Artifact Description of the paper: LazyD: Compiling For Just Enough Parallelism

CHRISMA PAKHA and SETH COPEN GOLDSTEIN, Carnegie Mellon University, USA

This artifact describes how to download and evaluate our compiler from the paper 'LazyD: Compiling For Just Enough Parallelism'. LazyD is a compiler and runtime for programs written in Cilk. The compiler converts fork-join and parallel-for constructs into low overhead parallel-ready sequential code. Parallelism is realized via work requests. Hence, the control overhead of managing parallel constructs is virtually eliminated when parallelism is not needed. LazyD attempts to improve the performance of a parallel program when the overhead of the parallel construct is significant and when using a coarser grain size can cause load imbalance on a higher core count. It relies on stack-walking, code-versioning, and polling to generate low-overhead parallel constructs. It is built on top of the OpenCilk compiler and takes advantage of Tapir.

The artifact contains instructions for downloading a Docker image containing the compiler, benchmarks, binary to generate the datasets, and scripts to evaluate LazyD's performance. Our artifact is based on the modified version of Cilkbench from https://github.com/neboat/cilkbench and PBBS-Bench from https://github.com/cmuparlay/pbbsbench. We use the same compilation command from CilkBench and PBBSBench but a different script to invoke the compilation command and run the evaluation.

1 RUNNING THE DOCKER.

Download the container containing the compiler, libraries, test scripts, and benchmarks:

```
docker pull cpakha/lazydcompiler:latest
```

All the experiments will be executed from within the container. The various compiled files, test data, and results files will be written in the container directory `/home/user/lazyDir`, which is mapped to the host filesystem at `<host_directory>` as specified in docker run command. Before executing the docker run command, `<host_directory>`, must exist. After creating `<host_directory>`, run the container with the following command:

```
docker run --privileged \
   -v <host_directory>:/home/user/lazyDir \
   -it cpakha/lazydcompiler:latest
```

Optionally, the following flags can make it easier to work with the system:

- `--user=<uid>:<gid>` This will set the user ID to <uid> and the group ID to <gid>. This will make it easier to modify container-written files from the host. You can determine your UID/GID on Linux with the `id` command.
- `--rm` Remove the container once the docker has been exited.

Authors' address: Chrisma Pakha, cpakha@andrew.cmu.edu; Seth Copen Goldstein, seth@cmu.edu, Carnegie Mellon University, Pittsburgh, Pennsylvania, USA.

If you are running the container for the first time, you will need to run the `setup.sh` script to initialize the `lazyDir` directory in the container with the benchmarks and various other files.

```
./setup.sh
```

Figure 1 shows the directory tree of the container after the `./setup.sh` script has been executed.

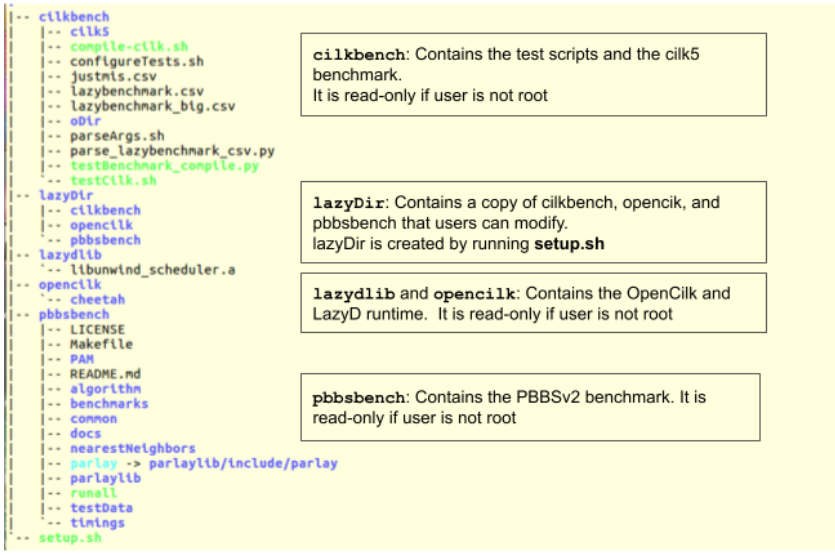

Fig. 1. *Tree structure of the container from /home/user.*

2 BASIC OPERATION.

The benchmarks can be compiled and executed using the `./testBenchmark_compile.py` script. The script processes a CSV file that contains the benchmarks and data sets to be used for each run. In `lazyDir/cilkbench` we supply four example .csv files: `justmis.csv`, `cilk5benchmark.csv`, `lazybenchmark.csv`, `lazybenchmark_big.csv`. `justmis.csv` contains information to run a single benchmark. `lazybenchmark.csv`, `cilk5benchmark.csv` contains information to run Cilk5 benchmarks, `lazybenchmark_big.csv` contains information to run the PBBSv2 benchmarks on small and large datasets, respectively.

