

MUPPET: Optimizing Performance in OpenMP via Mutation Testing

Dolores Miao¹, Ignacio Laguna², Giorgis Georgakoudis²,

Konstantinos Parasyris², Cindy Rubio-González¹

¹University of California, Davis

Davis, CA, USA

{wjmiao,crubio}@ucdavis.edu

²Lawrence Livermore National Laboratory

Livermore, CA, USA

{ilaguna,georgakoudis1,parasyris1}@llnl.gov

Abstract

Performance optimization continues to be a challenge in modern HPC software. Existing performance optimization techniques, including profiling-based and auto-tuning techniques, fail to indicate program modifications at the source level thus preventing their portability across compilers. This paper describes MUPPET, a new approach that identifies program modifications called *mutations* aimed at improving program performance. MUPPET’s mutations help developers reason about performance defects and missed opportunities to improve performance at the source code level. In contrast to compiler techniques that optimize code at intermediate representations (IR), MUPPET uses the idea of source-level *mutation testing* to relax correctness constraints and automatically discover optimization opportunities that otherwise are not feasible using the IR. We demonstrate the MUPPET’s concept in the OpenMP programming model. MUPPET generates a list of OpenMP mutations that alter the program parallelism in various ways, and is capable of running a variety of optimization algorithms such as Bayesian Optimization and delta debugging to find a subset of mutations which, when applied to the original program, cause the most speedup while maintaining program correctness. When MUPPET is evaluated against a diverse set of benchmark programs and proxy applications, it is capable of finding sets of mutations in 70% of the evaluated programs that induce speedup.

CCS Concepts: • Software and its engineering → Software performance; • Computing methodologies → Parallel programming languages.

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).

PMAM '24, March 3, 2024, Edinburgh, United Kingdom

© 2024 Copyright held by the owner/author(s).

ACM ISBN 979-8-4007-0599-1/24/03.

<https://doi.org/10.1145/3649169.3649246>

Keywords: mutation testing, OpenMP, performance optimization, Bayesian optimization, delta-debugging algorithm, dynamic program analysis

ACM Reference Format:

Dolores Miao¹, Ignacio Laguna², Giorgis Georgakoudis², Konstantinos Parasyris², Cindy Rubio-González¹. 2024. MUPPET: Optimizing Performance in OpenMP via Mutation Testing. In *The 15th International Workshop on Programming Models and Applications for Multicores and Manycores (PMAM '24), March 3, 2024, Edinburgh, United Kingdom*. ACM, New York, NY, USA, 10 pages. <https://doi.org/10.1145/3649169.3649246>

1 Introduction

Performance optimization continues to be a challenge in modern HPC software. The adoption of multi-core heterogeneous systems and the use of multi-process and multi-threaded programming models to fully utilize modern architectures are some of the factors that limit the ability of developers to solve performance issues; these issues can result in poor user experience, lower system throughput, limit scalability, and a waste of computational resources [5, 7, 53].

Problems with Existing Techniques. A large amount of work has been proposed to identify performance issues and a number of tools are used in the current HPC production environment to analyze applications’ performance [3, 21, 32, 45]. However, the process of isolating performance problems and/or generating tests to identify them is still mostly a manual process.

Most performance optimization techniques focus on highlighting “hot spots” but ultimately rely on programmers to identify code modifications that fix a performance problem or improve overall performance. Other approaches are based on the concept of quantifying hardware or runtime system events [17, 35, 36], but do not explicitly inform the programmer how to modify the code to improve performance. Compiler optimizations improve performance usually at the intermediate representation (IR) level; however, reasoning about correctness at the IR level is much more difficult than at the source level. As a result, compiler optimizations can leave

optimization opportunities on the table. Moreover, IR-level optimizations are not portable across compilers.

We could potentially solve performance problems given accurate performance models for each available platform and application. If performance models are available, we could simply check if the application’s behavior falls into the bounds of such models. However, such an ideal mechanism is hard to realize in practice as performance models are notoriously difficult to build accurately, given the complexity of the HPC software stack and underlying hardware. There are solutions to build performance models for specific aspects of the hardware and applications [12, 30, 51], but these models are usually not composable and as a result of little practical use in modeling an entire application and platform.

Our Contributions. We present an approach based on *mutation testing* [25] to identify source code changes, or *mutations*, that (1) improve performance, and (2) help developers reason about performance at the source-code level (in contrast to IR- or assembly-level like in existing methods). Since such an approach is based on source modifications, it is portable across compilers.

Mutation testing has been proposed to identify correctness faults [25], and assumes that a syntactic change (a mutant) along with an exploration campaign of multiple mutants can help discover programs’ defects faster than traditional methods. While some previous work has applied mutation testing to solve performance defects [10], mutation testing for performance has not been applied on parallel code and/or HPC programs. We demonstrate our approach in the OpenMP programming model, which is widely used in HPC.

We implement our approach in the framework named MUPPET (Mutation-Utilized Parallel Performance Enhancement Tester). First, MUPPET generates a list of OpenMP mutations that could alter the program performance in various ways. A mutation is defined as a change in an existing OpenMP directive in the program that could change the performance of the code block that the directive targets. MUPPET considers only mutations that are not likely to change the correctness of the code block. Next, MUPPET considers different optimization algorithms, such as Bayesian optimization (BO) [34] and delta debugging [56], to find a subset of mutations that, when applied to the original program, cause the highest speedup. We implement MUPPET in the clang/LLVM front-end and evaluate it in the NAS Parallel Benchmarks [31] and three proxy applications (LULESH [26], HPGC [14], CoMD [18]).

In summary, our contributions are:

- We present a source-level approach that uses mutation testing to optimize HPC code. Our approach considers four classes of source mutations and applies them in OpenMP directives. *To the best of our knowledge, we are the first to explore using mutation testing to optimize OpenMP code* (Section 3).

- We design and implement our idea in the MUPPET framework via the clang/LLVM front-end. Our approach integrates MUPPET with several optimization algorithms, such as BO and delta debugging. The output of MUPPET is a set of source modifications, or mutations, that produce a maximum speedup among the explored mutations, without affecting correctness (Section 4).
- We evaluate MUPPET on several benchmarks and proxy applications. We demonstrate that MUPPET is capable of identifying mutations that improve performance in 70% of the evaluated programs, with the best speedup in average running time of 15.64% (Section 5).

2 Overview

In this section, we describe the philosophy of our approach, provide background information on mutation testing, and provide a simple mutation example in a matrix multiply kernel that improves performance.

2.1 Approach’s Philosophy

Existing approaches to isolate performance issues are difficult to use in practice. A number of performance problems can be fixed by changes in the source code; however, existing methods do not directly point to developers’ source modifications that fix such issues. Compilers optimize code at the IR level but such solutions are not portable across compilers and make it harder to reason about correctness than solutions based on source modifications.

We believe that tools and techniques for performance optimization should have the following features:

- **Fine granularity detection:** tools should pinpoint, with fine granularity, the location (code line) of performance issues or potential performance improvements.
- **Guided fixes:** the approach should help programmers understand and reason about performance defects—without a good understanding, it is hard to solve the problem or avoid it in the future.
- **Automatic recommendations:** the approach should automatically suggest code modifications that improve performance or fix a performance problem.

We designed MUPPET using the above criteria to identify changes in OpenMP directives that improve performance.

2.2 Mutation Testing for Performance

2.2.1 Challenges. The key idea of MUPPET is to perform small changes in the code, called *mutations*, and use exploratory algorithms to search for cases where mutations improve performance or fix a performance problem. Mutation testing has been studied before to detect faulty programs by injecting small syntactical changes that expose correctness defects [25]. The idea of mutation testing is to generate sufficient data to expose real software defects in the code.

However, it is challenging to use traditional mutation testing in isolating performance defects because the syntactic changes could create faults, i.e., breaking the semantics of the program and producing incorrect programs.

2.2.2 Our Solution. Inspired by the previous work on mutation testing, we propose a different approach: *to inject only mutations that are semantically correct and do not yield an incorrect program for the purpose of exposing performance defects or speedup opportunities*. Semantically correct mutants, or *equivalent mutants*, are considered problematic for traditional mutation testing because by definition, they cannot fail the test suite, so they should be avoided to increase the effectiveness of mutation testing. In contrast, our approach explores semantically correct mutations, or a weaker form of mutations that successfully pass correctness tests, to identify any mutations that increase performance, thus indicating performance defects.

2.3 Mutation Example

Here, we present a synthetic matrix-multiplication example, shown in Listing 1, that demonstrates MUPPET’s capabilities—when we apply MUPPET, it can find a set of mutations that yields faster code execution.

