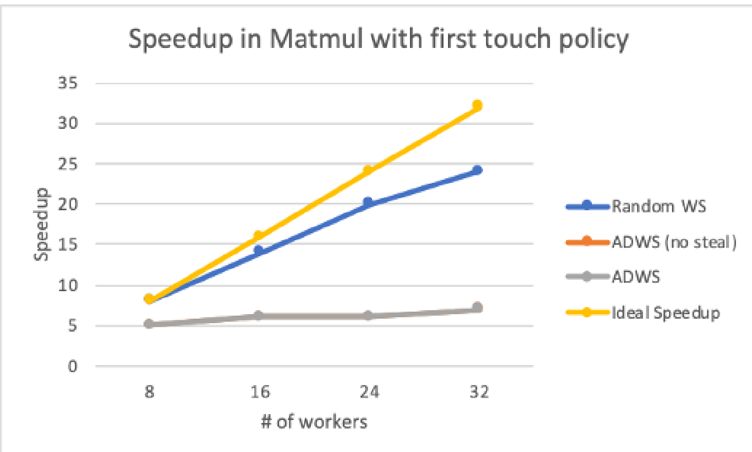
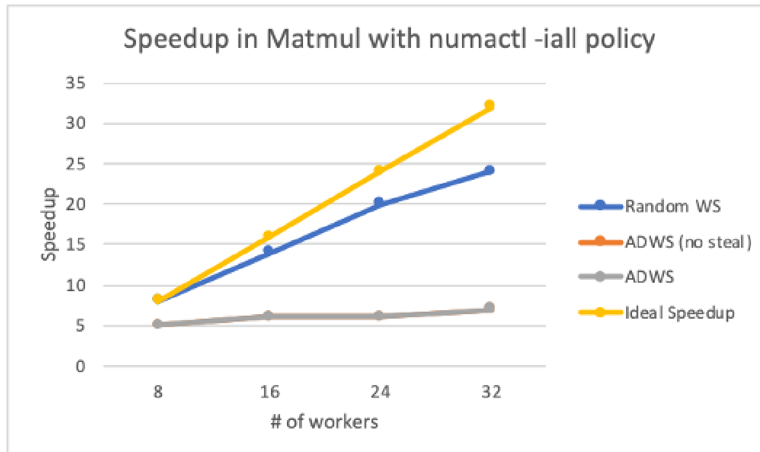
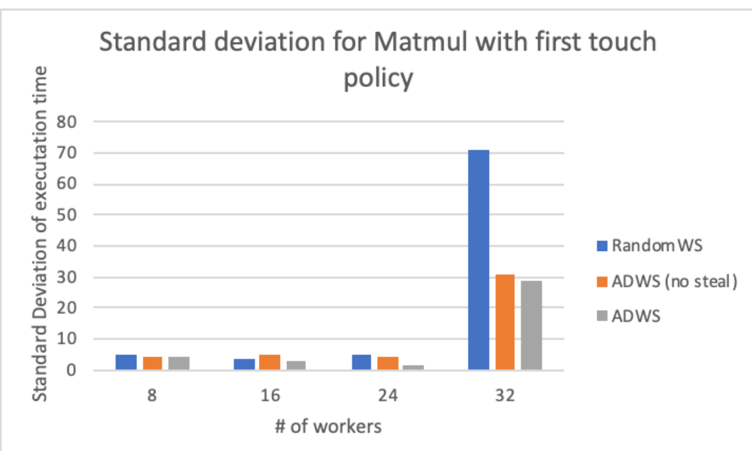
Project Report

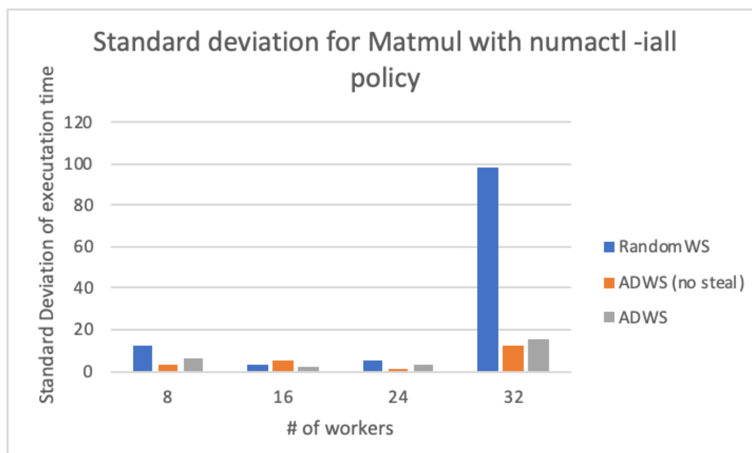
Aayush Gupta - 2017002
Apurv Singh - 2017027