2.1 Running the Test script to compile and execute MIS.

The following shows an example of compiling, executing, and verifying the MIS using the rMat-Graph_JR_12_2250000 dataset. This command must be executed at the `lazyDir/cilkbench` directory.

```
./testBenchmark_compile.py \
        --num_tests=<number of runs> \
        --num_cores=<number of workers> \
        --fg yes --schedule_tasks PBBS DELEGATEPRCPRL OPENCILKDEFAULT_FINE \
        --parallel_framework tapir lazyd0 \
        --ifile=justmis.csv
```

The above script will execute each benchmark in `justmis.csv` `<number of runs>` times on `<number of workers>` cores. Each benchmark will be compiled using the tapir (OpenCilk) and the LazyD frameworks (as per `-parallel_framework` argument), using both fine-grained and coarse-grained tasks (as per the `-fg` argument), and run using the PBBS, DELEGATEPRCPRL, and OPENCILKDEFAULT_FINE schedulers (as per the `-schedule_tasks` argument). Some combinations of options are not compatible, so only those that make sense are run. For the options listed here, only four cases make sense.

The format of the csv file is described in Section 2.3. The output of this script is located in `oDir/lazybenchmark_<datetime>/lazybenchmark_results.csv`, where <datetime> is the date you ran the command.

2.2 Under the Hood.

In this section, we describe the commands executed by `./testBenchmark_compile.py` to compile and execute the MIS benchmark as per the previous section. Feel free to skip this if you just want to run the benchmarks, but if you are interested in creating your own benchmarks, this section will be helpful.

To compile the OpenCilk and LazyD binary, run the following in the `user/lazyDir/cilkbench` directory:

```
# In lazyDir/cilkbench, compile MIS using LazyD
./testBenchmark_compile.py \
        --compile \
        --fg yes --schedule_tasks DELEGATEPRCPRL \
        --parallel_framework lazyd0 --ifile=justmis.csv
```

See Listing 1 for available options provided to the users. Note that these settings can only be used for the PBBSv2 benchmarks.

The actual compile commands generated can be seen if you run with the `-dryrun` option.

```
cd ./pbbs_v2/maximalIndependentSet/incrementalMIS \
   && make clean \
   && POLL0=1 GRAINSIZE8=1 DELEGATEPRCPRL=1 make \
   && mv MIS MIS.Dclnyuf
```

For Opencilk, run the following:

```
# In the cilkbench directory located inside lazyDir,
# compile MIS using OpenCilk
./testBenchmark_compile.py --compile \
  --parallel_framework tapir --ifile=justmis.csv
```

Use the following command to create the dataset needed to run MIS:

```
# In the cilkbench directory located inside lazyDir,
cd pbbs_v2/maximalIndependentSet/graphData/data \
   && make rMatGraph_JR_12_2250000
```

The dataset for each benchmark can be found in `/lazyDir/pbbs_v2/<benchmarkname>/<implname>/testInputs_{small}`.

Once the binary and dataset have been generated, run the following to evaluate the performance on *N* core *R* times:

```
# In user/lazyDir/cilkbench/pbbs_v2/maximalIndependentSet/incrementalMIS
CILK_NWORKERS=N ./MIS.<lazyd or opencilk> \
       -r R -o outputfile \
       -i ../graphData/data/rMatGraph_JR_12_2250000
```

To ensure that the MIS generates the correct output, run the following command:

```
# In user/lazyDir/cilkbench/pbbs_v2/maximalIndependentSet/bench
CILK_NWORKERS=1 ./MISCheck \
      ../graphData/data/rMatGraph_JR_12_2250000 \
      ../incrementalMIS/outputfile
```

On a successful run, the program should return zero or not print any error messages. Once a benchmark has been compiled, `testBenchmark_compile.py` will not recompile the benchmark again and instead immediately executes the benchmark and verifies it using the following command.

```
# In the cilkbench directory located inside lazyDir,
# run the following
./testBenchmark_compile.py \
        --execute \
        --num_cores=N --num_tests=R \
        --ifile=justmis.csv
```

In order for the user to force a recompilation using the `testBenchmark_compile.py`, we provide a script `rm-all-exes.sh` that looks for all executables in the `cilk5` directory and `pbbs_v2/<benchmarkname>/<implname>` directory and removes it.

