SAND2020-7142C

Evaluating the Efficiency of OpenMP Tasking for Unbalanced Computation on Diverse CPU Architectures

Stephen L. Olivier[0000−0001−6247−8980]

Center for Computing Research, Sandia National Laboratories
Albuquerque, NM, USA, slolivi@sandia.gov

Abstract. In the decade since support for task parallelism was incorporated into OpenMP, its use has remained limited in part due to concerns about its performance and scalability. This paper revisits a study from the early days of OpenMP tasking that used the Unbalanced Tree Search (UTS) benchmark as a stress test to gauge implementation efficiency. The present UTS study includes both Clang/LLVM and vendor OpenMP implementations on four different architectures. We measure parallel efficiency to examine each implementation's performance in response to varying task granularity. We find that most implementations achieve over 90% efficiency using all available cores for tasks of $O(100k)$ instructions, and the best even manage tasks of $O(10k)$ instructions well.

Keywords: OpenMP Tasks · Unbalanced Tree Search · Load balancing.

1 Introduction

The introduction of asynchronous task parallelism was the primary focus of version 3.0 of the OpenMP® API specification published in 2008 [24]. Subsequent versions of the specification up to and including version 5.0 [25] have added numerous enhancements to the OpenMP tasking model. Tasking has carved out an important role in OpenMP as the mechanism for asynchronous device offload, but its use remains somewhat limited in CPU-only OpenMP programs. Common concerns include finding the optimal task granularity to amortize the overhead costs of task creation, scheduling, and synchronization while at the same time exposing sufficient application parallelism.

Shortly after the first OpenMP 3.0 implementations appeared, the Unbalanced Tree Search Benchmark (UTS) [20] was ported to the OpenMP tasking model as a stress test [21]. The OpenMP tasking version of UTS was initially compared against an OpenMP version that handled load balancing at user level and a Cilk [10] version. An expanded study [22] included comparisons to Cilk++ [15] (forerunner of Intel® Cilk™ Plus) and Threading Building Blocks (TBB) [26]. The results shed light on the ability of runtime systems of the time to cope with large numbers of tasks generated in an unpredictable manner. UTS was later added to the Barcelona OpenMP Tasks Suite (BOTS)[1].

[1] https://github.com/bsc-pm/bots

At the time of this writing, the UTS OpenMP tasking studies are just over a decade old. They were carried out on a board comprised of eight dual-core "Santa Rosa" Opteron processors manufactured in a 90nm feature size. In contrast, Intel and AMD are currently transitioning down from 14nm to smaller feature sizes, and processors with up to 72 cores per chip have been deployed in high performance computing (HPC) systems. Arm systems capable of 64-bit server-class computing were not even available until recently. Compilers and runtime systems have evolved in the intervening years as well. Many are now based on the LLVM project [14], and its permissively licensed open-source code base has enabled cooperation among vendors and researchers while still allowing vendors to maintain custom versions with proprietary optimizations for added value. In addition, the evolution of the OpenMP tasking model since its inception has required some changes to the data structures and algorithms used in implementations. In light of these developments in hardware and software, the time is ripe to reprise the UTS stress testing evaluation of OpenMP tasking.

We do not attempt to reproduce exactly the earlier UTS OpenMP tasking studies. The problem size used then is much too small for current systems, and only one machine was used. The present study explores the following dimensions:

- Diversity of architectures (IBM POWER9, Arm Thunder X2, Intel Xeon Skylake, and Intel Xeon Phi Knights Landing);
- Comparison of Clang/LLVM and vendor implementations;
- Measuring parallel efficiency as a function of task granularity;
- Quantifying load balancing operations per thread per unit time.

This effort aims to offer insights into the present state of OpenMP tasking efficiency for the benefit of OpenMP users and implementors.

2 UTS: The Benchmark and Its Implementation

The UTS benchmark is a traversal of a dynamically generated tree. The end result is a count of all the tree nodes. Since the computation of the result does not require storage of tree nodes already explored, it is possible to generate and process massive problems on even a small system. A variety of tree types are specified in the original UTS paper [20], but this study is confined to the "binomial" tree type, which is particularly challenging to load balance due to its unpredictability. In particular, simply distributing nodes near the root of the tree across threads is not sufficient, because some of those nodes produce very few descendants and the size of the subtree rooted at any node in the tree is not known *a priori*. Rather, continuous dynamic load balancing is required. Though the benchmark itself is synthetic, it is representative of applications that perform an exhaustive search of a large irregular state space.

The key benchmark parameters are the root branching factor (b_0), the non-root branching factor (m), and the probability of generating children (q). At the start of the benchmark only the root node of the tree exists. After the b_0

child nodes of the root are generated, each of those nodes and each of their descendants determine the number of children to generate by sampling a binomial probability distribution. Each non-root node has m children with probability q and no children with probability $1 - q$. Each node is identified by a 20-byte descriptor. The generation of a child executes a SHA-1 cryptographic hash [9] on the combination of the parent's descriptor and its child index. Thus, successful completion of the benchmark on an n-node tree requires n SHA-1 evaluations.

An additional parameter is useful for the present study, compute granularity (g). This parameter specifies the number of times to repeat the SHA-1 hash at each node and it defaults to 1. Repeating the hash does not change the result of the computation, but it changes the amount of work done at each node. The effect of increasing the compute granularity is to coarsen the tasks.

