Institut für Höchstleistungsrechnen

Projektarbeit

Execution of Task Parallel OpenMP Programs in Distributed Memory Environments - Scheduler

Markus Baur
2872854

Course of Study:	Simulation Technology
Examiner:	Prof. Dr.-Ing. Dr. h.c. Dr. h.c. Prof. E.h. Michael M. Resch
Supervisor:	Dr. Thomas Bönisch
Commenced:	September 1, 2018
Completed:	November 11, 2018

Contents

Listings

1 Introduction

1.1 Motivation

Writing software for only one node is become increasingly pointless, especially in high performance computing (HPC) applications, because processor clock speeds are increasing slower and slower and the number of cores per chip is also limited. Thus waiting a few years or adding cores to one node is not as rewarding as it used to be several years ago. In order to overcome this problem one can add more nodes. More nodes usually requires introducing MPI[1] to the program and eventually a redesign of the whole application. Furthermore it is not trivial to write a correct MPI program opposed to one which only uses OpenMP.

One idea to solve this problem is to take tasks from OpenMP programs and find a way to execute them on a different node. So the target of this work is to present a preprocessor, a runtime and a scheduler to transform shared memory OpenMP tasks to distributed memory tasks. The former two already exist so the focus was on finishing those and writing a scheduler.

This work is a follow up on the two bachelor theses of Johannes Erwerle, [Erw18], and myself, Markus Baur, [Bau18], where the target was to create a runtime and a preprocessor in order to distribute OpenMP tasks to several nodes. In order to bring it all together and to write a simple scheduler more work was needed, so the target of this paper is to finish it up and bring all pieces together.

The final deliveries of the previous work contained a preprocessor which was able to rewrite OpenMP tasks into a structure usable for offloading the work to another node, but the runtime lacked support for running arbitrary tasks as well as moving memory between nodes. Those two parts as well as the integration are the main targets of this work.

1.2 State of earlier work

1.2.1 Preprocessor state

The already existing preprocessor was in a somehow usable state, some types were not consistently applied throughout the project. On top of that the extraction and rewriting of the main function proved to be insufficient for integration with the runtime. There was one more problem with the

[1] MPI, the short name for Message Passing Interface, is a library and runtime to allow passing messages and memory between shared memory systems. More information can be found at [MPI18].

preprocessor, it can only be built within the clang[2] source tree, which made it difficult to change the code and fix bugs.

1.2.2 Runtime state

The runtime worked well with it's built in test cases, but it was difficult and error prone to add functionality. Furthermore it lacked support for moving memory between nodes, and, on top of the one task representation from the preprocessor it contained two more, one for the runtime node and one for the worker nodes respectively. It also used in tree builds with a make file which made it hard to tweak the build system for modularity and keep the source directory clean.

During the course of this work the runtime proved to be not extendable, at least by the author of this work, so it was rewritten from scratch but copying most algorithms and the layout from the existing runtime.

1.3 Targets

The targets for this work are outlined again below, some of the targets could not be met due to time constraints as well as complexity issues.

1.3.1 Scheduler targets

The runtime had no scheduler, the demonstration code executed one task and then exited. So there was a need for a scheduler to track the state of all tasks and decide which tasks are able to run and on which node they should execute. Some things which were mentioned here are dependency tracking for tasks the same way OpenMP does it and memory locality, which means scheduling tasks on nodes where the memory needed for them is already present. During the course of this work it was decided that turning the scheduler in a separate library would benefit the extendability of it, so this was added to the things wanted for the end result.

1.3.2 Runtime targets

The target for the runtime was to allow memory transfers across nodes and support for the scheduler from above. It should also link with the applications created from a preprocessed source in order to allow a ordinary C++ source file with OpenMP tasks to be preprocessed, compiled and ran with a as close as possible experience to a shared memory run of the same unprocessed source file.

[2]Clang is used to parse the C++ code, provide the lexer and other tools to work with C++ source code. Further information about clang can be found at [Cla18b].