Section 1

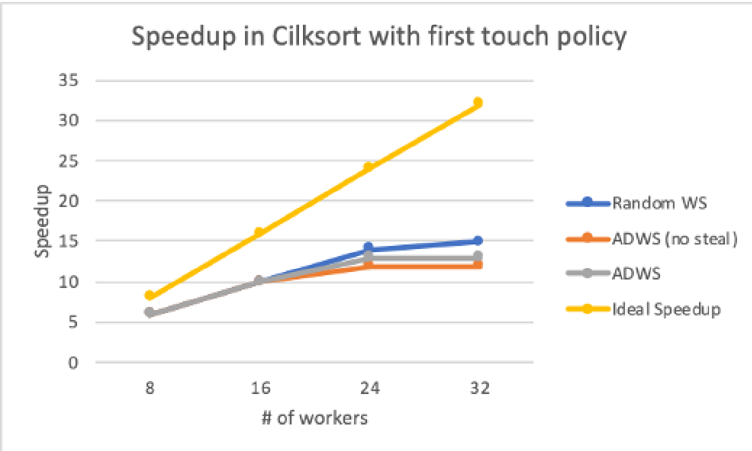


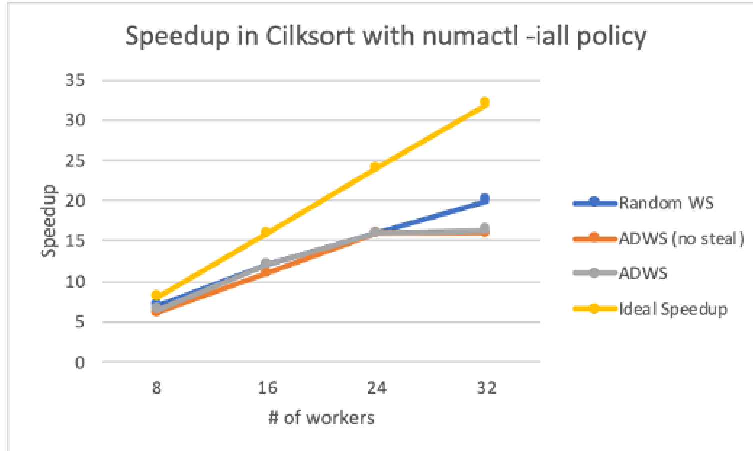


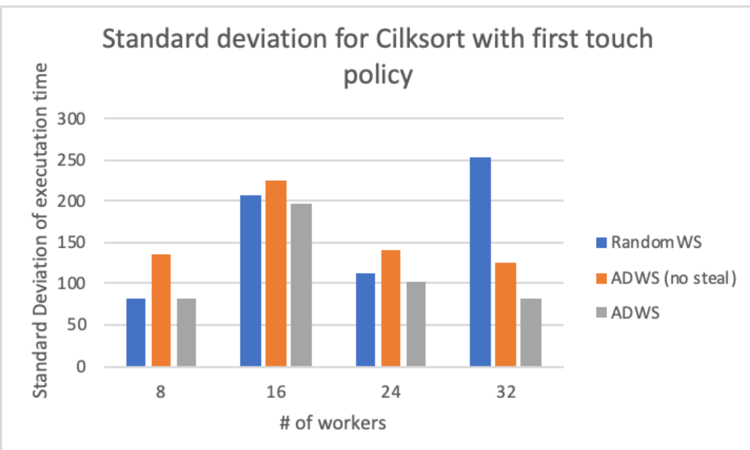


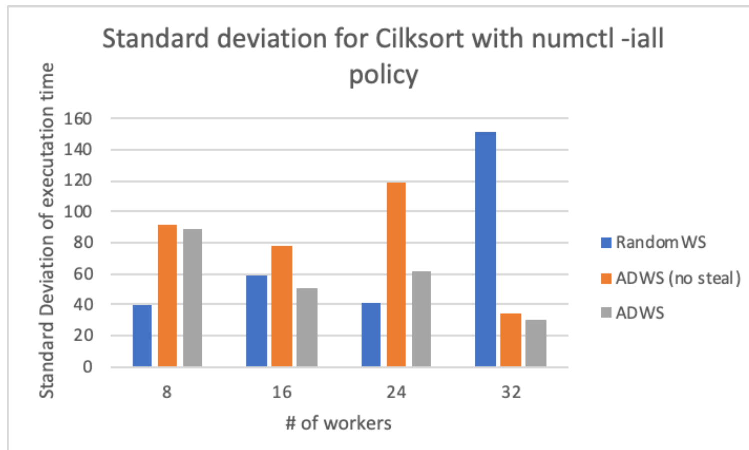


The speedup obtained in matmul.cpp benchmark by AWDS is almost equivalent to that of the speedup of ADWS (no steal). But we do not see a significant increment in the speedup in comparison to Random WS.
We see that for a lower number of workers all 3 of them show very less std. But for 32 workers Random WS is very high but ADWS and ADWS (no steal) are low. ADWS (no steal) is even lower than that of ADWS.