The version of UTS used in this study is based on the implementation in the Barcelona OpenMP Tasks Suite (BOTS). The code for the recursive function that performs the tree traversal is shown in Figure 1. The code in the figure includes some minor simplifications, but it also shows one substantive change from the BOTS version that has been made to the actual code run in the experiments. That change is the **if** statement that ensures recursive calls are only made in the case where a child node itself has children.[2] This change aids analysis of the benchmark by ensuring that all tasks do the same number of SHA-1 hash operations (including only those performed within that task itself, not its descendants). Recall that each node has m children with probability q and no children with probability $1 - q$. Thus, each task performs exactly $m \times g$ hash operations, where g is the compute granularity parameter specifying the number of repetitions of the hash operation for each node. In the original BOTS version, the **if** statement was not present, and tasks were created for child nodes that themselves produced no children and thus did no SHA-1 hash operations.[3]

3 Test Problem

The problem input used in the study is tree "T3S", found in the small.input file in the inputs/uts subdirectory of the BOTS distribution. The root node of the tree has $b_0 = 2000$ child nodes. Each non-root nodes has $m = 5$ children with probability $q = 0.200014$ and no childen with probability $(1 - q) = 0.799986$. The resulting tree has 111 345 631 nodes and with a maximum depth of 17 844 nodes. Only 22 268 727 nodes (19.99964% of the total nodes) have children, while the remaining 89 076 904 nodes (80.00036% of the total nodes) have no children. These numbers match closely the expected number of nodes with no children based on the parameterized bias of the probability distribution given by q and

[2] An **if** clause on the **task** construct would still create a task, though it would be undeferred. The combination of **final** and **mergeable** clauses would allow but not require that child tasks be merged, and it would require additional look-ahead since the parent task must also be final to enable merging of the child tasks.

[3] The version used in the 2009 UTS OpenMP tasking study [21] also had uniform work per task, but with each task performing the SHA-1 hash for only a single node.

```
unsigned long long search(Node *parent, int numChildren)
{
  Node n[numChildren], *nodePtr;
  int i, j;
  unsigned long long subtreesize = 1, partialCount[numChildren];

  // Recurse on the children of Node
  for (i = 0; i < numChildren; i++) {
    nodePtr = &n[i];

    // The following line is the work (one or more SHA-1 ops)
    for (j = 0; j < granularity; j++) {
      sha1_rng(parent->state.state, nodePtr->state.state, i);
    }

    // Sample a binomial distribution to determine the number of children of child i
    nodePtr->numChildren = uts_numChildren(nodePtr);

    if (nodePtr->numChildren > 0) {
      // Traverse the subtree rooted at child i to get subtree size
      #pragma omp task untied firstprivate(i, nodePtr) shared(partialCount)
        partialCount[i] = search(nodePtr, nodePtr->numChildren);
    }
    else
      partialCount[i] = 1;
  }

  // Wait for all subtree traversals
  #pragma omp taskwait

  // Combine subtree counts from children to get total size of subtree rooted at Node
  for (i = 0; i < numChildren; i++) {
    subtreesize += partialCount[i];
  }

  return subtreesize;
}
```

Fig. 1. UTS code

$(1-q)$. Since the parallel code used in the experiments is structured to create one task per child-producing node, 22 268 727 is also the number of OpenMP tasks. The compute granularity is varied in the experiments, but where not specified explicitly it is only one SHA-1 hash operation per tree node.

Each experiment consisted of ten trials. Perhaps due in part to effective load balancing, percent standard deviation was no more than 2% and in most cases a fraction of a percent. Hence, error bars are omitted from the graphs.

4 Experimental Setup

The present study spans four different architectures and 2-3 OpenMP implementations per architecture:

- **Xeon SKL**: Intel® Xeon® "Skylake" Platinum 8160 Processors, dual socket with 24 cores per socket (48 cores total), 2 hardware threads per core, 2.1 GHz, 192 GB DDR4 memory, Red Hat® Enterprise Linux® 7.1. *Compilers:* Intel® C/C++ Compiler 19.0.5 using "-fopenmp -O3 -xHost"; Clang LLVM 9.0.1 using "-fopenmp -O3 -march=native" with LLVM OpenMP Runtime.
- **IBM P9**: IBM® POWER9™ 8335-GTW Processors, dual socket with 22 cores per socket (44 cores total), 4 hardware threads per core, 2.3 GHz, 256GB DDR4 memory, Red Hat® Enterprise Linux® 7.6. *Compilers:* PGI® Compiler 20.1 using "-mp -O3 -tp=pwr9"; Clang LLVM 9.0.1 using "-fopenmp -O3 -mcpu=pwr9" with LLVM OpenMP Runtime.
- **Arm TX2**: Marvell® ThunderX2® CN9975-2000 Arm® v8 Processors, dual socket with 28 cores per socket (56 cores total), 2 hardware threads per core[4] , 2.0 GHz, 128 GB DDR4 memory, Tri-Lab Operating System Stack (TOSS) based on Red Hat® Enterprise Linux® 7.6. *Compilers:* Arm® Compiler 20.0 ("armclang") using "-fopenmp -O3 -mcpu=native"; Clang LLVM 9.0.1 using "-fopenmp -O3 -mcpu=native" with LLVM OpenMP Runtime.
- **Xeon Phi**: Intel® Xeon Phi™ "Knights Landing" 7250 Processor, single socket with 68 cores, 1.4 GHz, 4 hardware threads per core, 16GB Multi-Channel MCDRAM on-package memory, 96 GB DDR4 memory, Cray Linux® Environment (CLE) based on SUSE Linux® Enterprise Server. *Compilers:* Intel® C/C++ Compiler 19.0.4 using "-fopenmp -O3 -xMIC-AVX512"; Cray® Compiling Envronment (CCE) "Cray clang" 9.1.2 using "-fopenmp -O3 -h cpu=mic-knl"; Clang LLVM 9.0.1 using "-fopenmp -O3 -mcpu=knl" with LLVM OpenMP Runtime.