2 Implementation

2.1 Build system

In order to properly build the complex integration from section 2.2 the build system for both existing parts had to be modified, the preprocessor could not be built outside of the clang source tree and the runtime used a Makefile for in source builds. This system was not capable of a such tight integration, so a switch to `cmake` was made in order to control the behaviour of the build system and allow for more modules than before.

2.1.1 Preprocessor build system

The build in the clang tree was not abandoned because it had some advantages, namely the headers were there and all the compiler macros which were needed to build it are present in the clang tree. Furthermore there are some clang versions with which the preprocessor refused to build. So a working clang version was added as a dependency to the source tree[1].

The source directory of the scheduler is linked to the correct place within the clang tree by `cmake` and a normal build of the whole clang tree is thus part of a normal build of the preprocessor. After the clang build finished the resulting application is copied to the output directory of the `cmake` build. The limitation here is that a reload of the `cmake` project requires a full rebuild of clang. This has to be addressed in future work because there were much more important issues at hand, e.g. writing the scheduler or the runtime, and it can be mitigated by using `ccache` or a other caching solution.

2.1.2 Scheduler build system

The scheduler is a component which should be easily changed depending on the system the code is about to execute. In order to achieve this form of modularity it is built into a separate shared library, even if the sources are in the same directory as the ones of the runtime. Currently this library is not path agnostic, so it is hard to find it when it is not in the path where it was built. This issue is shared with the runtime, but as mentioned earlier the build system was not the main target of this work, it was only needed in order to deal with the complexity.

[1] The version is the git commit with the hash 537ae129b767ac40785b17328ba1aaca7e5f5ace from the 31^{st} of October 2018 in the official clang mirror repository on github, [Cla18a].

2.1.3 Runtime build system

The runtime is built as a shared library with `cmake` and links together all the parts, it pulls in MPI and the scheduler library.

2.2 Runtime

The only two things which were planned to be done with the runtime were to add the memory transfer and writeback and the integration with the preprocessor and the scheduler. This soon proved not to work, after the implementation of arbitrary tasks the runtime did not work anymore. After all debugging efforts did not yield any tangible result and the original author of the code was unavailable, it was decided by the current author to write a new runtime. Many parts from the old runtime were incorporated in the new design to speed up the development process, especially the design was copied almost completely.

2.2.1 Worker nodes

Worker nodes execute tasks on request by the runtime. The memory required in order to run them is requested from the node on which the tasks was created and the difference is written back to this node. TODO

2.2.2 Runtime nodes

There are not much changes from the original runtime nodes, in short the runtime still handles task creation and distribution, the only new feature is the scheduler integration. On the down side the rewrite of the runtime showed that running more than one task per node, which would mean a capacity greater than `1` requires much more synchronization and locking so the capacity was set to `1` for each worker node for now. The implementation of taskwait was also not working and was scrapped in favour of memory transfer and write back.

2.2.3 Communication

The new runtime system uses one task class on all endpoints, so there is no type conversion between the runtime, the workers and the preprocessed program anymore. Some component of the task are not present on every endpoint though, some are hard to serialize, like the dependencies, others make no sense to synchronize, for example the thread in which the task is executing.

This communication in implemented using MPI and uses MPI tags to distinguish between different types of messages which then lead to different handlers being called by the receiving routine of the runtime or worker nodes. Every worker also has an associated runtime node where tasks are created and from which task runs are requested. Incoming tasks on the runtime are handed over to the scheduler which is also running within each runtime node.

2.3 Scheduler

Currently the scheduler is only a list of work items to do and due to the rewrite of the runtime not much work could be put into a sophisticated scheduler. The current algorithm searches for the worker with the most free capacity and schedules new tasks there. For the test cases[2] the algorithm uses all capacity of a single node test machine. One issue with the scheduler is, that dependencies are not taken into account because they are most likely evaluated, for example array indices are calculated and then applied. Such a dependency might look like this `depends(out: A[i][j])` where `i` and `j` are variables in the outer scope. So this is a point which might also be a starting of future work.