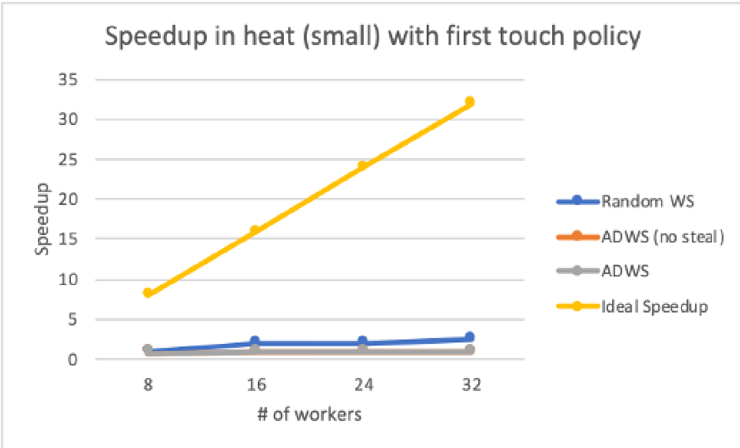


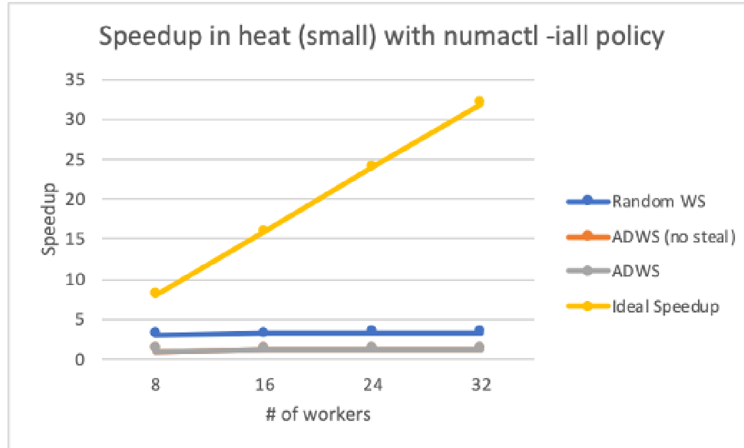


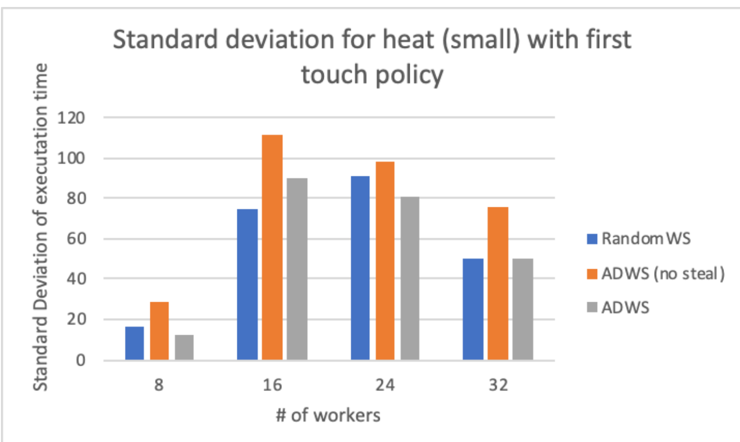


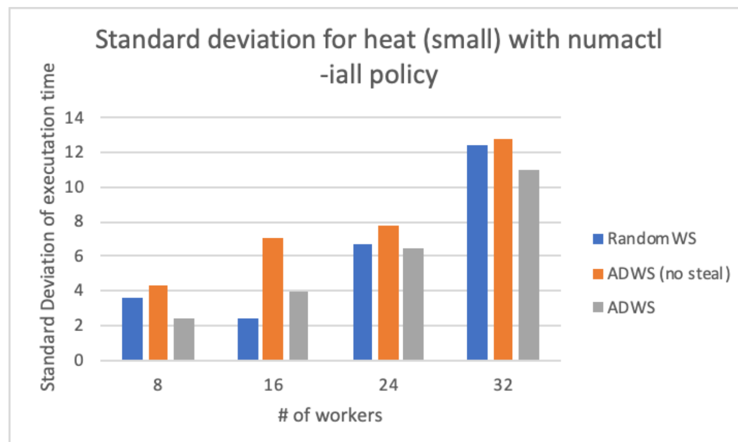
The speedup obtained in cilksort.cpp benchmark by AWDS is a bit better than that of the speedup of ADWS (no steal). For most of the plot ADWS, ADWS (no steal) and Random WS are equivalent. But on a higher number of workers, Random WS becomes better.

We see that for lower number of workers ADWS (no steal) shows more std whereas Random WS and ADWS show similar std. But for 32 workers Random WS is very high but ADWS and ADWS (no steal) are low. ADWS is even lower than that of ADWS (no steal).