Clock speeds quoted above are as reported by **/proc/cpuinfo**, but processors may operate at higher "turbo" speeds given sufficient thermal headroom. To enable the large stack sizes required by the recursion (and recursive parallelism) in UTS, the system stack limit is set to "unlimited" via the **ulimit** command and the **OMP_STACKSIZE** environment variable is set to 100MB. For the Intel TBB version of UTS, per-thread stack size is provided as an argument at TBB runtime initialization. Intel Cilk Plus limits maximum spawn depth to 1024 tasks, rendering it unable to run our test problem regardless of stack size.

Two major OpenMP implementations not included in the study are IBM XL and GCC. Unfortunately, the executable generated by the XL 20.1 compiler encounters a segmentation fault each time, regardless of stack size. This issue has been reproduced by an IBM compiler engineer. GCC 9.2 correctly executes UTS, but the task parallel OpenMP program does not scale at all: Even 2-thread executions run no faster than the sequential program. Code inspection reveals

[4] Each core has 4, but the BIOS configuration on the test system only has 2 enabled.

that GCC continues to employ a centralized queue for OpenMP tasks, while most other implementations use scalable distributed work-stealing schedulers.

While task reductions would be useful for the expression of the UTS tree traversal code, they are not used in the version tested in this study. Of the few compilers that so far claim support for this OpenMP 5.0 feature, only GCC successfully compiled a UTS version adapted to use task reductions. The others reject the use of the **in_reduction** clause on orphaned tasks. Bug reports have been filed for clang and LLVM, with fixes expected to be available in the 11.0 release and subsequently in derivative vendor implementations.

5 Results

The primary independent variable in this study is task granularity. Recall that the granularity of each task is the product $m \times g$ where m is the non-root branching factor and g is the number of repeated SHA-1 hash operations per tree node. Since each non-leaf node in the tree, excluding the root node, has 5 children, the lowest granularity of 5 hash operations per task represents only one SHA-1 hash operation per child node in the tree. Coarser granularities are obtained by repeating the hash operations.

The number of SHA-1 hash operations per tree node is a metric particular to the UTS benchmark, but Tables 1 and 2 present task granularity in terms of execution time and instructions, respectively. This data is taken from sequential executions, representing lower bounds since the time to do the calculations in each task may increase in parallel executions. This "work-time inflation" can result, e.g., from cache and NUMA effects [23]. Moreover, the integer-heavy instruction mix of SHA-1 means that these numbers may not be universally applied to other programs. In spite of these differences, our results provide some rough guidance for acceptable granularity of OpenMP tasks.

The time required to perform one SHA-1 hash operation (the first column of numbers in Table 1) varies widely across the four systems, roughly 3× slower on Xeon Phi compared to Xeon Skylake. Differences in clock speed and in core and

Table 1. Translating task granularity from SHA-1 operations / task to time / task

Architecture and Implementation	Time (μs) per op.	Time (μs) per recursive call at granularity					
		5 ops.	10 ops.	20 ops.	40 ops.	80 ops.	160 ops.
Xeon SKL - ICC	0.22	1.12	2.23	4.47	8.94	17.9	35.7
Xeon SKL - Clang	0.18	0.89	1.78	3.55	7.10	14.2	28.4
IBM P9 - PGI	0.31	1.53	3.06	6.13	12.2	24.5	49.0
IBM P9 - Clang	0.29	1.45	2.90	5.80	11.6	23.2	46.4
Arm TX2 - Armclang	0.32	1.61	3.22	6.43	12.9	25.7	51.4
Arm TX2 - Clang	0.34	1.73	3.45	6.90	13.8	27.6	55.2
Xeon Phi - ICC	0.64	3.21	6.42	12.8	25.7	51.4	103
Xeon Phi - Clang	0.74	3.68	7.36	14.7	29.4	58.9	118
Xeon Phi - CCE	0.63	3.14	6.29	12.6	25.2	50.3	101

Table 2. Translating task granularity from SHA-1 operations / task to machine instructions / task

Architecture and Implementation	Kilo instr. per op.	Kilo instr. per recursive call at granularity					
		5 ops.	10 ops.	20 ops.	40 ops.	80 ops.	160 ops.
Xeon SKL - ICC	1.74	8.72	17.4	34.9	69.7	139	279
Xeon SKL - Clang	1.70	8.51	17.0	34.0	68.1	136	272
IBM P9 - PGI	1.65	8.26	16.5	33.1	66.1	132	264
IBM P9 - Clang	1.67	8.35	16.7	33.4	66.8	133	267
Arm TX2 - Armclang	1.39	6.97	13.9	27.9	55.7	111	223
Arm TX2 - Clang	1.51	7.59	15.2	30.4	60.7	121	243
Xeon Phi - ICC	1.70	8.51	17.0	34.0	68.1	136	272
Xeon Phi - Clang	1.71	8.57	17.1	34.3	68.6	137	274
Xeon Phi - CCE	1.63	8.15	16.3	32.6	65.2	130	261

memory subsystem design contribute to these different computation rates. Using different compilers on the same system mostly results in similar SHA-1 execution rates, with some differences attributable to optimization choices and vectorization capability. The number of instructions required to perform one SHA-1 hash operation (the first column of numbers in Table 2) is a much narrower range (1.39-1.74 kilo-instructions) across systems than the time per operation. This observation suggests that generalizations of task granularity trends across systems may be more meaningful when expressed in terms of instructions per task rather than time per task.

5.1 Comparing Parallel Efficiency

The ability to compare across platforms with different architectures and core counts makes *percent parallel efficiency* an ideal metric. It is calculated by the formula $\frac{speedup}{number\_of\_threads} \times 100$, where *speedup* is $\frac{sequential\_excution\_time}{parallel\_execution\_time}$. Ideal speedup is a speedup equal to the number of threads, yielding a percent parallel efficiency of 100%.

