# Course Project: Deadline-2

## Improving the Locality of ArgoLib Parallel Runtime

Due by 11:59pm on 6th November 2022
(Total **15%** weightage)

Instructor: Vivek Kumar

**No extensions will be provided**. Any submission after the deadline will not be evaluated. If you see an ambiguity or inconsistency in a question, please seek clarification from the teaching staff.

**Plagiarism**: **This is a pair programming-based project that you must do with the group member that you have already chosen. No change to the group is allowed. You are not allowed to discuss the approach/solution outside your group.** You should never misrepresent some other group's work as your own. In case any plagiarism case is detected, it will be dealt as per the new plagiarism policy of IIITD **and will be applied to each member in the group.**

## General Instructions

a) Hardware requirements:
    a) You will require a Linux/Mac OS to complete this project. For your final experimentations, we will provide you access to a 32-core / 20-core processor running Ubuntu OS.
b) Software requirements are:
    a) C++11 compiler and GNU make.
    b) You **must** do version controlling of all your code using github. You should only use a **PRIVATE** repository. If you are found to be using a **PUBLIC** access repository, then it will be considered plagiarism. NOTE that TAs will check your github repository during the demo.

### Requirements

1) This is a group project where each group comprises a maximum of **two** students. You cannot change your group member once the project deadline-1 has started.
2) You may choose to work as a single member group, but there will not be any relaxations in marking scheme/deadlines, and the same rubric will be followed for each group.
3) We are going to use deadline chaining in this course project, where the runtime built for the first deadline will be the input for the next deadline. Hence, don't miss any deadline, as in that case you would miss the next deadline automatically.

# Project Details

In this deadline, you have to further optimize your ArgoLib parallel runtime by improving its support for locality over modern multicore processors. You will have to implement **ONE** of the optimizations described below. You should implement this optimization on top of your base implementation of ArgoLib, i.e., the version of ArgoLib you implemented under **Part-1** of the Project deadline-1. You must implement your solution by using the facilities provided by Argobots. You should use compile time flags to enable/disable building of your default implementation of ArgoLib, and for building your modified ArgoLib implementation. The options available for improving the locality are as follows:

**1) NUMA awareness by top-level task distribution and using hierarchical work-stealing**

This optimization was covered in Lecture 11 slides 23-26. You will have to first divide the top-level async computations into “N” parts, where “N” is the total number of NUMA domains. After this you would have to implement/use a hierarchical work-stealing implementation inside ArgoLib.

You can use the following format for reading the NUMA processor design given by the user (feel free to use a different format but it should allow us to create NUMA domains as below):
Example_1:
0:4
1:8
In the above format for each line, "x:y” means “x” is the hierarchy level and “y” is total number of NUMA_domain/workers below it. For this example, line_1 is 0:4 i.e. this processor (root=0) contains 4 sockets (4 NUMA domains). Line_2 is 1:8 signifies level 1 of hierarchy having 4 distinct NUMA domains and each having 8 workers/cores.
Example_2:
0:2
1:2
2:2
Here, processor (root=0) contains 2 sockets. Level_1: Each socket contains 2 NUMA domains (i.e. 2x2=4 NUMA domains). Level_2 (leaf level): Each sub domain contains 2 cores/workers (total workers in system = 2x2x2 = 8).

Once each of the NUMA node receives a seed task (one per node), hierarchical work-stealing should be used for dynamic load balancing. Lecture 11 slides 23-26 explained an implementation of hierarchical work-stealing runtime. A detailed implementation of such a runtime is also explained as HWS scheduler in the following paper that:
https://hal.inria.fr/file/index/docid/429624/filename/RR-7077.pdf

Note that although the above paper is about work-stealing across clusters, you have to implement it for a shared memory NUMA processor that can also be imagined like a distributed platform. This is due to difference in latency to access memory across different NUMA domains.