By default, `testBenchmark_compile.py` uses the `numactl -interleave=all` options to improve parallel performance. However, it may not work on all machines. If users encounter any issue related to NUMA, add the `--disable_numa` flag when executing `testBenchmark_compile.py`.

2.3 Compiling and Executing the PBBSv2 benchmark.

We use a CSV file to specify the benchmarks and datasets to be compiled and executed. If a CSV file is not specified, it will use lazybenchmark.csv as the default file.

The `lazybenchmark.csv` for PBBS_v2 is in the following format:

- The first column shows the name of the folder that contains the benchmark.
- The second column shows the name of the benchmark and its implementation.
- The third column shows the name of the binary that runs the algorithm.
- The fourth column shows the name of the binary to verify correctness.
- The fifth column shows the name of the folder that contains the dataset.
- The sixth column is currently not used.
- The seventh column shows the dataset used to run the algorithm.

Note that the first line of the CSV file must be a comment (indicated by the # symbol). To run a large set of the PBBSv2 benchmarks using LazyD, run the following command in the `lazyDir/cilkbench` directory.

```
# For LazyD
./testBenchmark_compile.py \
        --num_tests=<number of runs> \
        --num_cores=<number of workers> \
        --fg both --schedule_tasks DELEGATEPRCPRL \
        --parallel_framework lazyd0 tapir
```

If using `lazybenchmark.csv` as its input, it requires 6-10GB of memory.
For `lazybenchmark_big.csv`, it requires 50-70 GB of memory. The result of the evaluation is located in `home/user/lazyDir/cilkbench /oDir/lazybenchmark_*/results.csv`. It is in the following format:

- BENCHMARK: The first column shows the name of the benchmark.
- COMPILES: The second column shows whether it compiles.
- DATASET: The third column shows the dataset.
- NUM CORES: The fourth column shows the number of workers.
- STATUS: The fifth column shows whether the benchmark was successfully executed or not.
- DISABLE_NUMA: The sixth column shows whether numactl –interleave=all was used or not when executing the benchmark.
- PARALLEL_FRAMEWORK: The seventh column shows which parallel framework that was used.
- TASK_SCHEDULER: The eighth column shows which code generation strategy was used for parallel-for.
- PFOR_MAXGRAINSIZE: The ninth column shows the maximum grainsize used for parallel-for (if IGNORE_USERS_PFORGRAINSIZE is false, then this value does not affect the grainsize set by the user).
- IGNORE_USERS_PFORGRAINSIZE: The tenth column shows whether to ignore the grainsize set by the user or not.
- The rest of the columns indicate the execution time of the benchmark for each run. If there is an error when executing the benchmark, the twelfth column shows what went wrong.

3 LAZYD'S CLAIM.

The following experiment can be used to test our claims:

```
# Medium scale machine
# 8 = number of workers
# 20 = number of runs
./run-eval.sh 8 20 <disable_numa>

# Large scale machine:
./run-eval-big.sh 72 20 <disable_numa>
```

This could take a while to run (3-5 hours). We use the PBBSv2 benchmark as it allows us to modify the grainsize of a parallel-for needed to show our claim. For the Cilk5 benchmark, our results show performance improvements for Fib, Cholesky, and FFT, a performance drop for nqueens and matmul, and similar performance for the rest of the benchmarks. We suspect that the performance drop of nqueens is caused by an increase in the complexity of performing register allocation due to our slow path, which causes sub-optimal register allocation on the hot path.