Listing 1. Example code with a mutation found by MUPPET that improves performance.

```

1 #define ARRAY_SIZE (2048)
2 double A[ARRAY_SIZE][ARRAY_SIZE];
3 double B[ARRAY_SIZE][ARRAY_SIZE];
4 double C[ARRAY_SIZE][ARRAY_SIZE];
5
6 int main(void) {
7     // initialize array and timer setup omitted
8     float var = 2.3f;
9     #pragma omp parallel for shared(var)
10    // mutation adds an OpenMP directive
11    #pragma omp tile sizes(16,16,16)
12    for (int i = 0; i < ARRAY_SIZE; ++i)
13        for (int j = 0; j < ARRAY_SIZE; ++j)
14            for (int k = 0; k < ARRAY_SIZE; ++k) {
15                C[i][j] += var*A[i][k]*B[k][j];
16            }
17    // end processing omitted
18 }
```

Originally, the code has only the OpenMP `parallel` directive to parallelize the loop. Then, MUPPET applies mutations to the existing OpenMP directives found in the code. Note that while MUPPET only considers semantically correct mutations (and are likely to produce a correct program), it relies on existing correctness checks of the program, as shown in Section 5.1.1 for the evaluated programs. When we run MUPPET on this example with delta debugging, after 20 tryouts, MUPPET reports a mutation that, when applied to the program, improves performance. With BO, it takes

66 tryouts to finish the optimization process; but the mutation was reported with 11 tryouts. The identified mutation is highlighted in the source code. In this simple example, the mutation is the addition of the OpenMP tile construct, which tiles one or more loops. In the end, MUPPET reports to the developer that adding this construct to the loop introduces a 18.84x speedup, from 7.116801 seconds to 0.377674 seconds.

3 Approach

3.1 Problem Statement

Given an OpenMP program P with running time T , MUPPET analyzes the program and generates a set of mutations, $M = \{m_1, m_2, \dots, m_n\}$, which potentially could induce program speedup. We define the program running time for the original variant program as:

$$T = P(\emptyset)$$

We define the running time for a variant program as:

$$T' = P(M'), \text{ where } M' \subseteq M, \text{ and } \text{accurate}(P, M') = \text{True}.$$

We define the ideal minimum program running time as:

$$\begin{aligned} T_{min} &= P(M_{min}), \\ \text{where } M_{min} &\subseteq M, \\ \text{and } \text{accurate}(P, M_{min}) &= \text{True}, \\ \text{and } \forall M' \subseteq M, T' &\geq T_{min} \end{aligned}$$

The goal of MUPPET is find a subset of M , $M_{min'}$, with $T_{min'}$ as close to T_{min} as possible.

3.2 Tool Workflow.

The overall workflow of MUPPET is illustrated in Figure 1. The purposes of these modules are described below:

- *Mutation generator* analyses the program and finds a set of source code mutations, which can potentially be applied to change the OpenMP parallelism of the program.
- *Transformer* generates a program variant with a subset of mutations found in the *Mutation generator* module.
- *Tester* runs the mutated programs from *Transformer* and tests the performance speedup and correctness of the mutated variant.
- *Optimizer* applies a user-specified optimization algorithm to find the minimum of the function $T' = P(M')$.

Next, we delve into the details of these modules, following the order as they appear in Figure 1.

3.3 Mutation Generator

The Mutation Generator module traverses the abstract syntax tree (AST) of the program, looking for source code locations that potentially can be mutated so that program parallelism is changed. The time complexity of this step is $O(n)$ where n is the number of statement nodes on the AST.

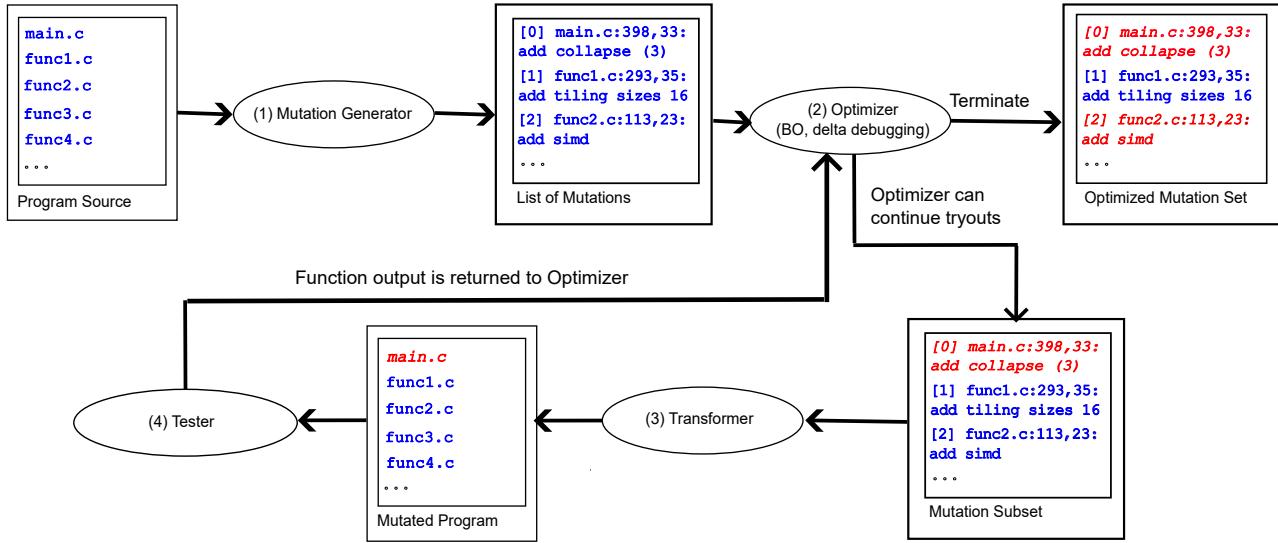


Figure 1. The workflow of MUPPET. *Red texts in italic* indicates the mutation is applied, or the source file is changed.

The mutators in MUPPET focus on mutating parallel/loop OpenMP constructs such as the `parallel` directive, `for` directive, or the `parallel for` directive. All of these directives specify a source code region to be executed in parallel, but the parallelism may not be high enough to utilize all available cores for the OpenMP program. It also looks for the beginning of `for` loops for SIMD mutations.

3.3.1 Mutation Classes. There are four types of mutations possible to apply to certain source code locations:

1. **Collapse Mutations** add a `collapse` clause to a multiple dimension `parallel for` loop. Collapse clauses may potentially improve parallelism by having more iterations, thus higher hardware thread usage, at the top level of the loop.
2. **SIMD Mutations** add a `simd` clause to an OpenMP parallelism-related directive such as a `parallel for` loop, or a `omp for` loop. SIMD clauses or directives hint at the compiler to check if there is a possibility to vectorize the loops and apply SIMD vectorization if possible.
3. **Tiling Mutations** add `tile` directives at the top of a multiple-dimension OpenMP loop. Tile directives split the loop space into smaller-sized "tiles", and each tile is ideally only accessed by one OpenMP thread. This design can potentially improve cache locality depending on how the data within memory is accessed within the loops, and thus may also introduce performance speedup. Due to the difficulty in determining loop size at compile time, MUPPET only supports setting a fixed set of differently sized tiles as different mutations. For example, we can only set the tile size as a power of

8, 16, or 32. Given the limitations, users can still see from the optimization results whether using a smaller or larger-sized tile can have a higher speedup.

4. **Firstprivate Mutations** put read-only shared variables into a `firstprivate` clause for an OpenMP parallel region, in order to reduce dependency between parallel threads.

Once such language constructs (`parallel for`, `for`, etc.) are detected, the Mutation Generator module will then check the associated source code around the current language construct. If the source code around it satisfies certain statically defined criteria (see below), then unique information regarding the current mutation, such as source location, the way source code is modified (insert before, insert after, modify), and the mutation type, is added to the list of mutations. The algorithm for this process is shown in Algorithm 1.

3.3.2 Criteria Selection. The criteria for each type of mutation simply follows the syntax of OpenMP language specifications. These criteria can be customized for any new type of mutations added. Here are some examples: "collapse" mutations are identified by an OpenMP directive followed by a rectangular, nested loop, within which there is no jump statements such as `break`, `continue` or `return`; "simd" and "tiling" mutations are identified by a serial or parallel loop statement without OpenMP parallel constructs or jump statements inside; lastly, every variable inside an OpenMP parallel region is checked for eligibility to become `firstprivate` variables. Some of the OpenMP mutations that can be applied to in the previously shown *matmul* example are shown in Figure 2. The one that shows the highest speedup in *matmul* is the tiling mutation.