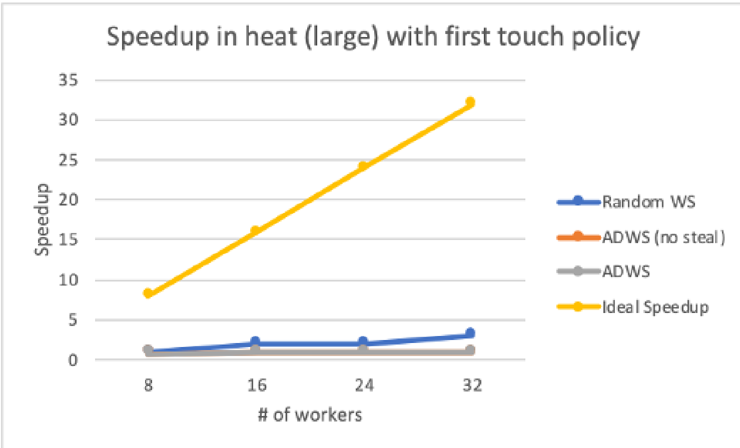


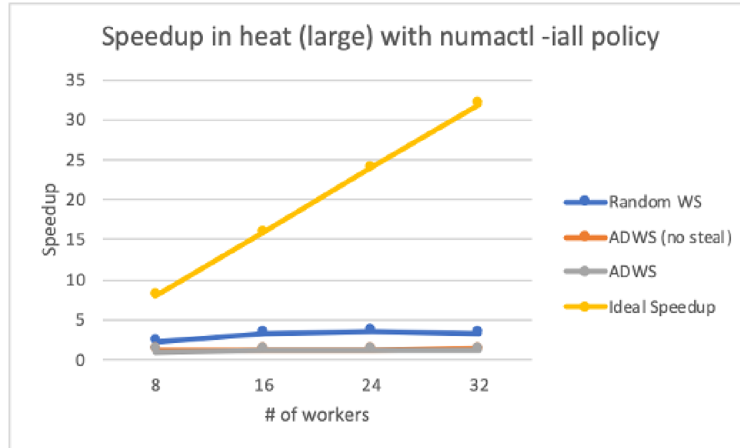


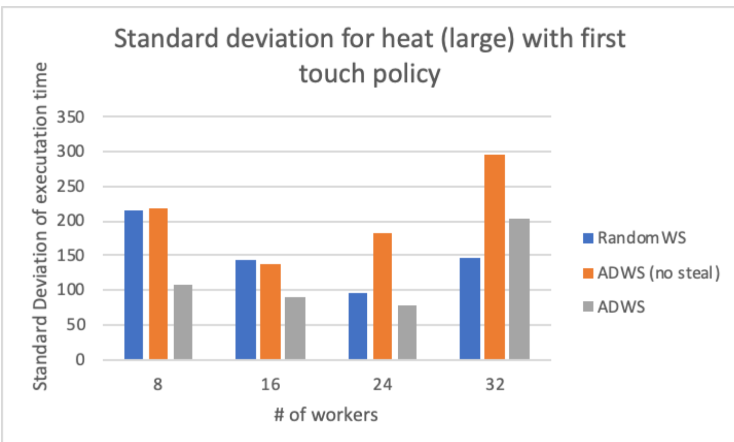


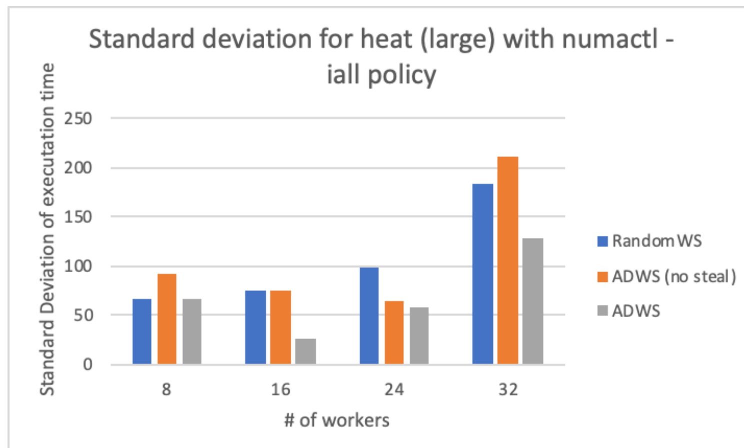
The speedup obtained in heat.cpp benchmark with the argument as 0 by AWDS is almost equivalent to that of the speedup of ADWS (no steal). But we do not see a significant increment in the speedup in any of them.

We see that ADWS (no steal) shows more std whereas Random WS and ADWS show similar std.