Figures 2 and 3 show percent parallel efficiency for the UTS benchmark across architectures and OpenMP implementations (and TBB on the Intel Skylake platform). The vertical axis indicates percent parallel efficiency. The horizontal axis indicates the task granularity on a logarithmic scale, in thousands of instructions, derived from the data in Table 2. For each platform, the number of OpenMP threads is equal to the number of available cores on the machine, and each thread is bound to a single core.

ICC on Intel Skylake and PGI on IBM POWER9 are the top performers among OpenMP implementations, bested only by TBB (compiled with ICC). Even at the lowest granularity all three exceed 65% efficiency, and at a granularity of 67-70 kilo-instructions per task, they exceed 90% efficiency. On Intel Skylake, IBM POWER9, and Arm ThunderX2, the Clang/LLVM implementation achieves 43.0-47.7% efficiency at the lowest granularity and above 80% with

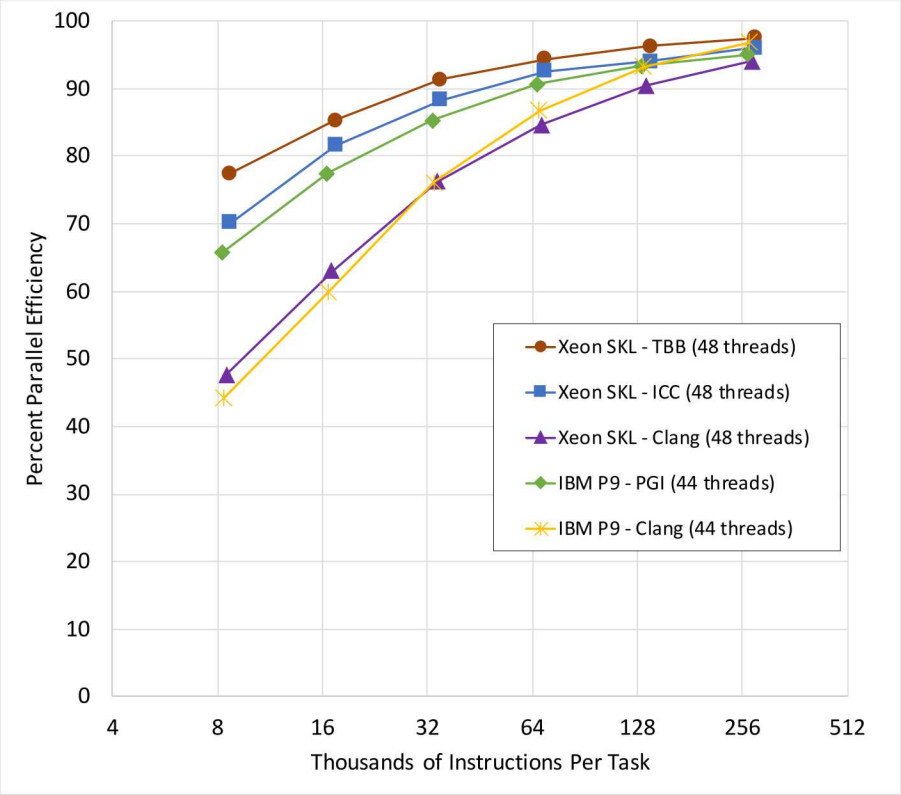

Fig. 2. Parallel efficiency of UTS as a function of task granularity on Intel Xeon Skylake and IBM POWER9 using various compilers

tasks of 61-68 kilo-instruction granularity. While the efficiency of the Arm implementation is similar to clang on ThunderX2 at low granularity, it achieves better efficiency at the coarser granularities. The Intel Xeon Phi exhibits the lowest efficiency of all architectures at the lowest granularity, but ICC fares much better than Clang or CCE. At a granularity of 65-68 kilo-instructions per task, CCE and ICC reach 80% efficiency while Clang lags behind at 72.3%.

Several trends emerge from the data. At the finest task granularity, the range of parallel efficiency is wide (16.8-77.5%). However, at the coarsest granularity it is much narrower (89.7-96.9%). Better performance at fine task granularity requires low overheads on the part of the OpenMP runtime implementations. Vendor implementations exhibit the best results on each architecture among those tested: ICC on Skylake and Xeon Phi, PGI on IBM, and armclang on ThunderX2. However, Clang/LLVM reaches reasonable efficiency at the coarser granularities on all architectures. Xeon Phi appears be the most challenging architecture for implementations to target efficiently, but it also has the most cores.