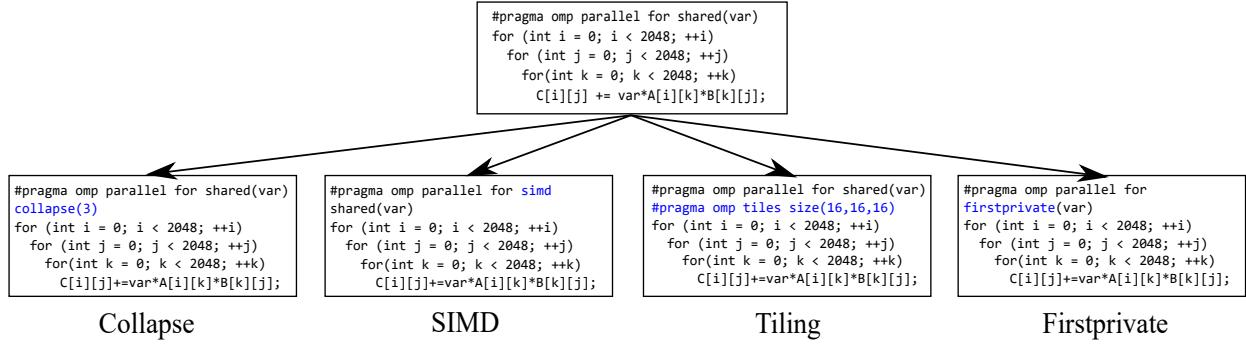


Figure 2. Classes of mutations in MUPPET.

