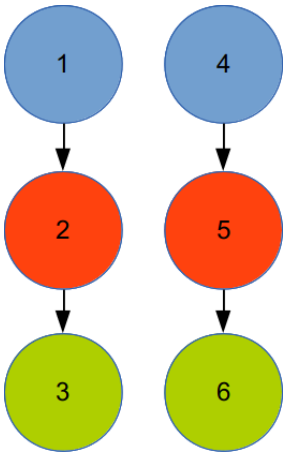

Fig. 4. Task dependency graph of the minimal example in listing in Figure 5. The blue nodes (1 and 4) represent task function `set_task`, the red nodes (2 and 5) `increment_task` and the green nodes (3 and 6) `output_task`.

one of `ERROR`, `WARNING`, `INFO` or `DEBUG`, which causes increasing amount of output. The default is `WARNING`.

`Init` will instantiate a runtime system which enables the queuing and asynchronous execution of tasks. The runtime will create one thread less than the number of threads specified in the call to `Init` as the main thread will also execute tasks. The threads will be constructed using the standard library `std::thread` class. This way portability is granted for each system which provides a C++11 compiler.

`Finish` will wait for all the tasks to be finished and destruct all threads, queues and the runtime.

C. Barriers

With the API function `Barrier` it is possible to halt the main execution thread, i.e. the code outside of tasks, until all tasks instantiated so far have finished. The call takes no arguments. The call to `Finish` contains a call to `Barrier`.

D. Minimal example

To sum up the API usage, we compile everything into a small example, shown in Figure 5. Internally, it produces the dependency graph shown in Figure 4 and prints the output shown in Figure 6.

III. CppSs - Implementation with variadic templates

The major design paradigm for CppSs was to avoid usage of external libraries. All code should be compilable with a standard C++ compiler. In order to achieve this goal, several features of C++11 were used, the most prominent one being variadic templates [8]. These are of central importance as the objects representing a task and an instance of a task are implemented as variadic templates, the function arguments of the taskified function being the template arguments. This is necessary because a function which the application programmer wants to taskify can have any number and type of arguments. These arguments are known at compile time, so an

```
#include <iostream>
#include "CppSs.h"

#define N_THREADS 2

void set(int *a, int b)
{
    (*a) = b;
}
CPPSS_TASKIFY(set,{OUT,PARAMETER})

void increment(int *a)
{
    ++(*a);
}
CPPSS_TASKIFY(increment,{INOUT})

void output(int *a)
{
    std::cout << (*a) << std::endl;
}
CPPSS_TASKIFY(output,{IN})

int main(void)
{
    int a[] = {1,11};

    CppSs::Init(N_THREADS,INFO);

    for (unsigned i=0; i < 2; ++i){
        set_task(&(a[i]), i);
        increment_task(&(a[0]));
        output_task(&(a[0]));
    }

    CppSs::Finish();

    return 0;
}
```

Fig. 5. Minimal complete example for CppSs. This code will produce a dependency graph as shown in Figure 4. The output will be similar to listing in Figure 6.

```
- 13:32:45.207 INFO:  ### CppSs::Init ###
- 13:32:45.207 INFO: adding worker: 1 of 2
- 13:32:45.207 INFO: Running on 2 threads.
1
2
- 13:32:45.207 INFO: Executed 6 tasks.
- 13:32:45.207 INFO:  ### CppSs::Finish ###
```

Fig. 6. Output from minimal example from listing in Figure 5.

implementation with variadic templates is the most efficient way to handle variable argument lists.

An excerpt of the `Task_functor` class declaration which stores the taskified function is shown in Figure 7.

In order to process the variable argument list at compile time, recursive template evaluation is necessary. For instance, the set of template functions used to retrieve the types of the task function arguments is shown in Figure 8.

IV. Conclusion

We developed a pure C/C++ library, which allows functions in C/C++ source code to be marked as tasks, specify their dependencies and to execute them asynchronously. Contrary

 ISBN: 978-1-61208-310-0

```
template<typename... ARGS>
class Task_functor : public Task_functor_base
{
    //...
    void (*m_f) (ARGS...);
}
```

Fig. 7. Excerpt from the class declaration of `Task_functor` which stores the taskified function. The member declaration shows the pointer to the actual function with a variable argument list.

```
template <typename fun, size_t i>
struct get_types_helper {
    static void get_types(
       std::vector<std::type_info const*> &types) {
        get_types_helper<fun, i-1>::get_types(types);
        types.push_back(&typeid(typename
           function_traits<fun>::template arg<i-1>::type));
    }
};

template <typename fun>
struct get_types_helper<fun,0> {
    static void get_types(
       std::vector<std::type_info const*> &types) {}
};

template <typename fun>
void get_types(std::vector<std::type_info const*> &types) {
    get_types_helper<fun, function_traits<fun>::nargs>::\
       get_types(types);
}
```

Fig. 8. Template functions to process argument types at compile time. A call to `get_types<function>(types)` will recursively get the type of each of `function`'s arguments and place their type in the array `types`.

to other similar task based programming models like OpenMP, SMPSs or OMPSs, no preprocessor directives are necessary and the instrumented code will compile with any compiler, which supports C++11 features such as variadic templates, smart pointers and initializer lists. The smallest versions that qualify of the GNU compiler collection (gcc) and the Intel C compiler (icc), both of which are widely available, are gcc 4.6 and icc 13.

The current version is capable of constructing the task dependency graph and execute the tasks asynchronously. Several directionality clauses are available.

The code was checked for correctness but has still to prove scalability in realistic scenarios. First performance tests showed more than three times faster execution when running on four cores compared with the serial version of the same algorithm. We believe that these results can be enhanced by revising the implementation of the queueing and dequeueing as well as the creation and destruction of task functor instances.

ACKNOWLEDGMENT

The authors acknowledge support by the H4H project funded by the German Federal Ministry for Education and Research (grant number 01IS10036B) within the ITEA2 framework (grant number 09011).

REFERENCES

[1] Message Passing Interface Forum (http://www.mpi-forum.org/) [retrieved: 09, 2013]. [Online]. Available: http://www.mpi-forum.org/

[2] OpenMP (http://openmp.org/wp/) [retrieved: 09, 2013]. [Online]. Available: http://openmp.org/wp/

[3] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou, "Cilk: an efficient multithreaded runtime system," in *Proceedings of the fifth ACM SIGPLAN symposium on Principles and practice of parallel programming*, ser. PPOPP '95. New York, NY, USA: ACM, 1995, pp. 207–216. [Online]. Available: http://doi.acm.org/10.1145/209936.209958

[4] TEXT - Towards EXaflop applicaTions (http://www.project-text.eu) [retrieved: 09, 2013]. [Online]. Available: http://www.project-text.eu/

[5] The OmpSs Programming Model (http://pm.bsc.es/ompss) [retrieved: 09, 2013]. [Online]. Available: http://pm.bsc.es/ompss

[6] C. Augonnet, S. Thibault, R. Namyst, and P.-A. Wacrenier, *StarPU: A Unified Platform for Task Scheduling on Heterogeneous Multicore Architectures*, 2009.

[7] XKAAPI - Kernel for Adaptative, Asynchronous Parallel and Interactive programming (http://kaapi.gforge.inria.fr/) [retrieved: 09, 2013]. [Online]. Available: http://kaapi.gforge.inria.fr/

[8] D. Gregor and J. Järvi, "Variadic templates for C++0x," *Journal of Object Technology*, vol. 7, no. 2, p. 31–51, 02/2008 2008. [Online]. Available: http://www.jot.fm/issues/issue_2008_02/article2.pdf

 ISBN: 978-1-61208-310-0