2.4 Integration

Due to the fact that the whole runtime system now uses one task representation the integration with the preprocessor was rather easy. The only change that was made were some adjustments to variable types and the handling of the main method, which was completely changed. Those variable types had to be changed because `void` pointers were used and indices of void pointers are not defined by the C++ standard.

The main function of the old program is now not amended but extracted into a function named `__main__1`. Furthermore there is a function called `__main__`, it is executed as a task and is another function in order to allow a proper freeing of the task resources even if the developer opted for a early return from his main function. This special first task is created by the runtime itself during the setup and then scheduled as soon as it is running.

2.5 Memory transfer

When a new task for execution arrives at a worker node, the first thing is to check it for the id of the node on which the task was created. If it is the same node as the one where the execution is about to take place only the pointer structure[3] needed for the extracted task function is created, otherwise a request for the memory is issued to the origin node which then responds with the number of variables and their sizes. Then they are transmitted in the same order as they were created during the preprocessor run. Upon receiving the memory a backup is made in order to only write back the changes. At last a array containing pointers to all transferred memory is created and then put into the created function for the task.

This all means that the memory captured by other tasks any task must not be freed, otherwise it would not be available for the created children tasks. This is a problem because taskwait instruction with which a task could wait for all its children are currently not supported.

[2] See section 3.1 for those.

[3] The extracted tasks use a single `size_t**` as input and there is some code emitted to cast all of this to the appropriate types used in the extracted task.

After a task finishes the changes are calculated using the backup made during the transmission and then the changes are sent to the origin node of the task. The algorithm to write back the changes currently requires 16 bytes per changed byte because MPI only allows for one datatype in one message so each address, which might be up to 8 bytes long, is joined with another 8 bytes which only contain 1 byte payload. So this is one point for future work, ideas here are first and foremost a much more efficient data transfer algorithm and further down the line a copy on write approach for all transferred data so there is no need to keep it in memory twice.

During the implementation of the memory transfer it was discovered that some memory might change rapidly, which means it is gone or changed before it is requested by the node on which the task will run. In order to mitigate this risk every variable which can be trivially copied is copied out. This applies mostly to loop counters and other simple types where the address is reused in every iteration.

3 Examples

All the talk about current capabilities is best explained by a simple test program in the repository which shall be explained here. Running a full stack, the preprocessor, then build the runtime into it requires some serious work before, so first there will be some minor examples to show how the runtime and the scheduler work before a full example with the preprocessor is given. There is a similar chapter in [Bau18], but the examples provided in this work are running and work correctly across multiple runtime and worker nodes.

3.1 The original source

The test program did not change from the previous work on the preprocessor, but for completeness it can be found below. All the following examples are descending from this and use the output of various levels of preprocessing.

```
1  #include <thread>
2  #include <chrono>
3
4  #define AS 1000
5
6  void test(int a[AS], int* p) {
7      for(int i = 0; i < AS; i++) {
8          #pragma omp task untied mergeable if(i == 3) final(i == 5) depend(in: a)
9          {
10             a[i] = i + *p;
11             // simulate a non trivial task
12             std::this_thread::sleep_for(std::chrono::milliseconds(1));
13         }
14     }
15 }
16
17 int main(int argc, char** argv) {
18     int* a = new int[AS];
19     int c = 0;
20     int* p = &c;
21
22     test(a, p);
23     return a[0];
24 }
```