Algorithm 1: The mutation generator algorithm.

```

1  Function GenerateMutations(StatementList):
2      M =  $\emptyset$ ;
3      foreach Statement in StatementList do
4          if Statement is an OpenMP directive then
5              if can add collapse mutation then
6                  M = M  $\cup$  CollapseMutation(Statement);
7
8              if can add SIMD mutation then
9                  M = M  $\cup$  SIMDMutation(Statement);
10
11             if can add tiling mutation then
12                 M = M  $\cup$  TilingMutation(Statement);
13
14             if can add firstprivate mutation then
15                 M = M  $\cup$  FirstprivateMutation(Statement);
16
17
18         if Statement is a for loop then
19             if can add SIMD mutation then
20                 M = M  $\cup$  SIMDMutation(Statement);
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65
66
67
68
69
70
71
72
73
74
75
76
77
78
79
80
81
82
83
84
85
86
87
88
89
90
91
92
93
94
95
96
97
98
99
100
101
102
103
104
105
106
107
108
109
110
111
112
113
114
115
116
117
118
119
120
121
122
123
124
125
126
127
128
129
130
131
132
133
134
135
136
137
138
139
140
141
142
143
144
145
146
147
148
149
150
151
152
153
154
155
156
157
158
159
160
161
162
163
164
165
166
167
168
169
170
171
172
173
174
175
176
177
178
179
180
181
182
183
184
185
186
187
188
189
190
191
192
193
194
195
196
197
198
199
200
201
202
203
204
205
206
207
208
209
210
211
212
213
214
215
216
217
218
219
220
221
222
223
224
225
226
227
228
229
230
231
232
233
234
235
236
237
238
239
240
241
242
243
244
245
246
247
248
249
250
251
252
253
254
255
256
257
258
259
259
260
261
262
263
264
265
266
267
268
269
269
270
271
272
273
274
275
276
277
278
279
279
280
281
282
283
284
285
286
287
287
288
289
289
290
291
292
293
294
295
296
297
297
298
299
299
300
301
302
303
304
305
306
307
308
309
309
310
311
312
313
314
315
316
317
318
319
319
320
321
322
323
324
325
326
327
328
329
329
330
331
332
333
334
335
336
337
338
339
339
340
341
342
343
344
345
346
347
348
349
349
350
351
352
353
354
355
356
357
358
359
359
360
361
362
363
364
365
366
367
368
369
369
370
371
372
373
374
375
376
377
378
379
379
380
381
382
383
384
385
386
387
388
389
389
390
391
392
393
394
395
396
397
398
399
399
400
400
401
402
403
404
405
406
407
408
409
409
410
411
412
413
414
415
416
417
418
419
419
420
421
422
423
424
425
426
427
428
429
429
430
431
432
433
434
435
436
437
438
439
439
440
441
442
443
444
445
446
447
448
449
449
450
451
452
453
454
455
456
457
458
459
459
460
461
462
463
464
465
466
467
468
469
469
470
471
472
473
474
475
476
477
478
479
479
480
481
482
483
484
485
486
487
488
489
489
490
491
492
493
494
495
496
497
497
498
499
499
500
500
501
502
503
504
505
506
507
508
509
509
510
511
512
513
514
515
516
517
518
519
519
520
521
522
523
524
525
526
527
528
529
529
530
531
532
533
534
535
536
537
538
539
539
540
541
542
543
544
545
546
547
548
549
549
550
551
552
553
554
555
556
557
558
559
559
560
561
562
563
564
565
566
567
568
569
569
570
571
572
573
574
575
576
577
578
579
579
580
581
582
583
584
585
586
587
588
589
589
590
591
592
593
594
595
596
597
597
598
599
599
600
601
602
603
604
605
606
607
608
609
609
610
611
612
613
614
615
616
617
618
619
619
620
621
622
623
624
625
626
627
628
629
629
630
631
632
633
634
635
636
637
638
639
639
640
641
642
643
644
645
646
647
648
649
649
650
651
652
653
654
655
656
657
658
659
659
660
661
662
663
664
665
666
667
668
669
669
670
671
672
673
674
675
676
677
678
679
679
680
681
682
683
684
685
686
687
688
688
689
690
691
692
693
694
695
696
697
697
698
699
699
700
701
702
703
704
705
706
707
708
709
709
710
711
712
713
714
715
716
717
718
719
719
720
721
722
723
724
725
726
727
728
729
729
730
731
732
733
734
735
736
737
738
739
739
740
741
742
743
744
745
746
747
748
749
749
750
751
752
753
754
755
756
757
758
759
759
760
761
762
763
764
765
766
767
768
769
769
770
771
772
773
774
775
776
777
778
779
779
780
781
782
783
784
785
786
787
788
788
789
790
791
792
793
794
795
796
796
797
798
799
799
800
801
802
803
804
805
806
807
808
809
809
810
811
812
813
814
815
816
817
818
819
819
820
821
822
823
824
825
826
827
828
829
829
830
831
832
833
834
835
836
837
838
839
839
840
841
842
843
844
845
846
847
848
849
849
850
851
852
853
854
855
856
857
858
859
859
860
861
862
863
864
865
866
867
868
869
869
870
871
872
873
874
875
876
877
878
879
879
880
881
882
883
884
885
886
887
888
888
889
889
890
891
892
893
894
895
896
896
897
898
899
899
900
901
902
903
904
905
906
907
908
909
909
910
911
912
913
914
915
916
917
918
919
919
920
921
922
923
924
925
926
927
928
929
929
930
931
932
933
934
935
936
937
938
939
939
940
941
942
943
944
945
946
947
948
948
949
950
951
952
953
954
955
956
957
958
959
959
960
961
962
963
964
965
966
967
968
969
969
970
971
972
973
974
975
976
977
978
979
979
980
981
982
983
984
985
986
986
987
988
989
989
990
991
992
993
994
995
995
996
997
997
998
999
999
1000
1000
1001
1002
1003
1004
1005
1006
1007
1008
1009
1009
1010
1011
1012
1013
1014
1015
1016
1017
1018
1019
1019
1020
1021
1022
1023
1024
1025
1026
1027
1028
1029
1029
1030
1031
1032
1033
1034
1035
1036
1037
1038
1038
1039
1040
1041
1042
1043
1044
1045
1046
1047
1048
1048
1049
1050
1051
1052
1053
1054
1055
1056
1057
1058
1059
1059
1060
1061
1062
1063
1064
1065
1066
1067
1068
1069
1069
1070
1071
1072
1073
1074
1075
1076
1077
1078
1079
1079
1080
1081
1082
1083
1084
1085
1086
1087
1088
1088
1089
1089
1090
1091
1092
1093
1094
1095
1096
1096
1097
1098
1098
1099
1099
1100
1101
1102
1103
1104
1105
1106
1107
1108
1109
1109
1110
1111
1112
1113
1114
1115
1116
1117
1118
1119
1119
1120
1121
1122
1123
1124
1125
1126
1127
1128
1129
1129
1130
1131
1132
1133
1134
1135
1136
1137
1138
1138
1139
1140
1141
1142
1143
1144
1145
1146
1147
1147
1148
1149
1149
1150
1151
1152
1153
1154
1155
1156
1157
1158
1158
1159
1160
1161
1162
1163
1164
1165
1166
1166
1167
1168
1168
1169
1169
1170
1171
1172
1173
1174
1175
1176
1176
1177
1178
1178
1179
1179
1180
1181
1182
1183
1184
1185
1185
1186
1186
1187
1188
1188
1189
1189
1190
1191
1192
1193
1194
1195
1195
1196
1196
1197
1198
1198
1199
1199
1200
1201
1202
1203
1204
1205
1206
1207
1208
1209
1209
1210
1211
1212
1213
1214
1215
1216
1217
1217
1218
1219
1219
1220
1221
1222
1223
1224
1225
1226
1227
1227
1228
1229
1229
1230
1231
1232
1233
1234
1235
1236
1236
1237
1238
1238
1239
1239
1240
1241
1242
1243
1244
1245
1246
1246
1247
1248
1248
1249
1249
1250
1251
1252
1253
1254
1255
1256
1256
1257
1258
1258
1259
1259
1260
1261
1262
1263
1264
1265
1266
1266
1267
1268
1268
1269
1269
1270
1271
1272
1273
1274
1275
1275
1276
1277
1277
1278
1278
1279
1279
1280
1281
1282
1283
1284
1285
1285
1286
1286
1287
1288
1288
1289
1289
1290
1291
1292
1293
1294
1295
1295
1296
1296
1297
1298
1298
1299
1299
1300
1301
1302
1303
1304
1305
1306
1307
1308
1309
1309
1310
1311
1312
1313
1314
1315
1316
1316
1317
1318
1318
1319
1319
1320
1321
1322
1323
1324
1325
1326
1326
1327
1328
1328
1329
1329
1330
1331
1332
1333
1334
1335
1335
1336
1336
1337
1338
1338
1339
1339
1340
1341
1342
1343
1344
1345
1345
1346
1347
1347
1348
1348
1349
1349
1350
1351
1352
1353
1354
1355
1355
1356
1356
1357
1358
1358
1359
1359
1360
1361
1362
1363
1364
1365
1365
1366
1367
1367
1368
1368
1369
1369
1370
1371
1372
1373
1374
1375
1375
1376
1376
1377
1378
1378
1379
1379
1380
1381
1382
1383
1384
1385
1385
1386
1386
1387
1388
1388
1389
1389
1390
1391
1392
1393
1394
1395
1395
1396
1396
1397
1398
1398
1399
1399
1400
1401
1402
1403
1404
1405
1405
1406
1406
1407
1408
1408
1409
1409
1410
1411
1412
1413
1414
1415
1415
1416
1416
1417
1418
1418
1419
1419
1420
1421
1422
1423
1424
1425
1425
1426
1426
1427
1428
1428
1429
1429
1430
1431
1432
1433
1434
1435
1435
1436
1436
1437
1438
1438
1439
1439
1440
1441
1442
1443
1444
1445
1445
1446
1446
1447
1448
1448
1449
1449
1450
1451
1452
1453
1454
1455
1455
1456
1456
1457
1458
1458
1459
1459
1460
1461
1462
1463
1464
1465
1465
1466
1466
1467
1468
1468
1469
1469
1470
1471
1472
1473
1474
1475
1475
1476
1476
1477
1478
1478
1479
1479
1480
1481
1482
1483
1484
1485
1485
1486
1486
1487
1488
1488
1489
1489
1490
1491
1492
1493
1494
1495
1495
1496
1496
1497
1498
1498
1499
1499
1500
1501
1502
1503
1504
1505
1505
1506
1506
1507
1508
1508
1509
1509
1510
1511
1512
1513
1514
1515
1515
1516
1516
1517
1518
1518
1519
1519
1520
1521
1522
1523
1524
1525
1525
1526
1526
1527
1528
1528
1529
1529
1530
1531
1532
1533
1534
1535
1535
1536
1536
1537
1538
1538
1539
1539
1540
1541
1542
1543
1544
1545
1545
1546
1546
1547
1548
1548
1549
1549
1550
1551
1552
1553
1554
1555
1555
1556
1556
1557
1558
1558
1559
1559
1560
1561
1562
1563
1564
1565
1565
1566
1566
1567
1568
1568
1569
1569
1570
1571
1572
1573
1574
1575
1575
1576
1576
1577
1578
1578
1579
1579
1580
1581
1582
1583
1584
1585
1585
1586
1586
1587
1588
1588
1589
1589
1590
1591
1592
1593
1594
1595
1595
1596
1596
1597
1598
1598
1599
1599
1600
1601
1602
1603
1604
1605
1605
1606
1606
1607
1608
1608
1609
1609
1610
1611
1612
1613
1614
1615
1615
1616
1616
1617
1618
1618
1619
1619
1620
1621
1622
1623
1624
1625
1625
1626
1626
1627
1628
1628
1629
1629
1630
1631
1632
1633
1634
1635
1635
1636
1636
1637
1638
1638
1639
1639
1640
1641
1642
1643
1644
1645
1645
1646
1646
1647
1648
1648
1649
1649
1650
1651
1652
1653
1654
1655
1655
1656
1656
1657
1658
1658
1659
1659
1660
1661
1662
1663
1664
1665
1665
1666
1666
1667
1668
1668
1669
1669
1670
1671
1672
1673
1674
1675
1675
1676
1676
1677
1678
1678
1679
1679
1680
1681
1682
1683
1684
1685
1685
1686
1686
1687
1688
1688
1689
1689
1690
1691
1692
1693
1694
1695
1695
1696
1696
1697
1698
1698
1699
1699
1700
1701
1702
1703
1704
1705
1705
1706
1706
1707
1708
1708
1709
1709
1710
1711
1712
1713
1714
1715
1715
1716
1716
1717
1718
1718
1719
1719
1720
1721
1722
1723
1724
1725
1725
1726
1726
1727
1728
1728
1729
1729
1730
1731
1732
1733
1734
1735
1735
1736
1736
1737
1738
1738
1739
1739
1740
1741
1742
1743
1744
1745
1745
1746
1746
1747
1748
1748
1749
1749
1750
1751
1752
1753
1754
1755
1755
1756
1756
1757
1758
1758
1759
1759
1760
1761
1762
1763
1764
1765
1765
1766
1766
1767
1768
1768
1769
1769
1770
1771
1772
1773
1774
1775
1775
1776
1776
1777
1778
1778
1779
1779
1780
1781
1782
1783
1784
1785
1785
1786
1786
1787
1788
1788
1789
1789
1790
1791
1792
1793
1794
1795
1795
1796
1796
1797
1798
1798
1799
1799
1800
1801
1802
1803
1804
1805
1805
1806
1806
1807
1808
1808
1809
1809
1810
1811
1812
1813
1814
1815
1815
1816
1816
1817
1818
1818
1819
1819
1820
1821
1822
1823
1824
1825
1825
1826
1826
1827
1828
1828
1829
1829
1830
1831
1832
1833
1834
1835
1835
1836
1836
1837
1838
1838
1839
1839
1840
1841
1842
1843
1844
1845
1845
1846
1846
1847
1848
1848
1849
1849
1850
1851
1852
1853
1854
1855
1855
1856
1856
1857
1858
1858
1859
1859
1860
1861
1862
1863
1864
1865
1865
1866
1866
1867
1868
1868
1869
1869
1870
1871
1872
1873
1874
1875
1875
1876
1876
1877
1878
1878
1879
1879
1880
1881
1882
1883
1884
1885
1885
1886
1886
1887
1888
1888
1889
1889
1890
1891
1892
1893
1894
1895
1895
1896
1896
1897
1898
1898
1899
1899
1900
1901
1902
1903
1904
1905
1905
1906
1906
1907
1908
1908
1909
1909
1910
1911
1912
1913
1914
1915
1915
1916
1916
1917
1918
1918
1919
1919
1920
1921
1922
1923
1924
1925
1925
1926
1926
1927
1928
1928
1929
1929
1930
1931
1932
1933
1934
1935
1935
1936
1936
1937
1938
1938
1939
1939
1940
1941
1942
1943
1944
1945
1945
1946
1946
1947
1948
1948
1949
1949
1950
1951
1952
1953
1954
1955
1955
1956
1956
1957
1958
1958
1959
1959
1960
1961
1962
1963
1964
1965
1965
1966
1966
1967
1968
1968
1969
1969
1970
1971
1972
1973
1974
1975
1975
1976
1976
1977
1978
1978
1979
1979
1980
1981
1982
1983
1984
1985
1985
1986
1986
1987
1988
1988
1989
1989
1990
1991
1992
1993
1994
1995
1995
1996
1996
1997
1998
1998
1999
1999
2000
2001
2002
2003
2004
2005
2005
2006
2006
2007
2008
2008
2009
2009
2010
2011
2012
2013
2014
2015
2015
2016
2016
2017
2018
2018
2019
2019
2020
2021
2022
2023
2024
2025
2025
2026
2026
2027
2028
2028
2029
2029
2030
2031
2032
2033
2034
2035
2035
2036
2036
2037
2038
2038
2039
2039
2040
2041
2042
2043
2044
2045
2045
2046
2046
2047
2048
2048
2049
2049
2050
2051
2052
2053
2054
2055
2055
2056
2056
2057
2058
2058
2059
2059
2060
2061
2062
2063
2064
2065
2065
2066
2066
2067
2068
2068
2069
2069
2070
2071
2072
2073
2074
2075
2075
2076
2076
2077
2078
2078
2079
2079
2080
2081
2082
2083
2084
2085
2085
2086
2086
2087
2088
2088
2089
2089
2090
2091
2092
2093
2094
2095
2095
2096
2096
2097
2098
2098
2099
2099
2100
2101
2102
2103
2104
2105
2105
2106
2106
2107
2108
2108
2109
2109
2110
2111
2112
2113
2114
2115
2115
2116
2116
2117
2118
2118
2119
2119
2120
2121
2122
2123
2124
2125
2125
2126
2126
2127
2128
2128
2129
2129
2130
2131
2132
2133
2134
2135
2135
2136
2136
2137
2138
2138
2139
2139
2140
2141
2142
2143
2144
2145
2145
2146
2146
2147
2148
2148
2149
2149
2150
2151
2152
21
```