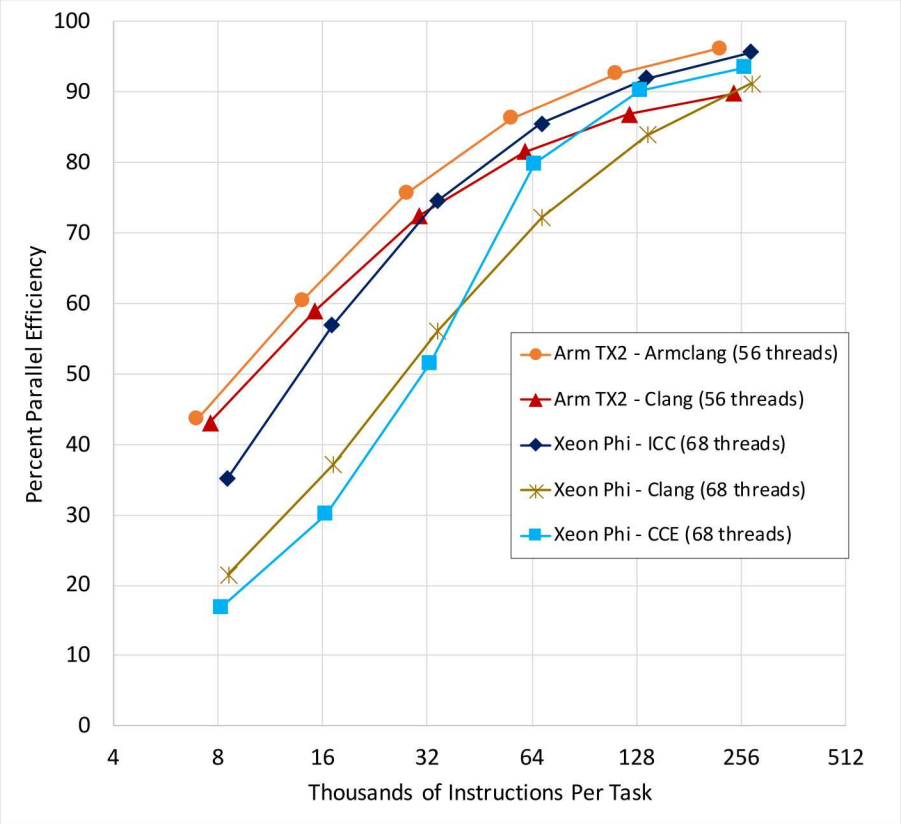

Fig. 3. Parallel efficiency of UTS as a function of task granularity on Arm ThunderX2 and Intel Xeon Phi Knights Landing using various compilers

The best implementations are successfully processing tasks consisting of $O(10k)$ instructions, but most implementations need a task granularity of $O(100k)$ instructions to reach high efficiency. Due to lack of space, the results in terms of execution time per task are not shown. However, Tables 1 and 2 can help to translate the results: In terms of execution time, the best implementations can manage tasks with only a few microseconds of work, but most require tasks to have at least tens of microseconds of work.

5.2 Thread Scalability and Simultaneous Multithreading

All platforms used in this study support multiple hardware threads per core, sometimes referred to as simultaneous multithreading (SMT). To assess the benefits of SMT, we compared the speedup of executions using only one OpenMP thread per core and executions using a number of OpenMP threads equal to the number of available hardware threads. Figure 4 shows the results across

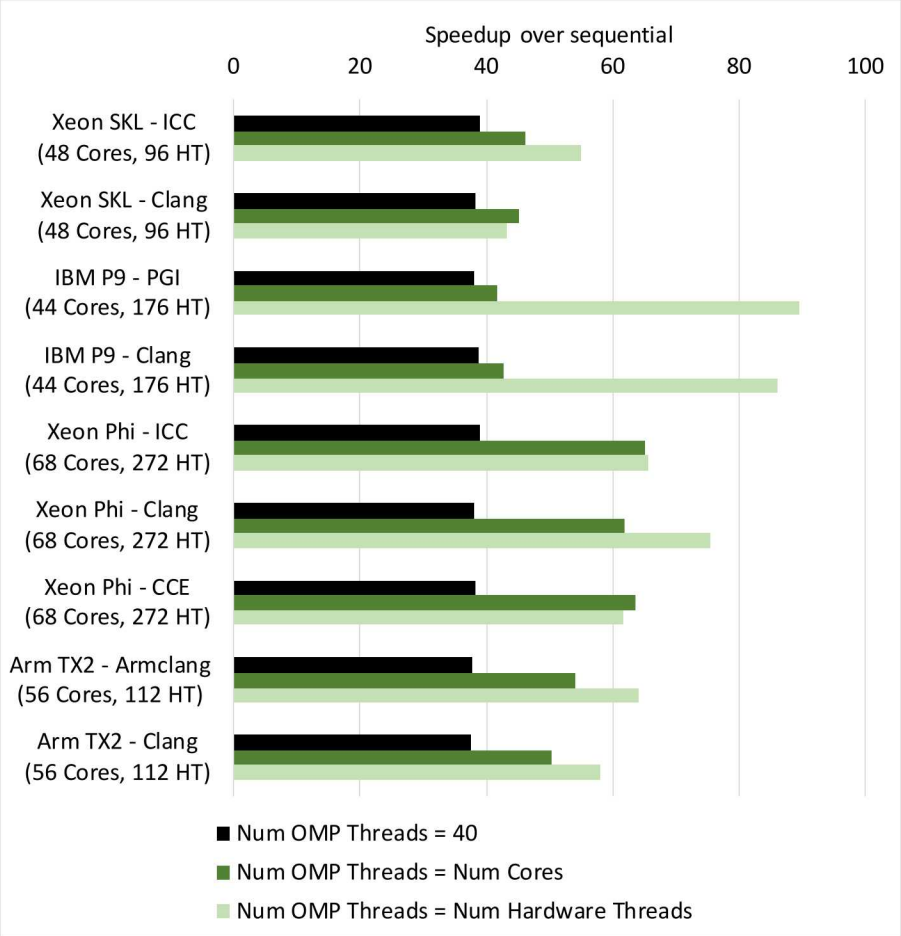

Fig. 4. Speedup at the coarsest granularity, varying thread count

the various architectures with the maximum task granularity from the earlier experiments (223-279 kilo-instructions per task). Also included are results using 40 threads, which allows comparison of speedup for the same thread count across the architectures. All implementations achieve over 37X speedup using 40 threads, and speedup continues to improve as more threads are added from 40 threads to the number of threads equal to the number of cores on each architecture.[5] POWER9 exhibits the best improvement from SMT, with its 4 hardware threads more than doubling the performance compared to using a single thread per core. The Arm ThunderX2 system shows a more modest benefit from SMT.