We used the following machine during our experiment:

```
Architecture:          x86_64
CPU op-mode(s):        32-bit, 64-bit
Byte Order:            Little Endian
CPU(s):                40
On-line CPU(s) list:   0-39
Thread(s) per core:    2
Core(s) per socket:    10
Socket(s):             2
NUMA node(s):          2
Vendor ID:             GenuineIntel
CPU family:            6
Model:                 79
Model name:            Intel(R) Xeon(R) CPU E5-2640 v4 @ 2.40GHz
Stepping:              1
CPU MHz:               3394.335
CPU max MHz:           3400.0000
CPU min MHz:           1200.0000
BogoMIPS:              4794.19
Virtualization:        VT-x
L1d cache:             32K
L1i cache:             32K
L2 cache:              256K
L3 cache:              25600K
```

We do not use hyperthreading during our evaluation (the number of cores used < #PROC/(#threads per CPU) cores). For Opencilk, we enable the worker pinning flag in their runtime source code. For LazyD, we pinned each worker to a core. We noticed that in some machines, this actually degrades the performance of LazyD instead of improving it. We provide a flag `--disable_pinning` that disables the worker pinning in LazyD in case users notice performance degradation using worker pinning.

Users can use our `analyzecsv.py` script to analyze the output of `run-eval.sh`. This can be done using the following command:

```
./analyzecsv.py --ifile=oDir/lazybenchmark_output_files_<datetime>/lazybenchmark_results.csv
```

This will generate a table similar to Table 1 in CSV format. Users can use the `--tex` option to generate a latex table. Note that this script may not work for Cilk5 benchmarks. Alternatively, users can use any app to analyze the CSV files generated by run-eval.sh.

Our claims are as follows:

3.1 LazyD's parallel construct has a smaller overhead compared to OpenCilk's.

On a very fine-grained benchmark where the cost of forks or task-division is significant, our performance of LazyD is better than OpenCilk. This can be observed by comparing the performance of LazyD with OpenCilk.fine where on average (geomean), it is shown that LazyD outperforms OpenCilk.fine (refer to the geomean of LazyD vs Opencilk.fine at Table 1, Table 2, and Table 3).

3.2 Exposing more parallelism using LazyD does not degrade performance on average.

On average, we argue that LazyD's performance is comparable to that of OpenCilk (refer to the geomean of LazyD vs Opencilk.fine at Table 1, Table 2, and Table 3). We acknowledge that there are cases where the performance of LazyD is worse. Some of these benchmark performances can be improved by using LazyD with frequent polling (lazyd2). Further work is still needed to improve LazyD.

3.3 For programs with irregular parallelism, LazyD prevents load imbalance by exposing more parallelism.

One of the benefits of LazyD compared to OpenCilk is that it exposes more parallelism with a small overhead. This prevents load imbalance. This can be observed by comparing the performance of LazyD with OpenCilk on the MIS benchmark. On a smaller core count (see Table 1), LazyD is slightly worse compared to OpenCilk, but as the number of cores increases (see Table 3), it gradually improves over OpenCilk.

3.4 LazyD does not adversely impact the I-Cache miss rate.

LazyD's object size increases up to three times that of OpenCilk's object size. However, we show that this does not degrade the ICache miss rate (I$-MR) compared to OpenCilk. To run the I$-MR experiment, run the following command:

```
# Medium scale machine
./run-icache.sh 8 20 <disable_numa>

# Large scale machine:
./run-icache-big.sh 72 20 <disable_numa>
```

Users can analyze the result using the following command:

```
./analyzecsv.py --ifile=oDir/lazybenchmark_output_files_<datetime>/lazybenchmark_results.csv \
--icache
```

This will generate a table similar to Table 4 in CSV format.

It uses the following perf event: `icache.misses` and `icache.hit`. The last two columns of `lazybenchmark_results.csv` represent the icache misses and icache hits. Our results in Table 4 show that the Icache miss rate between OpenCilk and LazyD is similar.

4 TESTBENCHMARK_COMPILE.PY'S FLAG.

Listing 1 describe the flags available in `testBenchmark_compile.py`.

5 HOW TO COMPILE YOUR OWN CILK CODE.

If users are interested in compiling and testing their own Cilk code, use the following command to compile it:

```
clang -fforkd=lazy -ftapir=serial -mllvm -noinline-tasks=true \
      -mllvm -experimental-debug-variable-locations=false \
      -mllvm -disable-parallelepilog-insidepfor=true \
      -fuse-ld=lld  --opencilk-resource-dir=../../opencilk/cheetah/build/ \
      -Wall -O3  yourcilkcode.c   -o yourcilkcode
```