3.4 Optimizer

Once a list of mutations is generated, it is exported to the Optimizer. This module runs an optimization algorithm specified by the end user to find the minimum point of $T' = P(M')$. During the optimization process, it finds specific points on the $T' = P(M')$ function by selecting/deselecting a subset of mutations, sending these mutations to the Transformer and Tester module, and receiving T' from the Transformer and Tester module once the mutated program has finished execution and running time statistics are collected.

MUPPET supports two optimization algorithms: Bayesian Optimization (BO) [34] and delta debugging [56]. The goal of these algorithms, albeit vastly different in implementation, is the same: find the subset of source mutations that would introduce maximum speedup. We selected BO because it is a common optimization algorithm that does not have the assumption of the function forms, which makes it an appropriate algorithm to use in MUPPET. Delta debugging, on the other hand, was originally developed as a software testing algorithm to isolate bugs inside a program, which is then adapted into finding speedup in program variants in previous work such as Precimonious [44] with regards to precision

tuning. The inclusion in MUPPET of both algorithms shows how algorithms with vastly different original purposes can solve the same problem in different ways. MUPPET can also be extended to support other optimization algorithms such as genetic algorithm or simulated annealing.

For BO, since the input parameter of the function to be optimized, $T' = P(M')$, is a subset of mutations, which does not fit the function format of BO, we optimize $T' = P(Mb')$ instead where:

$$Mb' = \{mb_1, mb_2, \dots, mb_n\}$$

$$mb_i = \begin{cases} 1, & \text{if } m_i \in M' \\ 0, & \text{otherwise} \end{cases}$$

In this way, we convert the subset parameter into a list of binary parameters signaling whether a mutation is included in the subset so that BO can accept this list as input parameters for the function it optimizes.

As for delta debugging, we follow the LCCSEARCH algorithm in [44], where a change set in our adaptation of the algorithm is defined as the set of mutations that are applied to the original program, and the outputs are a minimal change set which causes speedup.

3.5 Transformer and Tester

The Transformer and Tester modules read the list of mutations from the Optimizer module, mutate the program into a variant, and run the variant to see if there is any speedup while maintaining the correctness of the program.

3.5.1 Compilation and Conflicts Checks. Even though there are already criteria placed in the Mutation Generator module for each mutation type to ensure that all mutations generated are syntactically correct, there are still situations where different mutations, when applied to the same programs at the same time, cause conflicts between them. If MUPPET lets these conflicts pass without checking during the transformer phase, it will cause a large number of mutated program variants that do not compile.

In order to save execution time, when the module transforms the program, it also statically checks and circumvents certain conflicts. These conflict checks can also be customized in the case where new types of mutations are implemented or new conflicts are discovered during testing. Currently, the conflict checks include: no tiling directives should be inside a SIMD region; and no SIMD directives or clauses should be inside a tile or collapse region.

4 Implementation Details

MUPPET is implemented with a variety of programming languages and toolsets. The Mutation Generator and the Transformer modules are implemented via Clang plugins. Clang plugin system is one of several systems in the Clang compiler architecture that are capable of performing source-to-source code transformation, along with libtooling and libclang. Clang plugin is used so that our code transformation runs alongside the build environment of the evaluated programs, with the same kind of dependency checks. MUPPET only requires minimal changes to the build scripts for it to work on new programs. This is described in 4.3.

The Optimizer and Tester modules, and the overarching framework managing the communication between modules, on the other hand, are implemented in Python. This is done to leverage the existence of a mature set of Python numerical optimization modules such as scikit-optimize [24].

4.1 Language Support

MUPPET uses the modular approach; each of the three modules can be replaced in order to implement an analogous functionality. Currently MUPPET targets C/C++ programs with OpenMP language constructs, though it is possible to target FORTRAN programs by rewriting the Mutation Generator and Transformer modules with a source-to-source FORTRAN compiler such as ROSE [40].

4.2 Customizing MUPPET Runtime Parameters

MUPPET supports BO and delta debugging in our implementation. BO is implemented with scikit-optimize, while delta debugging is implemented from scratch using the algorithm described in Precimonious [44], since it has no publicly available Python implementations.

Since running time for each program run may have variations that should not be counted as speedup, in order to suppress such variations, users can customize MUPPET parameters to change how it measures running time. The *times* parameter specifies MUPPET to run a number of repetitions for each variant, and collect running times for each run; the *shuffle* switch, only available for delta debugging, randomly shuffle the order of mutations so that delta debugging algorithm partitions these mutations differently each time (users can still specify the same random seed for the same shuffle result). Lastly, users can choose between using the minimum

running time in all repetitions as program running time, or use the average running time.

4.3 Integrating New Programs with MUPPET

For better management of programs in evaluation, MUPPET calls a customized version of the FAROS build system [22]. MUPPET calls a variety of functionalities offered in FAROS in order to analyze, transform, build and run the specified program. With FAROS, it is easy to add new programs to be mutated by simply adding new entries into the YAML config file.

4.3.1 Entries and Correctness. An example entry for a locally stored simple matrix multiplication program is shown in Listing 2. It sets up commands for each step used in MUPPET, such as building, calling plugins for mutations, running the program, extracting running time statistics from program output, and cleaning. The only required change to the *matmul* source code is (a) modify the build scripts (Makefile in this case) so that it accepts parameters for calling the Clang plugins; and (b) add correctness check code that parses program output in order to determine if the mutated program still runs correctly.

Listing 2. YAML config file for *matmul*.

```

1 matmul: fetch: 'cp -r ../../extra/matmul .'
2 build_dir: 'matmul'
3 build: {
4     omp: ['make CC=clang++ OPT_LEVEL=3 OMP=1'],
5 }
6 call_plugin: {
7     analysis: ['make func_analysis OMP=1'],
8     mutate: ['make trans_mutations OMP=1'],
9 }
10 copy: ['matmul']
11 bin: 'matmul'
12 run: './matmul'
13 input: ''
14 measure: 'Work consumed (\d+\.\d+) seconds'
15 clean: 'rm -r *.*; cp ../../extra/
matmul/*.* .'

```

5 Experimental Evaluation

This experimental evaluation answers the following research questions:

RQ1 Does MUPPET discover source code mutations that induce speedup for OpenMP programs?

RQ2 What are the factors that may determine the efficacy of MUPPET in finding these source code mutations?