Listing 3.1: Original source

3.2 One simple example

The following program is a copy and paste of all relevant files the preprocessor uses in order to build a full application. It only supports one worker and one runtime node and was created when the new runtime did not support memory transfer, but it can be used to show how the preprocessor transformes files and how the control flow works. This application can be built by simply linking it to the runtime library.

```
1  #include <mpi.h>
2  #include <iostream>
3  #include <thread>
4  #include <chrono>
5
6  #define AS 100
7
8  using namespace std;
9
10 #include <memory>
11
12 #include "tasking.h"
13
14 void x_1494453934 (size_t** arguments) {
15     void * a_pointer_1 = arguments[0];
16     void * a_pointer_0 = &(a_pointer_1);
17     int * a = *((int **) a_pointer_0);
18     void * i_pointer_0 = arguments[1];
19     int i = *((int*) i_pointer_0);
20     void * p_pointer_1 = arguments[2];
21     void * p_pointer_0 = &(p_pointer_1);
22     int * p = *((int **) p_pointer_0);
23
24     {
25             a[i] = i + *p;
26     }
27     current_task->worker->handle_finish_task(current_task);
28 }
29
30 int setup_1494453934() {
31     tasking_function_map[1494453934] = &x_1494453934;
32     return 1;
33 }
34
35 int tasking_setup_1494453934 = setup_1494453934();
36
37 void test(int a[AS], int* p) {
38     for(int i = 0; i < AS; i++) {
39         auto t_1494453934 = std::make_shared<Task>(1494453934);
40         t_1494453934->if_clause = (i == 3);
41         t_1494453934->final = (i == 5);
42         t_1494453934->untied = true;
43         t_1494453934->mergeable = true;
44         {
45             Var a_var = {"a", &(*a), at_firstprivate, 40000, 0};
46             t_1494453934->vars.emplace_back(a_var);
47         }
48         {
```

```
49              Var i_var = {"i", &(i), at_firstprivate, 4, 1};
50              t_1494453934->vars.emplace_back(i_var);
51          }
52          {
53              Var p_var = {"p", &(*p), at_firstprivate, 0, 0};
54              t_1494453934->vars.emplace_back(p_var);
55          }
56          t_1494453934->in.emplace_back("a");
57
58          current_task->worker->handle_create_task(t_1494453934);
59      }
60  }
61
62  int * __array__; // in order to test the values after completion
63  int cc;
64
65  void __main__1(int argc, char *argv[]) {
66
67      // Get the number of processes, test the world size for more than 2 workers
68      int world_size;
69      MPI_Comm_size(MPI_COMM_WORLD, &world_size);
70
71      if (world_size != 2) {
72          cout << "This code MUST be run by mpirun with 2 processes (no memory transfer)" << endl;
73          exit(EXIT_FAILURE);
74      }
75
76      {
77          __array__ = new int[AS];
78          cc = 2;
79          int* p = &cc;
80
81          test(__array__, p);
82          return a[0];
83      }
84  }
85
86  void __main__(int argc, char *argv[]) {
87      __main__1(argc, argv);
88      current_task->worker->handle_finish_task(current_task);
89  }
90
91  int main(int argc, char** argv) {
92      do_tasking(argc, argv);
93
94      auto e = EXIT_SUCCESS;
95      for (int i = 0; i < AS; i++) {
96          if (__array__[i] != i + cc) {
97              cout << "WRONG array value at " << i << ": " << __array__[i] << endl;
98              e = EXIT_FAILURE;
99          }
100     }
101
102     exit(e);
103 }
```

Listing 3.2: Simple example

In lines 14 to 35 the task code is extracted and the function is made available in the global `tasking_function_map`. The lines 15 to 22 recreate the variable names and types from the input to the function in order to let the code in line 25 access it the same way as if it was on shared memory. In the following lines, from line 39 to 58, the task is created according to the OpenMP declaration of the input in section 3.1. The additional code in lines 67 to 74 only check whether the correct preconditions for a successful program run are given. Finally, in the lines 94 to 100, the correct execution of the tasks is checked. The different main functions are used to contain the return at the end of the original main and the second wrapper is used to free the worker and finish the main task properly. The following code is the only thing needed to get this code to work, so it is not that difficult if the tasks are extracted. The name `tdomp` refers to the runtime and is the abbreviation for Task Distribution OpenMP.

```
1 add_executable(run_many test/run_many.cpp)
2 target_link_libraries(run_many tdomp)
```