The following are the options available when using LazyD compiler:

```
      -ftapir=serial -fforkd=lazy :
                    Use LazyD to lower parallel-for and fork-joins.

      -mllvm -noinline-tasks=true :
                    Prevent a parallel task from being inlined in another parallel task.
                    This is to prevent segfault in some of our benchmarks.
                    In the future, this will be fixed.

      -mllvm -experimental-debug-variable-locations=false :
                    In the future, this will be fixed.

      -mllvm -disable-parallelepilog-insidepfor=true :
                    Prevent parallelizing the epilog of loop strip-mined parallel-for.
                    This is to prevent segfault in some of our benchmarks.
                    In the future, this will be fixed.

      # Other available options
      -fpfor-spawn-strategy={1,2} :
                    1: Use divide and conquer (PRC) to lower a cilk_for construct.
                    2: Use PRL to lower a  cilk_for construct.
                    We set this value to 2 in our evaluation.
                    Users can modify this value in pbbsbench/common/parallelDef.

      -mllvm -lazy-disable-unwind-polling={false,true} :
                    Disable polling. Used for measuring performance without polling.

      -mllvm -lazy-enable-proper-polling={0,2} :
                    Controls the polling frequency.
                    0: Polling infrequently.
                    2: Polling frequently.
```

Benchmark	Dataset	Num Cores	OpenCilk(s)	LazyD	OpenCilk.fine
decisionTree-classify	covtype.data	1	7.66	-1.33 %	-0.88 %
deterministicBFS-	rMatGraph-J-12-2250000	1	1.16	-1.57 %	-11.07 %
doubling-lrs	chr22.dna	1	13.31	6.74 %	-7.69 %
histogram-wc	wikipedia250M.txt	1	5.92	-9.24 %	na
histogramStar-wc	wikipedia250M.txt	1	7.19	-5.63 %	-20.83 %
incrementalDelaunay-delaunay	2DinCube-1000000	1	3.17	-2.73 %	-2.89 %
incrementalMIS-MIS	rMatGraph-JR-12-2250000	1	0.15	-5.39 %	-26.27 %
incrementalMatching-matching	rMatGraph-E-12-2250000	1	0.59	-14.90 %	-81.87 %
incrementalRefine-refine	2DkuzminDelaunay-500000	1	4.82	-3.44 %	-6.16 %
incrementalST-ST	rMatGraph-E-12-2250000	1	1.49	-3.04 %	-35.93 %
listRank-bw	trigramString-25000000	1	2.31	0.46 %	-4.87 %
mergeSort-sort	randomSeq-10M-double	1	0.99	-5.22 %	-0.71 %
ndMIS-MIS	rMatGraph-JR-12-2250000	1	0.20	-2.19 %	-8.52 %
ndST-ST	rMatGraph-E-12-2250000	1	0.60	31.77 %	16.24 %
octTree-neighbors	2DinCube-10000000	1	5.84	6.25 %	-1.72 %
parallel-histogram	randomSeq-10M-int	1	0.19	1.00 %	0.43 %
parallel-index	wikipedia250M.txt	1	8.68	1.51 %	-17.58 %
parallelCK-	3DinCube-100000	1	6.27	-10.85 %	-1.48 %
parallelFilterKruskal-MST	rMatGraph-WE-12-2250000	1	2.86	0.01 %	-9.63 %
parallelKS-SA	trigramString-10000000	1	2.20	-6.94 %	-21.40 %
parallelKruskal-MST	rMatGraph-WE-12-2250000	1	5.96	-0.76 %	-11.97 %
parallelRadixSort-isort	randomSeq-10M-int-pair-int	1	0.25	0.06 %	-0.22 %
parallelRange-SA	trigramString-10000000	1	2.31	5.95 %	-14.06 %
parlayhash-dedup	randomSeq-10M-int	1	0.26	-4.43 %	-8.25 %
quickHull-hull	2DinSphere-10000000	1	0.36	-0.30 %	-0.61 %
quickSort-sort	randomSeq-10M-double	1	0.88	0.41 %	1.72 %
sampleSort-sort	randomSeq-10M-double	1	0.91	-2.06 %	-1.65 %
stableSampleSort-sort	randomSeq-10M-double	1	0.87	-7.76 %	-3.58 %
Min				-14.90 %	-81.87 %
Geomean				-1.49 %	-13.64 %
Max				31.77 %	16.24 %

Table 1. *The performance improvement of LazyD and OpenCilk.fine over OpenCilk on the PBBSv2 benchmark at 1 core. The higher the number, the better the performance. A negative number indicates a performance drop. na shows benchmarks that failed to be compiled or executed. For getting the performance of 1 core, we use five runs.*