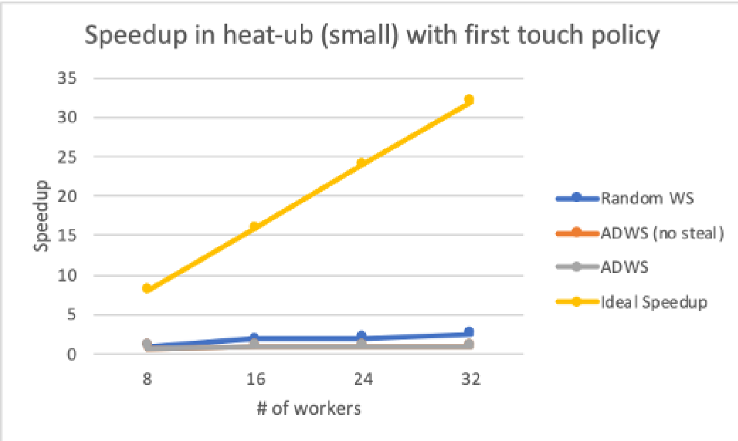


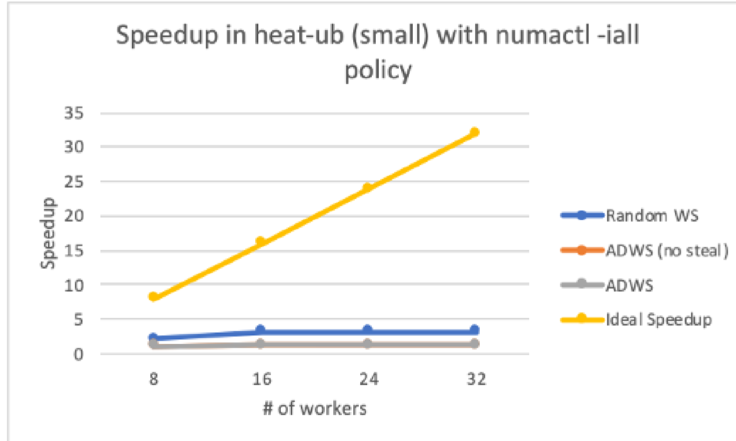


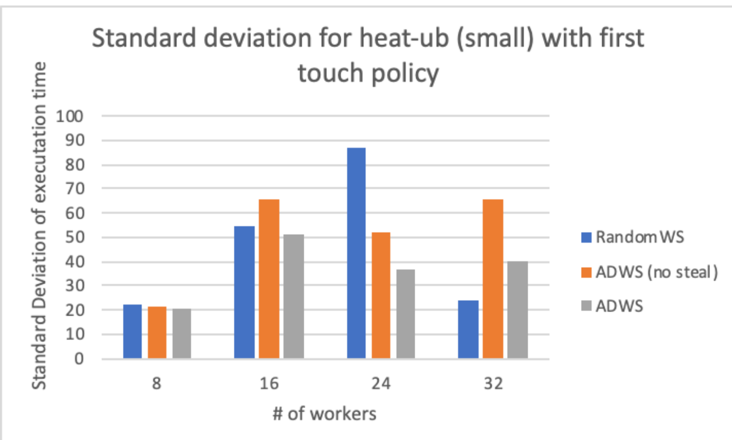


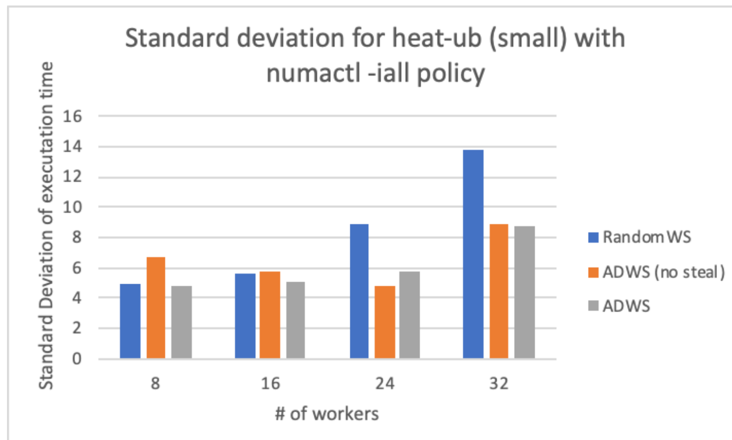
The speedup obtained in heat.cpp benchmark with the argument as 1 by AWDS is almost equivalent to that of the speedup of ADWS (no steal). But we do not see a significant increment in the speedup in any of them.

We see that ADWS (no steal) shows more std whereas Random WS and ADWS show similar std.