5.1 Evaluation Setup

5.1.1 Benchmarks. We use a set of 10 C/C++ OpenMP programs to evaluate MUPPET. The programs include benchmark programs such as NPB-CPP [31] and HPCG [14], and proxy applications such as LULESH [26] and CoMD [18]. We

Table 1. Mutation speedup discovered by delta debugging and Bayesian Optimization.

(a) Delta debugging

Program	Original Min.	Best Min.	Speedup	Original Avg.	Best Avg.	Avg. Speedup	No. Possible Mutations	No. Mutations in Best (collapse/simd/firstprivate/tile)
LULESH	11.095s	10.644s	4.23%	11.161s	10.741s	3.91%	0/95/0/222	0/7/0/8
HPCG	15.598s	14.071s	10.85%	15.654s	15.231s	2.78%	0/63/13/81	0/1/0/0
CoMD	2.296s	2.210s	3.90%	2.304s	2.232s	3.22%	0/78/13/132	0/25/4/7
FT.A	1.288s	1.244s	3.54%	1.305s	1.267s	3.06%	1/42/5/45	0/3/1/1
LU.A	4.933s	4.867s	1.35%	4.956s	4.890s	1.35%	3/100/6/186	0/1/0/1
MG.A	3.592s	3.124s	14.99%	3.603s	3.131s	15.07%	7/66/8/39	1/5/1/2
SP.A	31.534s	29.813s	5.77%	32.001s	30.948s	3.40%	64/267/3/396	0/6/0/4
BT.A	42.620s	42.344s	0.65%	42.686s	42.365s	0.75%	44/218/2/381	7/32/0/20
CG.B	22.432s	22.262s	0.76%	22.701s	22.376s	1.45%	0/18/11/27	0/1/1/1
EP.B	6.245s	6.238s	0.11%	6.251s	6.243s	0.13%	0/9/1/24	0/9/1/8

(b) Bayesian Optimization

Program	Original Min.	Best Min.	Speedup	Original Avg.	Best Avg.	Avg. Speedup	No. Possible Mutations	No. Mutations in Best (collapse/simd/firstprivate/tile)
LULESH	11.080s	10.790s	2.69%	11.136s	10.881s	2.34%	0/95/0/222	0/47/0/62
HPCG	15.577s	13.599s	14.54%	15.693s	14.584s	7.61%	0/63/13/81	0/26/4/26
CoMD	2.300s	2.225s	3.36%	2.310s	2.254s	2.47%	0/78/13/132	0/37/10/37
FT.A	1.287s	1.261s	2.11%	1.295s	1.265s	2.37%	1/42/5/45	1/23/3/12
LU.A	4.906s	4.812s	1.94%	4.940s	4.872s	1.40%	3/100/6/186	2/46/2/54
MG.A	3.593s	3.132s	14.74%	3.624s	3.134s	15.64%	7/66/8/39	6/35/3/10
SP.A	30.819s	32.563s	-5.36%	31.330s	33.274s	-5.84%	64/267/3/396	28/138/2/118
BT.A	42.607s	43.370s	-1.76%	42.709s	43.437s	-1.68%	44/218/2/381	26/114/2/111
CG.B	22.504s	22.391s	0.51%	22.590s	22.466s	0.55%	0/18/11/27	0/9/6/9
EP.B	6.244s	6.237s	0.10%	6.247s	6.243s	0.07%	0/9/1/24	0/4/0/7

use these programs in order to evaluate the efficacy of MUPPET in finding speedup in different programs, on a reference implementation or on manually optimized code.

On the benchmarks side, NPB-CPP is the C++ version of NAS Parallel Benchmarks ported to various programming frameworks on shared-memory architectures including OpenMP. We use 7 benchmark programs in varying problem sizes for evaluation: BT.A, CG.B, EP.B, FT.A, LU.A, MG.A, SP.A. HPCG is a benchmark program that performs multi-grid preconditioned conjugate gradient iterations. We run it with a grid size of 96*96*96. All benchmarks contain result verification routines in their source code, so we use them in order to determine program correctness.

On the proxy applications side, LULESH is a proxy application simulating the Shock Hydrodynamics Challenge Problem, while CoMD is a proxy application implementing classical molecular dynamics algorithms and workloads as used in materials science. Evaluating these programs may show the efficacy of MUPPET in helping software developers in scientific computing optimize the parallel performance of their programs. LULESH is run with the parameter `-i 1500 -s 35`, and CoMD with `-e -i 1 -j 1 -k 1 -x 20 -y 20 -z 20`. We use the approach presented in [28] to determine the correctness of the program. For LULESH, we consider

iteration count, final origin energy, and TotalAbsDiff as the output; for CoMD, we use the final energy as output.

5.1.2 Algorithm Parameters. We use both BO and delta debugging in our experimental evaluation. Given the fact that program running time varies across the programs being evaluated, we put a tryout limit of 100 on both algorithms instead of using a total time limit. Our parameters for BO are $n_calls = 100$, $n_initial_points = 10$, and $noise = 0.01$.

5.1.3 Evaluation Environment. We use a workstation computer with two 14-core Intel Xeon E5-2694v3 CPUs and 32GiB of RAM, running Ubuntu 22.04. We use Clang 16.0.6 with OpenMP 5.1 support as the compiler for both source-to-source code transformation, and for building and running the evaluated programs. Using OpenMP 5.1 enables us to build programs with collapse clauses as well.

We also ensure that performance variation is minimized between program runs. We avoid CPU context switching by limiting the programs to run on hardware threads on the second CPU by forcing the taskset `-c 14-27` command in FAROS. Hardware quiescing, as defined by [1], is also performed to reduce performance fluctuations, such as turning off both simultaneous multithreading and dynamic frequency scaling.

As for running time statistic collection, we run each mutated variant 5 times and use the minimum running time

as the program running time T . As a comparison, we also record the average running time for each tryout and evaluate if there is any possible discrepancy between average and minimum running time, but this statistic is not used as the fitness function output for optimization algorithms. We use the minimum running time for the fitness function because as stated in [1] it is best at rejecting noise introduced by the evaluation environment, since running time higher than the minimum must be due to such noise. However, we still calculate speedup for average running time to see how performance variability affects running time.

5.2 Evaluation Results

Even though we have taken various measures to reduce performance variability between each program run, it is still a factor that is not completely removed. Therefore, to determine if a program shows speedup when mutated, we use the 1% threshold. If amongst the 5 runs, the speedup between the minimum running time or between the average running time is lower than 1%, then the current subset of mutations is discarded.

The results of both algorithms can be found in Table 1. We compare the minimum running time for the mutated program against the minimum of the original (columns 2-4), and its average running time against the average running time of the original program (columns 5-7). Our evaluation shows that there are 7 out of 10 evaluated programs in which delta debugging can find a subset of mutations that, when applied, can cause speedup while maintaining the correctness of the program. The other 3 programs below the horizontal line in Table 1 show no speedup.

The speedup with regards to the minimum time ranges from 1.35% in LU to 14.99% in MG. Meanwhile, the speedup with regards to average time is generally about the same or smaller than the speedup with regards to the minimum time, especially so in HPCG where the speedup in average time is only 2.78% compared to the 10.85% speedup in minimum time. Such a drop in speedup is likely from increased inherent performance variability introduced by the sole mutation discovered. Running these programs more than 5 times, or further static program analysis, may be needed to determine a more robust speedup result. BO on the other hand can find mutation subsets that cause speedup in only 6 out of 10 programs, as it cannot find such a subset for SP. Furthermore, the speedup discovered is not greater than delta debugging except HPCG, which shows 7.61% in average running time speedup and 14.54% in minimum running time speedup.

We have also recorded the number of mutations that are applied in the subsets that cause the highest speedup with both algorithms (column 9 in Table 1), compared to the total number of possible mutations (column 8). The results in BO all have more mutations in the subsets, except EP which neither algorithm shows speedup. When we investigate all tryouts and their running times in programs such as SP, we

deduce that a lot of mutations in these programs cause negative speedup, while only a few cause positive speedup. BO works worse in programs like these compared to delta debugging because it takes more tryouts than delta debugging to remove mutations with negative speedups from consideration. On the other hand, programs like HPCG likely have a few mutations that cause large speedup, but most others do not cause negative speedup. In these cases, BO works better than delta debugging and can find a subset of mutations that contain mutations with both large and small speedup.