Listing 3.3: Simple example - build script

3.3 A full example

Now the goal is to transform the code of section 3.1 using all the parts put together. This is done in the `cmake` file below. Note that `simple.cpp` refers to the source of the example. The preprocessor has to be built before.

```
1 add_custom_command(
2         OUTPUT ${CMAKE_BINARY_DIR}/simple.pre.cpp
3         COMMAND ../processor/processor ${CMAKE_CURRENT_SOURCE_DIR}/simple.cpp --outfile
              ${CMAKE_BINARY_DIR}/simple.pre.cpp
4         DEPENDS ../processor/processor ${CMAKE_CURRENT_SOURCE_DIR}/simple.cpp
5 )
6
7 add_executable(simple ${CMAKE_BINARY_DIR}/simple.pre.cpp)
8 target_link_libraries(simple PRIVATE tdomp)
```

Listing 3.4: Original source - build script

Invoking `mpirun` on the final executable produced a running binary which executed in different processes and produced correct results during the write back. All the additional clauses of the OpenMP task were ignored for this to work. The output of the preprocessor for this can be found below. Note that there are some differences to the example above, especially there is not check for correctness at the end.

```
1
2  #include "omp.h"
3
4  #include <thread>
5  #include <chrono>
6  #include <iostream>
7
8  #define AS 1000
9
10 #include <memory>
11
```

```
12 #include "tasking.h"
13 #include "/tmp/tasking_functions/all.hpp" // the extracted task sources are stored there
14 void test(int a[AS], int* p) {
15     for(int i = 0; i < AS; i++) {
16         #pragma //omp task untied mergeable if(i == 3) final(i == 5) depend(in: a)
17 auto t_813322099 = std::make_shared<Task>(813322099);
18 t_813322099->if_clause = (i == 3);
19 t_813322099->final = (i == 5);
20 t_813322099->untied = true;
21 t_813322099->mergeable = true;
22 {
23         Var a_var = {"a", &(*a), at_firstprivate, 4000, 0};
24         t_813322099->vars.emplace_back(a_var);
25 }
26 {
27         Var i_var = {"i", &(i), at_firstprivate, 4, 1};
28         t_813322099->vars.emplace_back(i_var);
29 }
30 {
31         Var p_var = {"p", &(*p), at_firstprivate, 0, 0};
32         t_813322099->vars.emplace_back(p_var);
33 }
34 t_813322099->in.emplace_back("a");
35
36         current_task->worker->handle_create_task(t_813322099);
37     }
38 }
39
40 int __main__1(int argc, char *argv[]) {
41 {
42     int* a = new int[AS];
43     int b[AS];
44     int c = 0;
45     int* p = &c;
46
47     test(a, p);
48     return a[0];
49 }
50 return 0;
51 }
52 void __main__(int argc, char *argv[]) {
53     __main__1(argc, argv);
54     current_task->worker->handle_finish_task(current_task);
55 }
56 int main(int argc, char** argv) { do_tasking(argc, argv); }
```

Listing 3.5: Original source - preprocessed

4 User Guide

4.1 Add new examples

4.1.1 How to add another example

In the source code there is one directory called test, add the new source files in that directory and copy the relevant sections from the `CMakeLists.txt` in the same directory and modify them to include the new files. Note that you have to run the preprocessor on every file which contains OpenMP tasks as well as the file which contains the main function. After invoking `cmake` to build the new target wait for it to finish. Because it builds the whole clang frontend it may take more than 20 minutes for the first time on a standard workstation. It is also recommended to install `ccache` if frequent changes to any `CMakeLists.txt` are made because such a change currently triggers a full rebuild of clang on the next build.

4.1.2 How to run resulting applications

Currently all applications and libraries use the full path in which they were created as a link target, so running them somewhere else requires setting the `LD_LIBRARY_PATH` environment variable to the directory in which they are located. Furthermore all applications created with the current library require a execution by `mpirun`, otherwise they only display an error message and exit. On top of that the parameter `worker_per_runtime` limits the number of usable workers to `100` because support for multiple runtime nodes is not finished at the moment.