Benchmark	Dataset	Num Cores	OpenCilk(s)	LazyD	OpenCilk.fine
decisionTree-classify	covtype.data	8	1.12	-0.84 %	-1.34 %
deterministicBFS-	rMatGraph-J-12-2250000	8	0.22	0.50 %	-6.47 %
doubling-lrs	chr22.dna	8	1.94	2.75 %	-13.16 %
histogram-wc	wikipedia250M.txt	8	0.80	-3.51 %	na
histogramStar-wc	wikipedia250M.txt	8	1.09	-3.51 %	-18.74 %
incrementalDelaunay-delaunay	2DinCube-1000000	8	0.63	2.66 %	0.13 %
incrementalMIS-MIS	rMatGraph-JR-12-2250000	8	0.03	4.15 %	-22.62 %
incrementalMatching-matching	rMatGraph-E-12-2250000	8	0.11	-8.52 %	-65.95 %
incrementalRefine-refine	2DkuzminDelaunay-500000	8	0.90	7.15 %	-2.40 %
incrementalST-ST	rMatGraph-E-12-2250000	8	0.27	1.84 %	-29.99 %
listRank-bw	trigramString-25000000	8	0.38	-0.75 %	-4.80 %
mergeSort-sort	randomSeq-10M-double	8	0.14	-4.15 %	-2.38 %
ndMIS-MIS	rMatGraph-JR-12-2250000	8	0.05	1.28 %	-4.30 %
ndST-ST	rMatGraph-E-12-2250000	8	0.09	-1.18 %	-0.82 %
octTree-neighbors	2DinCube-10000000	8	0.86	5.88 %	-4.04 %
parallel-histogram	randomSeq-10M-int	8	0.03	17.23 %	5.30 %
parallel-index	wikipedia250M.txt	8	1.38	10.18 %	-9.77 %
parallelCK-	3DinCube-100000	8	0.91	-12.69 %	0.16 %
parallelFilterKruskal-MST	rMatGraph-WE-12-2250000	8	0.45	1.44 %	-7.97 %
parallelKS-SA	trigramString-10000000	8	0.35	-3.87 %	-16.18 %
parallelKruskal-MST	rMatGraph-WE-12-2250000	8	0.91	-0.54 %	-12.50 %
parallelRadixSort-isort	randomSeq-10M-int-pair-int	8	0.03	-5.00 %	-6.11 %
parallelRange-SA	trigramString-10000000	8	0.34	4.49 %	-8.24 %
parlayhash-dedup	randomSeq-10M-int	8	0.04	7.30 %	17.72 %
quickHull-hull	2DinSphere-10000000	8	0.06	-2.30 %	-0.42 %
quickSort-sort	randomSeq-10M-double	8	0.13	-7.79 %	-3.70 %
sampleSort-sort	randomSeq-10M-double	8	0.13	-0.18 %	1.77 %
stableSampleSort-sort	randomSeq-10M-double	8	0.13	-9.33 %	-2.24 %
Min				-12.69 %	-65.95 %
Geomean				-0.09 %	-9.80 %
Max				17.23 %	17.72 %

Table 2. *The performance improvement of LazyD and OpenCilk.fine over OpenCilk on the PBBSv2 benchmark at 8 cores. The higher the number, the better the performance. A negative number indicates a performance drop. na shows benchmarks that failed to be compiled or executed.*