[5] UTS places relatively low demands on memory, so it can be more amenable to adding threads compared to more memory-hungry applications, which can saturate the memory subsystem with fewer active threads than the total available cores.

On Skylake (2-way SMT), ICC delivers a performance improvement with SMT while Clang sees none. The reverse occurs on Xeon Phi, with its 4-way SMT.

5.3 Quantifying Load Balancing Operations

Due to the unpredictable imbalance of the dynamically generated tree traversed in UTS, nearly continuous load balancing is required to scale the computation across available threads. The OpenMP implementation is free to move any unexecuted task from the thread on which it was generated to another thread in the team. We instrumented the UTS source code to check the thread number at the start of each task and increment a counter if it differs from the thread number of the thread on which its parent task executed. Figure 5 reports on a log-log scale the number of these "moved" child tasks per thread per second at each granularity (in thousands of instructions, as in the earlier figures). This metric allows comparison across executions on different numbers of threads and with different total execution times. With finer-grained tasks, all implementations are performing thousands of load balancing operations per second per thread. Unsurprisingly, the rate of load balancing operations decreases as the granularity of the tasks becomes coarser. The one outlier in this respect is CCE on Xeon Phi, whose load balancing rate is flat across the finer granularities.

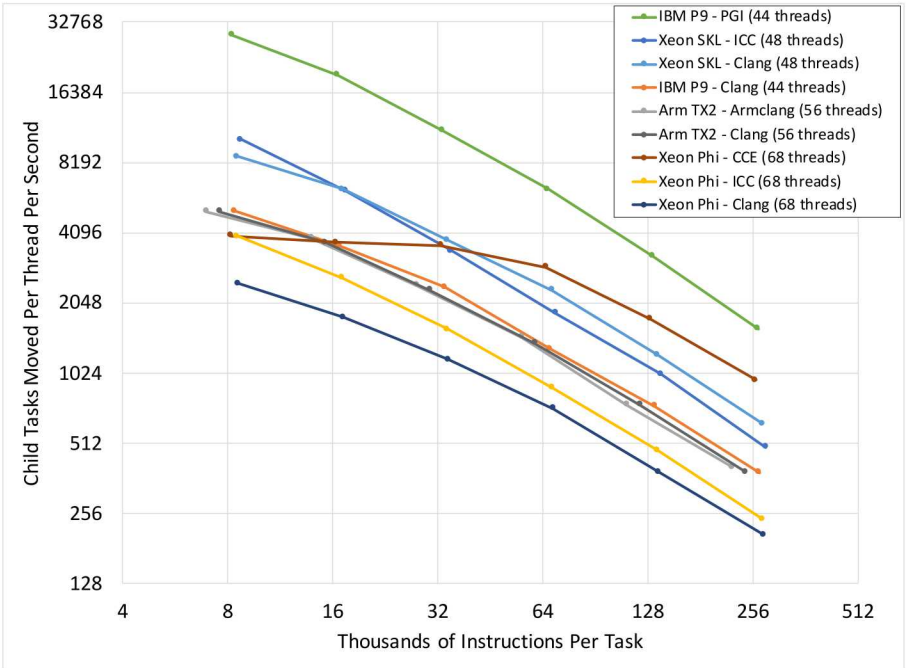

Fig. 5. Load balancing: Number of child tasks executing on different threads than their parent tasks, per thread, per second (log-log scale)

Table 3. Pearson correlation between speedup and number of moved child tasks per second per thread

SHA-1 ops. per task	5	10	20	40	80	160
Pearson correlation	0.69	0.59	0.42	0.42	0.38	0.12

The implementations that performed best in the parallel efficiency results, ICC on Xeon Skylake and PGI on IBM POWER9, carry out the most load balancing operations at the finest granularity setting. The least efficient implementations at the finest granularity setting, those on Xeon Phi, carry out the fewest load balancing operations. As a group, Clang/LLVM on IBM, Clang/LLVM on Arm, and Armclang produce nearly indistinguishable results on this metric throughout the range of granularities. The load balancing metric may also help to explain CCE's poor parallel efficiency at low granularities and better parallel efficiency at higher granularities, relative to other implementations: CCE is tied for the second-fewest tasks moved among implementations in the fine granularity executions but has the second-highest number of moved tasks in the coarse granularity executions.

Table 3 shows the Pearson correlation between speedup and the number of moved child tasks per second per thread, calculated across implementations at each granularity. A correlation coefficient near 1.0 or -1.0 indicates high positive or negative correlation, respectively, and a correlation coefficient near 0.0 indicates low correlation. Observe from the table that the finer the granularity of tasks, the more correlated are speedup and the number of load balancing operations. This data suggests that carrying out large numbers of load balancing operations per unit time becomes more important to the performance of OpenMP implementations as task granularity becomes finer.

Our moved child tasks metric is actually only a lower bound on the number of load balancing operations since we use untied tasks. Implementations can move untied tasks between threads during execution at any task scheduling points, such as at child task creation and when waiting in a **taskwait** region. Since we did not check the thread number after each task scheduling point, any such operations would have been missed. Unfortunately, repeated calls to **omp_get_thread_num()** can be expensive for some implementations. OMPT-based tools that introspect the OpenMP runtime library, once they are more universally supported, may be a better way to capture a more complete load balancing metric than the user-level code instrumentation we employed.