4.2 Distribute another program

4.2.1 Build preparations

After receiving a copy of the full source code of this project one should create a new folder within this project and place the sources of the application which is about to be distributed in this folder. The recommended way to build it is to add the new folder to the top level `CMakeLists.txt` of this project and add dependencies to `tdomp`, the runtime library, and `processor`, the preprocessor, in the `CMakeLists.txt` of your project. Due to the complex dependency tracking the only build system currently supported is `cmake`. Then add a preprocessor run to every source file from which tasks should be extracted to the applications build steps. An example how this step might look like can be found in the file `test/CMakeLists.txt` in this project. If the dependencies between the generated files and the original target are set up correctly, for example by specifying them as the sources for the original binary, a full `cmake --build` call is enough to build the preprocessor, the runtime and the

target application with the preprocessed sources. The first build might take very long, and, because the long build is done every time any `CMakeLists.txt` is changed, it is recommended to use `ccache` or another caching mechanism. It is also recommended to delete `/tmp/tasking_functions` before a build in which filenames are changed. In this folder support files are created which are becoming outdated after a filename is changed.

4.2.2 Running the resulting application

After the build finishes the resulting application can no longer be ran by invoking it directly, because one node, the first one, is reserved for the runtime. In order to run the resulting application `mpirun` has to be used. Currently there is support for one task per worker process, so one might opt into running multiple mpi processes per node.

4.2.3 Debugging the code base

Due to the nature of the preprocessor compiler errors show up on the preprocessed file. Make sure to edit the original file and not the preprocessed ones, as they will be overwritten if the original files change. Please keep in mind that none of the OpenMP task clauses and neither taskwait nor taskloop instructions are currently supported by the runtime.

In order to debug the resulting application it is recommended to take a look at [The18], there is a list of several debugging methods for parallel programs using mpi. The approach that worked best during the coding for this work was to use

```
mpirun -np 4 xterm -e gdb -ex run the-resulting-applcation
```

but other approaches might work better than this for specific problems.

5 Current issues and future work

5.1 Runtime improvements

5.1.1 Support for multiple tasks

Currently the runtime only supports one task per worker and supporting more than one would require some more on the runtime node and the scheduler. This is considered to be low hanging fruit because it will be a great benefit for the resulting applications and only a moderate effort. The only part missing here is to prevent the scheduler from scheduling too many tasks on a node because a node might report capacity as free which was already requested by the runtime for a task. This can be also used to support internal OpenMP, which means OpenMP structs in offloaded tasks. In order to make this work every request for threads have to be intercepted by the preprocessor and the worker capacity reduced by the number of threads requested by the task.

5.1.2 Deallocate finished tasks

Currently the memory captured when a task is created is not made available again because the worker is not signaled when every child task is finished as and it is unclear how to only free memory the task really owns contrary to the memory which is shared between potentially running tasks. This would require the worker nodes when all children of a task finish and potentially the taskwait structure to be supported in order to allow the developers to free the memory when all children which might use it have already exited. In order to achieve this some more design work is needed, especially since there is neither support for paused or waiting tasks nor resuming of such tasks at the moment. This leads to leaking some memory in parent tasks which should be avoided for a production ready code.

5.1.3 Memory alignment

For some code it is beneficial to use vector instructions which require a certain alignment of the memory. This is currently not considered and might thus lead to incorrect results for those applications. In order to get this to work some research into memory allocation is needed and the routines in `Worker::request_memory` have to be adjusted to that.

5.1.4 Transfer non trivial memory

Currently only memory which somehow resolve to a trivially copyable type can be transmitted. This memory can be plainly copied and then cast to the original type without losing information. In order to support more complex programs it might be needed to move structures with pointers in them between nodes so some kind of support for them would be necessary.

5.2 Scheduler improvements

Currently the clauses which are used to configure tasks are not transferred to the scheduler from the node on which the task was created. This would be a first step for a larger project, which would be to take all clauses and apply them as close as possible to the OpenMP standard. It includes the dependency tracking between the tasks as well as how to initialize memory and when to execute a task right away instead of submitting it to the scheduler.