Benchmark	Dataset	Num Cores	OpenCilk(s)	LazyD	OpenCilk.fine
decisionTree-classify	covtype.data	16	0.64	-3.85 %	-1.03 %
deterministicBFS-	rMatGraph-J-12-2250000	16	na	na	na
doubling-lrs	chr22.dna	16	1.11	1.85 %	-12.28 %
histogram-wc	wikipedia250M.txt	16	0.47	-3.05 %	na
histogramStar-wc	wikipedia250M.txt	16	0.60	-7.83 %	-19.26 %
incrementalDelaunay-delaunay	2DinCube-1000000	16	0.37	2.95 %	1.42 %
incrementalMIS-MIS	rMatGraph-JR-12-2250000	16	0.02	6.91 %	-22.49 %
incrementalMatching-matching	rMatGraph-E-12-2250000	16	0.06	-6.10 %	-57.97 %
incrementalRefine-refine	2DkuzminDelaunay-500000	16	0.51	7.38 %	-0.85 %
incrementalST-ST	rMatGraph-E-12-2250000	16	0.15	0.88 %	-28.71 %
listRank-bw	trigramString-25000000	16	0.20	-1.54 %	-3.88 %
mergeSort-sort	randomSeq-10M-double	16	0.08	-4.55 %	-0.13 %
ndMIS-MIS	rMatGraph-JR-12-2250000	16	0.03	3.87 %	-2.17 %
ndST-ST	rMatGraph-E-12-2250000	16	0.05	0.34 %	1.99 %
octTree-neighbors	2DinCube-10000000	16	0.50	6.03 %	-2.59 %
parallel-histogram	randomSeq-10M-int	16	0.02	18.73 %	8.62 %
parallel-index	wikipedia250M.txt	16	0.84	11.85 %	-7.95 %
parallelCK-	3DinCube-100000	16	0.56	-6.69 %	-1.35 %
parallelFilterKruskal-MST	rMatGraph-WE-12-2250000	16	0.26	2.25 %	-8.53 %
parallelKS-SA	trigramString-10000000	16	0.21	0.20 %	-10.20 %
parallelKruskal-MST	rMatGraph-WE-12-2250000	16	0.54	3.20 %	-5.77 %
parallelRadixSort-isort	randomSeq-10M-int-pair-int	16	0.02	-9.25 %	-0.73 %
parallelRange-SA	trigramString-10000000	16	0.19	1.87 %	-13.59 %
parlayhash-dedup	randomSeq-10M-int	16	0.02	-4.79 %	-3.96 %
quickHull-hull	2DinSphere-10000000	16	0.04	-10.21 %	-10.25 %
quickSort-sort	randomSeq-10M-double	16	0.08	-25.46 %	-4.20 %
sampleSort-sort	randomSeq-10M-double	16	0.08	-0.90 %	-2.45 %
stableSampleSort-sort	randomSeq-10M-double	16	0.07	-9.96 %	-2.59 %
Min				-25.46 %	-57.97 %
Geomean				-1.31 %	-9.32 %
Max				18.73 %	8.62 %

Table 3. *The performance improvement of LazyD and OpenCilk.fine over OpenCilk on the PBBSv2 benchmark at 16 cores. The higher the number, the better the performance. A negative number indicates a performance drop. na shows benchmarks that failed to be compiled or executed.*

Benchmark	Dataset	Num Cores	OpenCilk's i$-MR	LazyD's Δ.i$-MR
decisionTree-classify	covtype.data	8	0.22 %	0.14 %
deterministicBFS-	rMatGraph-J-12-2250000	8	na	na
doubling-lrs	chr22.dna	8	0.74 %	0.16 %
histogram-wc	wikipedia250M.txt	8	0.18 %	0.26 %
histogramStar-wc	wikipedia250M.txt	8	0.43 %	0.05 %
incrementalDelaunay-delaunay	2DinCube-1000000	8	0.77 %	0.16 %
incrementalMIS-MIS	rMatGraph-JR-12-2250000	8	1.39 %	0.34 %
incrementalMatching-matching	rMatGraph-E-12-2250000	8	0.69 %	0.39 %
incrementalRefine-refine	2DkuzminDelaunay-500000	8	0.75 %	0.32 %
incrementalST-ST	rMatGraph-E-12-2250000	8	1.71 %	1.39 %
listRank-bw	trigramString-25000000	8	1.47 %	0.69 %
mergeSort-sort	randomSeq-10M-double	8	0.06 %	0.02 %
ndMIS-MIS	rMatGraph-JR-12-2250000	8	0.11 %	0.20 %
ndST-ST	rMatGraph-E-12-2250000	8	0.65 %	0.32 %
octTree-neighbors	2DinCube-10000000	8	0.04 %	0.07 %
parallel-histogram	randomSeq-10M-int	8	0.09 %	0.06 %
parallel-index	wikipedia250M.txt	8	0.52 %	0.02 %
parallelCK-	3DinCube-100000	8	0.34 %	0.06 %
parallelFilterKruskal-MST	rMatGraph-WE-12-2250000	8	0.22 %	0.04 %
parallelKS-SA	trigramString-10000000	8	0.68 %	0.31 %
parallelKruskal-MST	rMatGraph-WE-12-2250000	8	0.16 %	0.01 %
parallelRadixSort-isort	randomSeq-10M-int-pair-int	8	0.20 %	0.10 %
parallelRange-SA	trigramString-10000000	8	0.38 %	0.08 %
parlayhash-dedup	randomSeq-10M-int	8	0.09 %	0.15 %
quickHull-hull	2DinSphere-10000000	8	0.37 %	0.08 %
quickSort-sort	randomSeq-10M-double	8	0.05 %	0.01 %
sampleSort-sort	randomSeq-10M-double	8	0.14 %	0.03 %
stableSampleSort-sort	randomSeq-10M-double	8	0.10 %	0.01 %
Min				0.01 %
Geomean				0.20 %
Max				1.39 %