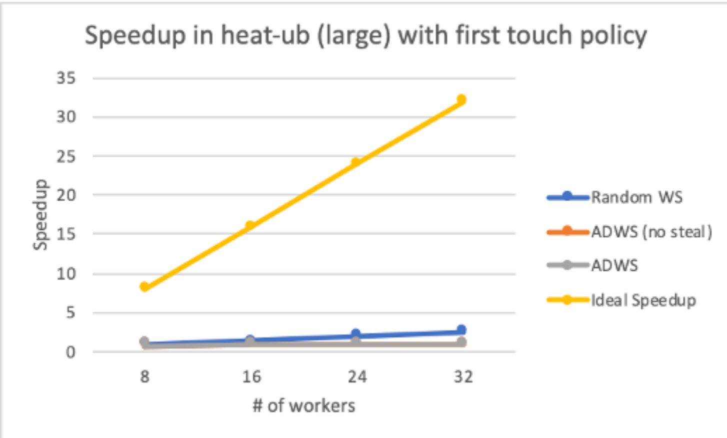


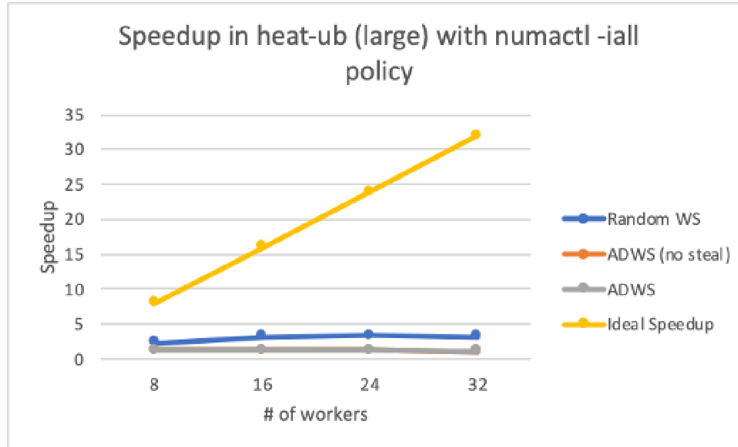


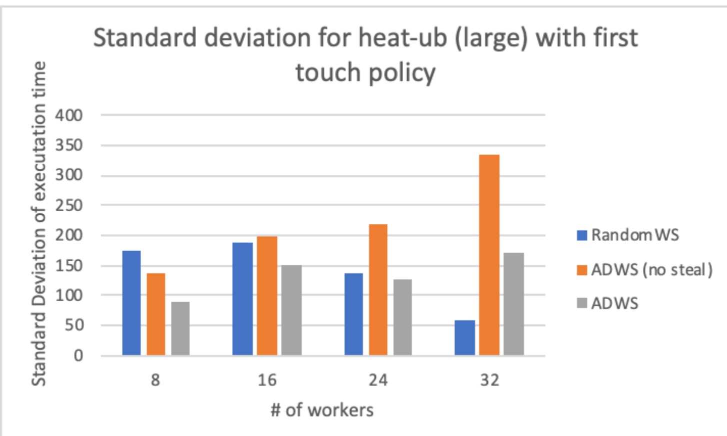


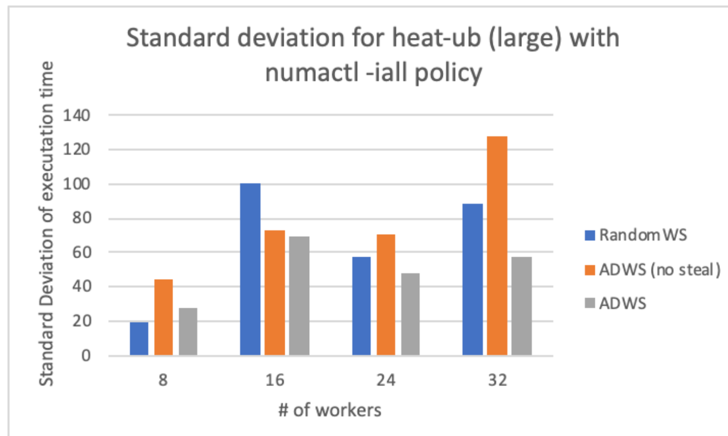
The speedup obtained in heat-ub.cpp benchmark with the argument as 0 by AWDS is almost equivalent to that of the speedup of ADWS (no steal). But we do not see a significant increment in the speedup in any of them.

We see that for a lower number of workers ADWS (no steal) shows more