5.3 Conclusion and outlook

In order to have a full product, support for multiple tasks, see section 5.1.1, and deallocation of finished tasks, see section 5.1.2, is a prerequisite. Furthermore the dependency tracking part of section 5.2 is also needed in order to allow simple non artificial programs to run with this work. The rest of this section will also be needed for larger applications and for performance improvements. Those points will most likely be addressed by future work on the topic first.

Bibliography

[Bau18] M. Baur. “Execution of Task Parallel OpenMP Programs in Distributed Memory Environments - Preprocessor.” Universität Stuttgart, 2018 (cit. on pp. 5, 11).

[Cla18a] Clang Authors. *clang*. 2018. URL: https://github.com/llvm-mirror/clang (cit. on p. 7).

[Cla18b] Clang Authors. *Clang: a C language family frontend for LLVM*. 2018. URL: http://clang.llvm.org/ (cit. on p. 6).

[Erw18] J. Erwerle. “Execution of Task Parallel OpenMP Programs in Distributed Memory Environments - Runtime.” Universität Stuttgart, 2018 (cit. on p. 5).

[MPI18] MPI Forum. *MPI Forum*. 2018. URL: https://www.mpi-forum.org/ (cit. on p. 5).

[The18] The Open MPI Project. *FAQ: Debugging applications in parallel*. 2018. URL: https://www.open-mpi.org/faq/?category=debugging (cit. on p. 17).

All links were last followed on November 11, 2018.

6 Appendix

6.1 Header files

In order to have a rough overview of the implementation the header files are added below. Small headers are omitted.

```
1  //
2  // Created by markus on 1/11/18.
3  //
4
5  #ifndef PROCESSOR_TASKING_H
6  #define PROCESSOR_TASKING_H
7
8  #include <mpi.h>
9
10 #include "../src/globals.h"
11 #include "../src/Task.h"
12 #include "../src/Runtime.h"
13 #include "../src/Worker.h"
14
15 void do_tasking(int arg_c, char** arg_v) {
16     MPI_Init(&arg_c, &arg_v);
17
18     // Get the number of processes
19     int world_size;
20     MPI_Comm_size(MPI_COMM_WORLD, &world_size);
21
22     // Get the rank of the process
23     int world_rank;
24     MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);
25
26     if (world_size < 2) {
27         std::cout << "This code MUST be run by mpirun!" << std::endl;
28         exit(EXIT_FAILURE);
29     }
30
31     if (world_rank == 0) {
32         auto r = std::make_shared<Runtime>(world_rank, world_size);
33         r->setup();
34         r->run();
35     }
36     else {
37         auto w = std::make_shared<Worker>(world_rank);
38         w->setup();
39         w->run();
40     }
41
```

```
42      MPI_Finalize();
43  };
44
45  void taskwait() {
46      throw "taskwait is not implemented yet.";
47  };
48
49
50
51
52
53  #endif //PROCESSOR_TASKING_H
```

Listing 6.1: Tasking header

```
1   //
2   // Created by markus on 05.11.18.
3   //
4
5   #ifndef PROJECT_RUNTIME_H
6   #define PROJECT_RUNTIME_H
7
8   #include <map>
9   #include <mpi.h>
10  #include <vector>
11  #include <memory>
12
13  #include "utils.h"
14  #include "Scheduler.h"
15  #include "Receiver.h"
16
17  class Runtime : public Receiver {
18  public:
19
20      Scheduler scheduler;
21
22      explicit Runtime(int node_id, int world_size);
23
24      void setup();
25      void run();
26      void handle_message();
27
28      void handle_create_task(STask task);
29      void handle_finish_task(int task_id, int used_capacity, int source);
30
31      void run_task_on_node(STask task, int node_id);
32      void shutdown();
33
34
35  private:
36      int world_size;
37      int next_task_id = 0;
38
39  };
40
41
42  #endif //PROJECT_RUNTIME_H
```