Table 4. *Column 4 shows the i$-MR of OpenCilk. Columns 5 show the absolute delta of i$-MR between LazyD with OpenCilk. A smaller delta suggests similar results. na shows benchmarks failed to compile or execute.*

```
-h, --help              Show this help message and exit
--compile               Only compile the benchmark
--num_cores NUM_CORES   [NUM_CORES ...]
                        Number of cores used. Default: 1
--num_tests NUM_TESTS   Number of runs per test
--execute               Only execute benchmark, do not compile
--disable_pinning       Disable worker pinning.
--disable_numa          Do not use numactl --interleave=all when running the benchmark
--icache                Run the icache experiment
--parallel_framework    {lazyd0,lazyd2,nopoll,serial,tapir}
                        [{lazyd0,lazyd2,nopoll,serial,tapir} ...]
                        The parallel framework to use. Default: tapir.

                        lazyd0 = LazyD with infrequent polling (sets env variable POLL0=1)
                        lazyd2 = LazyD with frequent polling   (sets env variable POLL2=1)
                        nopoll = LazyD without polling         (sets env variable NOPOLL=1)
                        serial = Sequential program            (sets env variable SEQUENTIAL=1)
                        tapir = OpenCilk program               (sets env variable OPENCILK=1)

--fg {yes,no,both}      Use finer grainsize. Default: no
                        If --noopt is false, user grainsize does not get affected.
                        Only used by PRC, PRL, DELEGATEPRC, PRCPRL, DELEGATEPRCPRL,
                        and OPENCILKDEFAULT_FINE.  (sets env variable GRAINSIZE8=1)

--noopt {yes,no,both}   Ignore parallel-for's grainsize set by the user. Default: no
                        Only used in PBBS and DELEGATEPRCPRL.  (sets env variable NOOPT=1)

--schedule_tasks        {PRC,PRL,DELEGATEPRC,PRCPRL,DELEGATEPRCPRL,OPENCILKDEFAULT_FINE,PBBS}
                        [{PRC,PRL,DELEGATEPRC,PRCPRL,DELEGATEPRCPRL,OPENCILKDEFAULT_FINE,PBBS} ...]
                        How to schedule parallel task in pfor. Only used for the PBBSv2 benchmarks.

                        PBBS : By default, it uses the PBBS scheduling mechanism.
                        The PBBS default scheduling mechanism uses divide and conquer
                        if the grainsize is not equal to 0.
                        If grainsize is set to 0, it uses cilk_for parallel construct.

                        OPENCILKDEFAULT_FINE: Similar to PBBS.
                        However, if the grainsize is set to 0,
                        the maximum grainsize is set to 8
                        (Default value used by PBBS is 2048).
                        (sets environment variable OPENCILKDEFAULT_FINE=1)

                        PRC: Similar to PBBS, except that we manually
                        lower the cilk_for in the source code using divide and conquer
                        with tail call elimination.
                        (sets environment variable PRC=1)

                        PRL: Use parallel-ready loop to lower the parallel-for.
                        (sets env variable PRL=1)

                        PRCPRL : Uses PRC and then PRL
                        for the remaining iteration. (sets env variable PRCPRL=1)

                        DELEGATEPRC :  Uses Explicit fork and then PRC
                        for the remaining iteration.
                        (sets env variable DELEGATEPRC=1)

                        DELEGATEPRCPRL :  Uses Explicit fork,
                        then PRC, and then PRL for the remaining iteration.
                        (sets environment variable DELEGATEPRCPRL=1)

--ifile IFILE           Input file
-v, --verbose           Verbose
--dryrun                Dry run, only print commands that would be executed
--wait_load WAIT_LOAD   The minimum load before the benchmark can be executed (Default=10)
```