std whereas Random WS and ADWS show similar std. But for more number of workers Random WS is very high but ADWS and ADWS (no steal) are low. ADWS is even lower than that of ADWS (no steal).



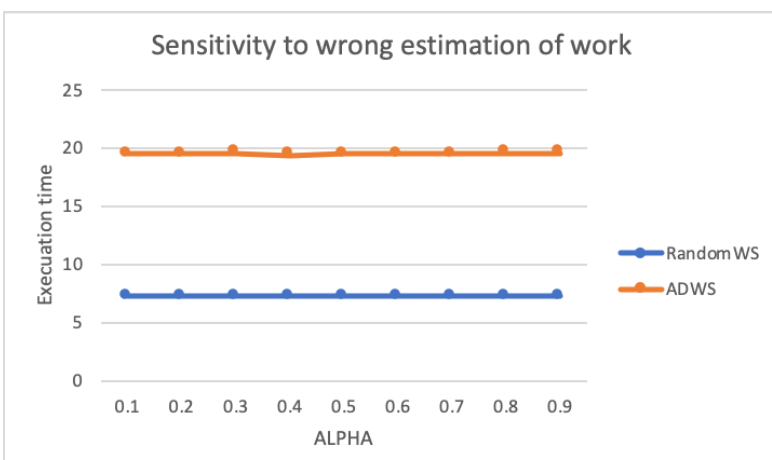






The speedup obtained in heat-ub.cpp benchmark with the argument as 1 by AWDS is almost equivalent to that of the speedup of ADWS (no steal). But we do not see a significant increment in the speedup in any of them.