Listing 6.2: Runtime header

```
//
// Created by markus on 05.11.18.
//

#ifndef LIBTDOMP_WORKER_H
#define LIBTDOMP_WORKER_H

#include <map>

#include "utils.h"
#include "Receiver.h"


class Worker : public Receiver {
public:
    explicit Worker(int node_id);

    // methods invoked locally
    void handle_create_task(STask task);
    void handle_finish_task(STask task);

    void handle_run_task(STask task);

    void request_memory(int origin, STask task);
    void handle_request_memory(int task_id, int source);

    void write_memory(STask task);
    void handle_write_memory(int task_id, int source);

    void setup();
    void run();
    void handle_message();

private:
    std::map<int, STask> created_tasks;
    std::map<int, STask> running_tasks; // In order not to deallocate tasks before their thread
        ends

    int runtime_node_id;
    int capacity = 1;
    int free_capacity = 1;

    bool should_run = false;
};


#endif //LIBTDOMP_WORKER_H
```

Listing 6.3: Worker header

```
//
// Created by markus on 06.11.18.
//

#ifndef PROJECT_SCHEDULER_H
#define PROJECT_SCHEDULER_H

#include <map>
```

```
#include <memory>
#include <vector>
#include <list>

#include "utils.h"

class Task;

class Scheduler {
public:
    void add_worker(std::shared_ptr<RuntimeWorker> worker);

    void enqueue(STask task);
    void set_finished(STask task);
    bool work_available();

    STask first_unfinished_child(int task_id);

    std::pair<int, STask> get_next_node_and_task();

    std::map<int, STask> get_all_tasks() {return created_tasks;}; // TODO remove

    std::map<int, std::shared_ptr<RuntimeWorker> > workers;


    std::map<int, STask> created_tasks;
    std::map<int, STask> running_tasks;
    std::list<STask> ready_tasks;

private:

    bool can_run_task(STask task);

};


#endif //PROJECT_SCHEDULER_H
```

Listing 6.4: Scheduler header

```
//
// Created by markus on 05.11.18.
//

#ifndef LIBTDOMP_TASK_H
#define LIBTDOMP_TASK_H

#include <vector>
#include <list>
#include <string>
#include <thread>
#include <mutex>

#include "utils.h"

class Worker;

class Task {
```

```
19 public:
20     explicit Task(int code_id);
21 
22     void prepare();
23     void update(STask other);
24     void free_after_run(bool ran_local);
25     void free_after_children();
26 
27     std::vector<int> serialize();
28 
29     static STask deserialize(int * input);
30 
31     int code_id;                // serialized
32     int task_id;                // serialized
33     int parent_id;              // serialized
34     int origin_id;              // serialized
35     bool finished;              // serialized
36     bool running;
37     int capacity;
38     int variables_count = -1;   // serialized (and calculated if unknown)
39     std::thread* run_thread = nullptr;
40     Worker * worker; // shared_ptr and weak_ptr do not work here (shared_from_this is not
           working reliably)
41 
42     std::list<std::weak_ptr<Task> > children;
43     bool children_finished = false;
44     size_t** memory;
45     std::mutex memory_lock; // Only one thread should mess with the memory at one time
46 
47 
48     bool if_clause; // if false, parent may not continue until this is finished
49     bool final; // if true: sequential, also for children
50     bool untied; // ignore, continue on same node, schedule on any
51     bool shared_by_default; // ignore: syntax: default(shared | none) - are values shared by
           default or do they have to have a clause for it
52     bool mergeable;// ignore
53     std::vector<Var> vars; //all variables, in order
54     std::vector<std::string> in; //list of strings, runtime ordering for siblings, array
           sections?
55     std::vector<std::string> out;
56     std::vector<std::string> inout;
57     int priority; //later
58 };
59 
60 
61 #endif //LIBTDOMP_TASK_H
```

Listing 6.5: Task header

Declaration

I hereby declare that the work presented in this thesis is entirely my own and that I did not use any other sources and references than the listed ones. I have marked all direct or indirect statements from other sources contained therein as quotations. Neither this work nor significant parts of it were part of another examination procedure. I have not published this work in whole or in part before. The electronic copy is consistent with all submitted copies.

---

place, date, signature