6 Related Work

Mutation Testing. Mutation testing has been proposed to identify correctness defects [25]. The assumption in mutation testing is that a syntactic change (a mutant) can help discover programs' defects. Mutation testing, however, has not been applied deeply in HPC programs and on performance defects. Some attempts to build mutation testing for cloud systems have been reported [8]. Mutation operators (i.e., syntactic changes) have been proposed to reveal faults in small-size MPI programs [46]. With the increased use of LLVM, researchers are exploring the support of mutation testing in LLVM [11]. To the best of our knowledge, the only work that considers mutation testing for performance is [10]. However, this work does not consider parallelism and mutations in numerical (floating-point) code—these two aspects are critical to HPC applications. To the best of our knowledge, we are the first to explore using mutation testing for performance in OpenMP scientific codes.

General Auto-tuning. There is a significant corpus of past work on auto-tuning techniques. Typical examples include ATLAS [50], Active Harmony [48], FFTW [19], POET [54], CHILL [9], GEIST [49], OpenTuner [4], CLTune [38], Apollo [6, 52], and Dutta et al. [15, 16]. Their common theme is that they tune compile-time, such as tiling, or runtime parameters, such as the number of threads, presupposing a given source code representation of a program. Typical search algorithms for tuning they propose include random, grid, or Bayesian search, or various machine learning-based search models. By contrast, MUPPET mutates the source code of the program, which exposes a large, combined set of both source code modifications as compile-time parameters and their possible configurations as runtime parameters to tune for. Furthermore, MUPPET automates the generation of tuned source code variants without user intervention and it is the first to propose the delta debugging search algorithm for tuning. Integrating machine-learning techniques for fast searching in MUPPET is an interesting future extension.

A number of papers research domain-specific tuning using code generation, alternate data layouts, or algorithmic parameters, such as [2, 13, 20, 27, 37] for linear algebra kernels and [33, 41, 42, 55] for stencils. Those approaches require users to express the programs in specialized domain-specific

languages amenable to tuning, which limits their generality. MUPPET tunes unaltered, user-provided, general OpenMP code to generate tuning source code variants and optimizing runtime parameters.

Auto-tuning OpenMP. Specifically on OpenMP, Adaptive OpenMP [23, 29], Sreenivasan et al. [47] propose OpenMP language extensions to support auto-tuning on OpenMP regions, such as scheduling policies of parallel loops, number of threads or teams. Those approaches require significant refactoring of the code and domain-specific knowledge from the programmer to successfully integrate tuning extensions and their possible configuration parameters in their OpenMP code. Instead, MUPPET treats source code modifications as a tunable parameter and independently explores the runtime configuration space.

Bliss [43] proposes probabilistic Bayesian optimization to tune hardware (core frequency, hyperthreading) and software execution parameters (OpenMP threads, algorithmic alternatives) for the whole application, specified by the user. Bliss does not modify the program’s source code and tunes all regions in unison, by contrast, MUPPET both enables source code modifications and specializes tuning to each region, since mutations are region-specific.

Scalable Record-Replay [39] is a mechanism that extracts the LLVM IR of OpenMP GPU target region kernels to tune for each kernel in parallel the GPU launch bounds as compile-time parameters, by modifying the IR to re-compile, and the number of threads/teams as runtime parameters. Performing the kind of mutations in MUPPET on LLVM IR is challenging compared to source code, which motivates our choice of a source code mutation tool. Nevertheless, the idea of extracting OpenMP regions and tuning them independently is a possible extension to MUPPET to speed up search time.

7 Conclusion

We presented MUPPET, a novel application of mutation testing aimed at improving the performance of OpenMP programs. MUPPET uses different search algorithms to apply and compose program mutations to reduce application execution time. Because program transformations are performed at the source level, MUPPET’s mutations are transferable across different OpenMP implementations and compilers. We demonstrate that MUPPET is capable of identifying mutations that improve performance in 70% of the evaluated programs achieving a maximum average speedup of 15.64%.

In the future, we plan to extend MUPPET to automatically update OpenMP code bases with the latest OpenMP features that improve performance while maintaining correctness. Currently, it is the responsibility of the code maintainer to manually update their code base to use newly available OpenMP features, which require significant manual efforts. The source code and data of MUPPET are publicly available at <https://github.com/LLNL/MUPPET/>.

Acknowledgments

This work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344 (LLNL-CONF-858593), the U.S. Department of Energy, Office of Science, Advanced Scientific Computing Research, under award DE-SC0022182, and the National Science Foundation under award CCF-2119348.

References

- [1] 2018. Performance Engineering of Software Systems. <https://ocw.mit.edu/courses/6-172-performance-engineering-of-software-systems-fall-2018/>
- [2] Walid A. Abu-Sufah and Asma Abdel Karim. 2013. Auto-tuning of Sparse Matrix-Vector Multiplication on Graphics Processors. In *ISC (Lecture Notes in Computer Science)*, Vol. 7905. Springer, 151–164.
- [3] Laksono Adhianto, S. Banerjee, Michael W. Fagan, et al. 2010. HPC-TOOLKIT: tools for performance analysis of optimized parallel programs. *Concurr. Comput. Pract. Exp.* 22, 6 (2010), 685–701.
- [4] Jason Ansel, Shoail Kamil, Kalyan Veeramachaneni, et al. 2014. OpenTuner: an extensible framework for program autotuning. In *PACT*. ACM, 303–316.
- [5] Md. Abul Kalam Azad, Nafees Iqbal, Foyzul Hassan, et al. 2023. An Empirical Study of High Performance Computing (HPC) Performance Bugs. In *MSR*. IEEE, 194–206.
- [6] David Beckingsale, Olga Pearce, Ignacio Laguna, et al. 2017. Apollo: Reusable Models for Fast, Dynamic Tuning of Input-Dependent Code. In *IPDPS*. IEEE Computer Society, 307–316.
- [7] Alexandru Calotoiu, Torsten Hoefler, Marius Poke, et al. 2013. Using automated performance modeling to find scalability bugs in complex codes. In *SC*. ACM, 45:1–45:12.
- [8] Pablo C. Cañizares, Alberto Núñez, and Mercedes G. Merayo. 2018. Mutomvo: Mutation testing framework for simulated cloud and HPC environments. *J. Syst. Softw.* 143 (2018), 187–207.
- [9] Chun Chen, Jacqueline Chame, and Mary Hall. 2008. *CHiLL: A framework for composing high-level loop transformations*. Technical Report. Citeseer.
- [10] Pedro Delgado-Pérez, Ana Belén Sánchez, Sergio Segura, et al. 2020. Performance mutation testing. *Software Testing, Verification and Reliability* (2020), e1728.
- [11] Alex Denisov and Stanislav Pankevich. 2018. Mull It Over: Mutation Testing Based on LLVM. In *ICST Workshops*. IEEE Computer Society, 25–31.
- [12] Nan Ding and Samuel Williams. 2019. An Instruction Roofline Model for GPUs. In *PMBS@SC*. IEEE, 7–18.
- [13] Jack J. Dongarra, Mark Gates, Jakub Kurzak, et al. 2018. Autotuning Numerical Dense Linear Algebra for Batched Computation With GPU Hardware Accelerators. *Proc. IEEE* 106, 11 (2018), 2040–2055.
- [14] Jack J. Dongarra, Michael A. Heroux, and Piotr Luszczek. 2016. High-performance conjugate-gradient benchmark: A new metric for ranking high-performance computing systems. *Int. J. High Perform. Comput. Appl.* 30, 1 (2016), 3–10.
- [15] Akash Dutta, Jordi Alcaraz, Ali TehraniJamsaz, et al. 2023. Performance Optimization using Multimodal Modeling and Heterogeneous GNN. In *HPDC*. ACM, 45–57.
- [16] Akash Dutta, Jordi Alcaraz, Ali TehraniJamsaz, et al. 2022. Pattern-based Autotuning of OpenMP Loops using Graph Neural Networks. In *AI4S*. IEEE, 26–31.
- [17] Alexandre E Eichenberger, John Mellor-Crummey, Martin Schulz, et al. 2013. OMPT: An OpenMP tools application programming interface for performance analysis. In *International Workshop on OpenMP*. Springer, 171–185.

[18] The Exascale Co-Design Center for Materials in Extreme Environments (ExMatEx). 2013. CoMD - Classical molecular dynamics proxy application. <https://github.com/ECP-copa/CoMD>.

[19] Matteo Frigo and Steven G Johnson. 2005. The design and implementation of FFTW3. *Proc. IEEE* 93, 2 (2005), 216–231.

[20] Mark Gates, Jakub Kurzak, Piotr Luszczek, et al. 2017. Autotuning Batch Cholesky Factorization in CUDA with Interleaved Layout of Matrices. In *IPDPS Workshops*. IEEE Computer Society, 1408–1417.