6 Related Work

Early work pertaining to the OpenMP tasking model included experimental studies [4, 7] and a treatment on the design rationale [3]. BOTS [8] was among the earliest benchmark suites for OpenMP tasking and included kernels like FFT and linear algebra, many based on recursive parallelism. Later efforts provided basic microbenchmarks [5], benchmarks exercising task dependences [29], and

evaluations of NUMA impacts on tasking [28]. A study applying OpenMP tasks to a graph problem is noteworthy for its scale, having run on an entire 1024-core SGI Altix UV system [1]. A more recent application study used a Fast Multipole Method (FMM) mini-application with results on some of the same architectures that we used [2].

Several efforts have focused on developing tools for analysis of programs using OpenMP tasking [11, 16–18, 27]. Previous work to analyze and reduce overheads has taken several directions, including cutoffs (adaptive [6] or static [13]) to limit parallelism. Others studied task granularity through profiling [12] or dynamic adjustment of granularity [19]. The unpredictable parallelism of UTS makes it a challenging target for the use of cutoffs or aggregation techniques.

7 Conclusions

The UTS benchmark is an extreme stress test, resulting in thousands of load balancing operations per second per thread. The study's focus on parallel efficiency allows comparison across diverse architectures and OpenMP implementations. The results illustrate that all implementations tested, except GCC, can efficiently manage tasks of $O(100k)$ instructions per task using all available cores, and the best implementations perform well with tasks of even $O(10k)$ instructions per task. The adequate efficiency of OpenMP tasking in Clang/LLVM as demonstrated in this study is particularly important for the OpenMP community due to its free availability under a permissive license and its role as a base for vendors to build upon. Still, we find that vendor OpenMP implementations, many of which are LLVM-based, do perform best. The overarching conclusion is that OpenMP tasking can be very efficient for unbalanced computation on a variety of architectures.

Acknowledgment

This work used advanced architecture testbed systems provided by the National Nuclear Security Administration's Advanced Simulation and Computing Program. Sandia National Laboratories is a multimission laboratory managed and operated by National Technology and Engineering Solutions of Sandia, LLC., a wholly owned subsidiary of Honeywell International, Inc., for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-NA-0003525.

References

1. Adcock, A.B., Sullivan, B.D., Hernandez, O.R., Mahoney, M.W.: Evaluating OpenMP tasking at scale for the computation of graph hyperbolicity. In: Rendell, A.P., Chapman, B.M., Müller, M.S. (eds.) IWOMP 2013: Proc. 9th Intl. Workshop on OpenMP. Lecture Notes in Computer Science, vol. 8122, pp. 71–83. Springer (2013)

2. Atkinson, P., McIntosh-Smith, S.: On the performance of parallel tasking runtimes for an irregular fast multipole method application. In: de Supinski, B.R., Olivier, S.L., Terboven, C., Chapman, B.M., Müller, M.S. (eds.) IWOMP 2017: Proc. 13th Intl. Workshop on OpenMP. Lecture Notes in Computer Science, vol. 10468, pp. 92–106. Springer (2017)
3. Ayguadé, E., Copty, N., Duran, A., Hoeflinger, J., Lin, Y., Massaioli, F., Teruel, X., Unnikrishnan, P., Zhang, G.: The design of OpenMP tasks. IEEE Transactions on Parallel and Distributed Systems **20**, 404–418 (March 2009)
4. Ayguadé, E., Duran, A., Hoeflinger, J., Massaioli, F., Teruel, X.: An experimental evaluation of the new OpenMP tasking model. In: Adve, V.S., Garzarán, M.J., Petersen, P. (eds.) Language and Compilers for Parallel Computers (LCPC). Lecture Notes in Computer Science, vol. 5234, pp. 63–77. Springer (2007)
5. Bull, J.M., Reid, F., McDonnell, N.: A microbenchmark suite for OpenMP tasks. In: Chapman, B.M., Massaioli, F., Müller, M.S., Rorro, M. (eds.) IWOMP 2012: Proc. 8th Intl. Workshop on OpenMP. Lecture Notes in Computer Science, vol. 7312, pp. 271–274. Springer (2012)
6. Duran, A., Corbalán, J., Ayguadé, E.: An adaptive cut-off for task parallelism. In: SC08: ACM/IEEE Supercomputing 2008. pp. 1–11. IEEE (2008)
7. Duran, A., Corbalán, J., Ayguadé, E.: Evaluation of OpenMP task scheduling strategies. In: Eigenmann, R., de Supinski, B.R. (eds.) IWOMP '08: Proc. Intl. Workshop on OpenMP. Lecture Notes in Computer Science, vol. 5004, pp. 100–110. Springer (2008)
8. Duran, A., Teruel, X., Ferrer, R., Martorell, X., Ayguadé, E.: Barcelona OpenMP Tasks Suite: A set of benchmarks targeting the exploitation of task parallelism in OpenMP. In: ICPP '09: Proc. 38th Intl. Conference on Parallel Processing. pp. 124–131. IEEE (Sept 2009)
9. Eastlake, D., Jones, P.: US Secure Hash Algorithm 1 (SHA-1). RFC 3174, Internet Engineering Task Force (Sep 2001), http://www.rfc-editor.org/rfc/rfc3174.txt
10. Frigo, M., Leiserson, C.E., Randall, K.H.: The implementation of the cilk-5 multithreaded language. In: PLDI '98: Proc. ACM SIGPLAN 1998 Conference on Programming Language Design and Implementation. p. 212?223. PLDI ?98, Association for Computing Machinery, New York, NY, USA (1998)
11. Fürlinger, K., Skinner, D.: Performance profiling for OpenMP tasks. In: Müller, M.S., de Supinski, B.R., Chapman, B.M. (eds.) IWOMP '09: Proc. 5th Intl. Workshop on OpenMP. Lecture Notes in Computer Science, vol. 5568, pp. 132–139. Springer (2009)
12. Gautier, T., Pérez, C., Richard, J.: On the impact of OpenMP task granularity. In: de Supinski, B.R., Valero-Lara, P., Martorell, X., Bellido, S.M., Labarta, J. (eds.) IWOMP 2018: Proc. 14th Intl. Workshop on OpenMP. Lecture Notes in Computer Science, vol. 11128, pp. 205–221. Springer (2018)
13. Iwasaki, S., Taura, K.: A static cut-off for task parallel programs. In: PACT 2016: Intl. Conference on Parallel Architecture and Compilation Techniques. pp. 139–150 (Sep 2016)
14. Lattner, C., Adve, V.: LLVM: A compilation framework for lifelong program analysis and transformation. In: CGO 2004: Intl. Symposium on Code Generation and Optimization. pp. 75–88. San Jose, CA, USA (Mar 2004)
15. Leiserson, C.E.: The cilk++ concurrency platform. The Journal of Supercomputing **51**(3), 244–257 (2010)
16. Lin, Y., Mazurov, O.: Providing observability for OpenMP 3.0 applications. In: Müller, M.S., de Supinski, B.R., Chapman, B.M. (eds.) IWOMP '09: Proc. 5th