We see that ADWS (no steal) shows more std whereas Random WS and ADWS show similar std.



We do not observe much change with the change in alpha values.

	n = 20	n = 40	n = 50
Random WS	0.316 ms	232.199 ms	32.674 s
ADWS	0.212 ms	284.591 ms	34.489 s

Fibonacci benchmark where ADWS is better than Random WS for n=20. But as n increases the overheads increase thus ADWS becomes worse than Random WS.

Section 2

Part A

Hclib-runtime.cpp

- Inline struct from check_in_finish and check_out_finish was removed
- slave_worker_finishHelper_routine and worker_routine:- was changed to incorporate poping first from the buffer.

Hclib-hpt.cpp

- Hpt_pop_task:- was changed to incorporate poping from the buffer the local deque (LIFO) and then migratory deque (FIFO).
- Hpt_steal_task:- was changed to incorporate stealing from a specified range which is specific to each worker. This method was implemented with the help of a task finding code as the backbone of the code.

Hclib-async.h

- async(T lambda, double weight):- a new async function which takes a weight as input as well and uses a new spawn function to spawn the tasks.

Hclib-asyncStruct.h

- void spawn(task_t * task, double weight):- a new construct for the new spawn was added.

Hclib-rt.h

- Added finish(std::function<void()> lambda, double totalWeight) function, a finish function that also accepts a total weight parameter.
- Added start_finish(double totalWeight) function, a start_finish function that also accepts a total weight parameter.
- Added check_in_finish(finish_t * finish) as inline struct was removed.

- Added check_out_finish(finish_t * finish) as inline struct was removed.
- Created a hc_context * get_hclib_context() function to access list of worker instances.
- Added execute_task(task_t* task) as inline struct was removed.

Hclib-internal.h

- Added an integer variable and a Node variable in finish_t to store the total weight and the stealRange node respectively.
- Added a migratory in hc_deque_t struct.
- Added the struct of Node.
    - This is the backbone code for task stealing process.
    - It contains the range parameter left, right and the pointer. It also stores the parent of the current steal node along with a list of its children and their count. There is boolean isActive to check whether the current node is active or not.
    - There are Node(constructor), addChild(to add a child node), activateStealRange and deactivateStealRange (to change the isActive variable) and the findTask functions.
    - findTask function iterates from the current node to the parent node to find a task. At any node, it tries to find a task in the allowed range from the buffer, migratory and local deques.

Hclib-fpp-project.cpp

- finish(std::function<void()> lambda, double totalWeight) function, a finish function that also accepts a total weight parameter.
- start_finish(double totalWeight) function, creates a new finish object.if there exists a parent then it assigns the finish with the Node variables from the parent stealRange node, if there doesn't exist a parent then it assigns the complete range. This new finish function is set to be the finish of the current worker.
- spawn(task_t * task, double weight) function, with the help of variables from the stealRange node deterministically gets the steal range of the task. Then it creates a new finish for the task. The task is assigned the new finish and is then pushed into either the current local deque or the migratory deque of another worker depending on its steal range.

Part B

Major issue face during the implementation of the project was:

- Stealing within steal range
    - Our code tried to steal in migratory deque > local deque priority order.
    - But we observed that there were very less executable tasks in migratory deque in general. This caused the code to crash and give segfault.
    - So we changed the preference order to local deque > migratory deque. This solved our problem as most of the times there is some executable task in the local deque.

- Our solution prevents multiple accesses to the migratory deque.

Section 3

During deterministic allocation, if the current range that needs to be partitioned is within a single worker (eg 3.1 to 3.8) then we do not further divide this range, we give the task this complete range as steal range.
This has helped us reduce the load on our back end task finding code and has also prevented precision problems.