[21] Markus Geimer, Felix Wolf, Brian JN Wylie, et al. 2010. The Scalasca performance toolset architecture. *Concurrency and Computation: Practice and Experience* 22, 6 (2010), 702–719.

[22] Giorgis Georgakoudis, Johannes Doerfert, Ignacio Laguna, et al. 2020. FAROS: A Framework to Analyze OpenMP Compilation Through Benchmarking and Compiler Optimization Analysis. In *JWOMP (Lecture Notes in Computer Science)*, Vol. 12295. Springer, 3–17.

[23] Giorgis Georgakoudis, Konstantinos Parasyris, Chunhua Liao, et al. 2023. Machine Learning-Driven Adaptive OpenMP For Portable Performance on Heterogeneous Systems. [arXiv:cs.PL/2303.08873](https://arxiv.org/abs/cs.PL/2303.08873)

[24] Tim Head, Manoj Kumar, Holger Nahrstaedt, et al. 2021. scikit-optimize/scikit-optimize.

[25] Yue Jia and Mark Harman. 2010. An analysis and survey of the development of mutation testing. *IEEE transactions on software engineering* 37, 5 (2010), 649–678.

[26] Ian Karlin, Abhinav Bhatele, Jeff Keasler, et al. 2013. Exploring Traditional and Emerging Parallel Programming Models Using a Proxy Application. In *IPDPS*. IEEE Computer Society, 919–932.

[27] Jakub Kurzak, Hartwig Anzt, Mark Gates, et al. 2016. Implementation and Tuning of Batched Cholesky Factorization and Solve for NVIDIA GPUs. *IEEE Trans. Parallel Distributed Syst.* 27, 7 (2016), 2036–2048.

[28] Ignacio Laguna, Paul C. Wood, Ranvijay Singh, et al. 2019. GPUMixer: Performance-Driven Floating-Point Tuning for GPU Scientific Applications. In *ISC (Lecture Notes in Computer Science)*, Vol. 11501. Springer, 227–246.

[29] Chunhua Liao, Daniel J Quinlan, Richard Vuduc, et al. 2009. Effective source-to-source outlining to support whole program empirical optimization. In *International Workshop on Languages and Compilers for Parallel Computing*. Springer, 308–322.

[30] Yu Jung Lo, Samuel Williams, Brian Van Straalen, et al. 2014. Roofline model toolkit: A practical tool for architectural and program analysis. In *International Workshop on Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems*. Springer, 129–148.

[31] Júnior Löff, Dalvan Griebler, Gabriele Mencagli, et al. 2021. The NAS Parallel Benchmarks for evaluating C++ parallel programming frameworks on shared-memory architectures. *Future Gener. Comput. Syst.* 125 (2021), 743–757.

[32] Diogo Marques, Helder Duarte, Aleksandar Ilic, et al. 2017. Performance Analysis with Cache-Aware Roofline Model in Intel Advisor. In *HPCS*. IEEE, 898–907.

[33] Kazuaki Matsumura, Hamid Reza Zohouri, Mohamed Wahib, et al. 2020. AN5D: automated stencil framework for high-degree temporal blocking on GPUs. In *CGO*. ACM, 199–211.

[34] Jonas Mockus. 1994. Application of Bayesian approach to numerical methods of global and stochastic optimization. *J. Glob. Optim.* 4, 4 (1994), 347–365.

[35] Philip J Mucci, Shirley Browne, Christine Deane, et al. 1999. PAPI: A portable interface to hardware performance counters. In *Proceedings of the department of defense HPCMP users group conference*, Vol. 710. Citeseer.

[36] Hitoshi Nagasaka, Naoya Maruyama, Akira Nukada, et al. 2010. Statistical power modeling of GPU kernels using performance counters. In *International conference on green computing*. IEEE, 115–122.

[37] Rajib Nath, Stanimire Tomov, Jack Dongarra, et al. 2010. Autotuning dense linear algebra libraries on gpus and overview of the magma library. In *6th International Workshop on Parallel Matrix Algorithms and Applications (PMAA'10)*.

[38] Cedric Nugteren and Valeriu Codreanu. 2015. CLTune: A generic auto-tuner for OpenCL kernels. In *2015 IEEE 9th International Symposium on Embedded Multicore/Many-core Systems-on-Chip*. IEEE, 195–202.

[39] Konstantinos Parasyris, Giorgis Georgakoudis, Esteban Rangel, et al. 2023. Scalable Tuning of (OpenMP) GPU Applications via Kernel Record and Replay. In *SC*. ACM, 28:1–28:14.

[40] Dan Quinlan and Chunhua Liao. 2011. The ROSE source-to-source compiler infrastructure. In *Cetus users and compiler infrastructure workshop, in conjunction with PACT*, Vol. 2011. Citeseer, 1.

[41] Prashant Singh Rawat, Miheer Vaidya, Aravind Sukumaran-Rajam, et al. 2018. Domain-Specific Optimization and Generation of High-Performance GPU Code for Stencil Computations. *Proc. IEEE* 106, 11 (2018), 1902–1920. <https://doi.org/10.1109/JPROC.2018.2862896>

[42] Prashant Singh Rawat, Miheer Vaidya, Aravind Sukumaran-Rajam, et al. 2019. On Optimizing Complex Stencils on GPUs. In *IPDPS*. IEEE, 641–652.

[43] Rohan Basu Roy, Tirthak Patel, Vijay Gadepally, et al. 2021. Bliss: auto-tuning complex applications using a pool of diverse lightweight learning models. In *PLDI*. ACM, 1280–1295.

[44] Cindy Rubio-González, Cuong Nguyen, Hong Diep Nguyen, et al. 2013. Precimorous: tuning assistant for floating-point precision. In *SC*, William Gropp and Satoshi Matsuoka (Eds.). ACM, 27.

[45] Sameer S Shende and Allen D Malony. 2006. The TAU parallel performance system. *The International Journal of High Performance Computing Applications* 20, 2 (2006), 287–311.

[46] Rodolfo Adamshuk Silva, Simone da Rocio Senger de Souza, and Paulo Sergio Lopes de Souza. 2012. Mutation operators for concurrent programs in MPI. In *2012 13th Latin American Test Workshop (LATW)*. IEEE, 1–6.

[47] Vinu Sreenivasan, Rajath Javali, Mary Hall, et al. 2019. A framework for enabling OpenMP autotuning. In *International Workshop on OpenMP*. Springer, 50–60.

[48] Cristian Tapus, I-Hsin Chung, and Jeffrey K. Hollingsworth. 2002. Active Harmony: Towards Automated Performance Tuning. In *Proceedings of the 2002 ACM/IEEE Conference on Supercomputing (SC '02)*. IEEE Computer Society Press, Washington, DC, USA, 1–11.

[49] Jayaraman J. Thiagarajan, Nikhil Jain, Rushil Anirudh, et al. 2018. Bootstrapping Parameter Space Exploration for Fast Tuning. In *ICS*. ACM, 385–395.

[50] R Clinton Whaley and Jack J Dongarra. 1998. Automatically tuned linear algebra software. In *SC'98: Proceedings of the 1998 ACM/IEEE conference on Supercomputing*. IEEE, 38–38.

[51] Samuel Williams, Andrew Waterman, and David Patterson. 2009. Roofline: an insightful visual performance model for multicore architectures. *Commun. ACM* 52, 4 (2009), 65–76.

[52] Chad Wood, Giorgis Georgakoudis, David Beckingsale, et al. 2021. Artemis: Automatic Runtime Tuning of Parallel Execution Parameters Using Machine Learning. In *ISC (Lecture Notes in Computer Science)*, Vol. 12728. Springer, 453–472.

[53] Yi Yang, Ping Xiang, Mike Mantor, et al. 2012. Fixing performance bugs: An empirical study of open-source GPGPU programs. In *2012 41st International Conference on Parallel Processing*. IEEE, 329–339.

[54] Qing Yi, Keith Seymour, Haishang You, et al. 2007. POET: Parameterized optimizations for empirical tuning. In *2007 IEEE International Parallel and Distributed Processing Symposium*. IEEE, 1–8.

[55] Xin You, Hailong Yang, Zhonghui Jiang, et al. 2021. DRStencil: Exploiting Data Reuse within Low-order Stencil on GPU. In *HPCC/DSS/SmartCity/DependSys*. IEEE, 63–70.

[56] Andreas Zeller and Ralf Hildebrandt. 2002. Simplifying and Isolating Failure-Inducing Input. *IEEE Trans. Software Eng.* 28, 2 (2002), 183–200.