Intl. Workshop on OpenMP. Lecture Notes in Computer Science, vol. 5568, pp. 104–117. Springer (2009)
17. Lorenz, D., Mohr, B., Rössel, C., Schmidl, D., Wolf, F.: How to reconcile event-based performance analysis with tasking in OpenMP. In: Sato, M., Hanawa, T., Müller, M.S., Chapman, B.M., de Supinski, B.R. (eds.) IWOMP 2010: Proc. 6th Intl. Workshop on OpenMP. Lecture Notes in Computer Science, vol. 6132, pp. 109–121. Springer (2010)
18. Lorenz, D., Philippen, P., Schmidl, D., Wolf, F.: Profiling of OpenMP tasks with Score-P. In: ICPPW 2012: 41st International Conference on Parallel Processing Workshops. pp. 444–453. IEEE Computer Society (2012)
19. Navarro, A., Mateo, S., Pérez, J.M., Beltran, V., Ayguadé, E.: Adaptive and architecture-independent task granularity for recursive applications. In: de Supinski, B.R., Olivier, S.L., Terboven, C., Chapman, B.M., Müller, M.S. (eds.) IWOMP 2017: Proc. 13th Intl. Workshop on OpenMP. Lecture Notes in Computer Science, vol. 10468, pp. 169–182. Springer (2017)
20. Olivier, S., Huan, J., Liu, J., Prins, J., Dinan, J., Sadayappan, P., Tseng, C.W.: UTS: An unbalanced tree search benchmark. In: Almási, G., Cascaval, C., Wu, P. (eds.) LCPC 2006: Proc. 19th Intl. Workshop on Languages and Compilers for Parallel Computing. LNCS, vol. 4382, pp. 235–250. Springer (2007)
21. Olivier, S.L., Prins, J.F.: Evaluating OpenMP 3.0 run time systems on unbalanced task graphs. In: Muller, M.S., de Supinski, B.R., Chapman, B.M. (eds.) IWOMP '09: Proc. 5th Intl. Workshop on OpenMP. Lecture Notes in Computer Science, vol. 5568, pp. 63–78. Springer (2009)
22. Olivier, S.L., Prins, J.F.: Comparison of OpenMP 3.0 and other task parallel frameworks on unbalanced task graphs. Intl. Journal of Parallel Programming **38**(5-6), 341–360 (2010)
23. Olivier, S.L., de Supinski, B.R., Schulz, M., Prins, J.F.: Characterizing and mitigating work time inflation in task parallel programs. In: SC 12: Proc. Intl. Conference on High Performance Computing, Networking, Storage and Analysis. pp. 65:1–65:12. IEEE Computer Society Press (2012)
24. OpenMP Architecture Review Board: OpenMP application programming interface, version 3.0 (May 2008), https://www.openmp.org/wp-content/uploads/spec30.pdf
25. OpenMP Architecture Review Board: OpenMP application programming interface, version 5.0 (November 2018), https://www.openmp.org/wp-content/uploads/OpenMP-API-Specification-5.0.pdf
26. Reinders, J.: Intel Threading Building Blocks: Outfitting C++ For Multi-Core Processor Parallelism. O'Reilly (2007)
27. Schmidl, D., Philippen, P., Lorenz, D., Rössel, C., Geimer, M., an Mey, D., Mohr, B., Wolf, F.: Performance analysis techniques for task-based OpenMP applications. In: Chapman, B.M., Massaioli, F., Müller, M.S., Rorro, M. (eds.) IWOMP 2012: Proc. 8th Intl. Workshop on OpenMP. Lecture Notes in Computer Science, vol. 7312, pp. 196–209. Springer (2012)
28. Terboven, C., Schmidl, D., Cramer, T., an Mey, D.: Assessing OpenMP tasking implementations on NUMA architectures. In: Chapman, B.M., Massaioli, F., Müller, M.S., Rorro, M. (eds.) IWOMP 2012: Proc. 8th Intl. Workshop on OpenMP. Lecture Notes in Computer Science, vol. 7312, pp. 182–195. Springer (2012)
29. Virouleau, P., Brunet, P., Broquedis, F., Furmento, N., Thibault, S., Aumage, O., Gautier, T.: Evaluation of OpenMP dependent tasks with the KASTORS benchmark suite. In: DeRose, L., de Supinski, B.R., Olivier, S.L., Chapman, B.M., Müller, M.S. (eds.) IWOMP 2014: Proc. 10th Intl. Workshop on OpenMP. Lecture Notes in Computer Science, vol. 8766, pp. 16–29. Springer (2014)