**2) Improving the locality of iterative algorithms by using trace/replay**

Iterative applications using task-parallelism can be represented as following pseudocode:

```
for(int i=0; i<MAX; i++) {
   /* Below computation generates exact same computation graph for all value of “i” */
   computation_kernel_using_fork_join();
   /* Next iteration of for loop start only after all the tasks in current iterations are terminated */
}
```

The program can be executed using a version of ArgoLib that supports trace/replay as follows:

```
for(int i=0; i<MAX; i++) {
   /* Below computation generates exact same computation graph for all value of “i” */
   argolib::start_tracing();
   computation_kernel_using_fork_join(); /* User code – not part of the ArgoLib */
   argolib::stop_tracing();
   /* Next iteration of for loop start only after all the tasks in current iterations are terminated */
}
```

Pseudocode implementation of the above two APIs inside the ArgoLib runtime are as following:

```
/* Task metadata will have “unsigned int ID” counter */
/* Worker metadata has “unsigned int AC” & “unsigned int SC” counters as mentioned in Slide #16 */
// Global variables
static int tracing_enabled = false;
static int replay_enabled = false;

void argolib::start_tracing() {
   tracing_enabled = true;
   reset_worker_AC_counter(numWorkers); // See Lecture #13, Slides #16
   /* Each worker’s AC value set to (workerID * UINT_MAX/numWorkers) */
   reset_worker_SC_counters(numWorkers);
}

void argolib::stop_tracing() {
   if(replay_enabled == false) {
      list_aggregation(numWorkers);  // See Lecture #13, Slides #35-36
      list_sorting(numWorkers);                    // See Lecture #13, Slides #37
      create_array_to_store_stolen_task(numWorkers);       // See Lecture #13, Slides #39-40
      replay_enabled = true;
   }
}
```

Changes to rest of the default APIs inside ArgoLib will be as following:

```
Task_handle* argolib_fork(fork_t fptr, void* args) {
   …..
   task = create_task(fptr, args);
   task->ID = ++current_worker()->AC;
   if(replay_enabled) sendTo_thief_if_task_was_stolen_during_tracing(); // See Lecture #13, Slides #41-43
   else push_to_deque_as_in_default_case();
}

Task* steal() {
   W = current_worker();
   // See Lecture #13, Slides #44
   if(replay_enabled) spin_until_victim_transferred_task_at_index_SCvalue();
   else {
      /* Steal from victims as in default case */
      Task *t = stolen_task();
      if(tracing_enabled) record_task_stolen_from_victim(); // See Lecture #13, Slides #19
   }
}
```

Rest of the algorithm is clearly explained using detailed animation in Lecture #13 slides. Hence, we are not referring any paper for trace/replay. You can use the default concurrent deque both during trace and replay, i.e., you are not required to switch to private deque implementation during the replay phase.

**3) Implementing/demonstrating any other locality/NUMA optimization not taught in lectures**

Instead of implementing any of the above two optimizations taught in lectures, you can also implement some idea/optimization related to improving the locality/performance over NUMA architectures. If you wish to do so, you would have to first email me a detailed writeup of your proposed implementation. **The deadline to do so is 26th October midnight**. Your proposed implementation should be at par in terms of the effort required in above two options. Your implementation should not be outside ArgoLib.

## Testing

You can test your implementation (any of the above options) using the iterative averaging example taught in Lecture 13. The sequential implementation of that benchmark is available in CSE513 course GitHub repository. Please feel free to modify the testcase to suit your purpose. **You must submit your modified testcase (and Makefile) along with your runtime implementation**.

For this project you require numactl (https://linux.die.net/man/8/numactl) for using interleaved memory allocation policy. You will also have to bind worker threads to the physical cores. You can use the Argobots APIs for binding the execution streams to individual physical cores. ~~You can borrow this code from HClib: https://github.com/vivkumar/cse502/blob/master/hclib/src/hclib-thread-bind.c~~