University of Dundee

COMPROF and COMPLACE

Kirkpatrick, Ryan; Brown, Christopher; Janjic, Vladimir

Published in:
2022 IEEE 29th International Conference on High Performance Computing, Data, and Analytics (HiPC)

DOI:
10.1109/HiPC56025.2022.00040

Publication date:
2023

Licence:
CC BY

Document Version
Peer reviewed version

Link to publication in Discovery Research Portal

Citation for published version (APA):
Kirkpatrick, R., Brown, C., & Janjic, V. (2023). COMPROF and COMPLACE: shared-memory communication profiling and automated thread placement via dynamic binary instrumentation. In *2022 IEEE 29th International Conference on High Performance Computing, Data, and Analytics (HiPC)* (pp. 236-245). IEEE. https://doi.org/10.1109/HiPC56025.2022.00040

General rights
Copyright and moral rights for the publications made accessible in Discovery Research Portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

• Users may download and print one copy of any publication from Discovery Research Portal for the purpose of private study or research.
• You may not further distribute the material or use it for any profit-making activity or commercial gain.
• You may freely distribute the URL identifying the publication in the public portal.

Take down policy
If you believe that this document breaches copyright please contact us providing details, and we will remove access to the work immediately and investigate your claim.

Download date: 31. Jul. 2023

COMPROF and COMPLACE: Shared-Memory Communication Profiling and Automated Thread Placement via Dynamic Binary Instrumentation

Ryan Kirkpatrick
School of Computer Science
University of St Andrews
St Andrews, UK
rw86@st-andrews.ac.uk

Christopher Brown
School of Computer Science
University of St Andrews
St Andrews, UK
cmb21@st-andrews.ac.uk

Vladimir Janjic
School of Science and Engineering
University of Dundee
Dundee, UK
vjanjic001@dundee.ac.uk

***Abstract*—This paper presents COMPROF and COMPLACE, a novel profiling tool and thread placement technique for shared-memory architectures that requires no recompilation or user intervention. We use dynamic binary instrumentation to intercept memory operations and estimate inter-thread communication overhead, deriving (and possibly visualising) a *communication graph* of data-sharing between threads. We then use this graph to map threads to cores in order to optimise memory traffic through the memory system. Different paths through a system's memory hierarchy have different latency, throughput and energy properties, COMPLACE exploits this heterogeneity to provide automatic performance and energy improvements for multi-threaded programs. We demonstrate COMPLACE on the NAS Parallel Benchmark (NPB) suite where, using our technique, we are able to achieve improvements of up to 12% in the execution time and up to 10% in the energy consumption (compared to default Linux scheduling) while not requiring any modification or recompilation of the application code.**

***Index Terms*—NUMA, Thread Placement, Data Placement, Cache Optimisation, Energy Optimisation**

I. Introduction

Inter-thread communication is an important determiner of an application's performance, particularly in HPC systems. As compute capacity is "scaled-out", inter-core communication often becomes the main bottleneck, resulting in CPUs being significantly under-utilised. This situation is possible whether the communication is via explicit message-passing (e.g. MPI applications) or implicit memory updates (in shared-memory NUMA systems). GPU designers have mitigated this situation and increased utilisation through hardware thread multiplexing – a technique with limited mileage for CPU workloads. There is a relatively limited set of existing tools to profile and understand communication, particularly on shared-memory systems. Under these circumstances developers face an uphill battle optimising their applications, and runtime systems lack the visibility necessary to schedule them efficiently. In our view the key to achieving scale for a large class of applications lies in studying and manipulating communication.

For the purpose of open access, the author has applied a Creative Commons Attribution (CC BY) license to any Author Accepted Manuscript version arising

Significant research has also been invested in improving energy footprints of computer applications, with a special focus on large-scale highly-parallel applications that have become essential in both science and industry. Such applications usually run for a long time and consume lots of resources, resulting in potentially huge energy footprints. Reducing this footprint as much as possible while still retaining acceptable performance is of paramount interest. Inefficient communication can result in slow and energy-inefficient routes being taken through a memory or communication network, sometimes with overheads several orders of magnitude higher than the fastest inter-node path.

This paper presents a novel profiling tool, COMPROF, for understanding patterns of communication between threads in a parallel application. This allows us to profile the application and build a graph of inter-thread communication. We also present a runtime optimisation algorithm that uses the communication graph to map the application threads to the cores of a system. We package these techniques together into a new profiling tool and thread placement technique, called COMPLACE. The objective of the placement algorithm is to group the threads that communicate intensively between themselves so that they share memory resources. This allows us to improve data placement and avoid thread communication over slow caches, thus improving both performance and energy efficiency of parallel applications. By using dynamic binary translation instead of simulation we achieve lower-overhead profiling than existing work. The concrete research contributions of this paper are:

1) a novel communication profiling method, deriving thread communication graphs of parallel application;
2) a runtime optimisation algorithm that takes as an input the communication graph and decides on placement of threads to the cores of a system, aiming to both increase the performance and reduce energy consumption; and,
3) demonstrations of the performance and energy consumption improvements on the *Ocean* benchmark from Parsec, and a set of realistic benchmarks from the NAS

Parallel Benchmark (NPB) suite [3].

II. Motivation

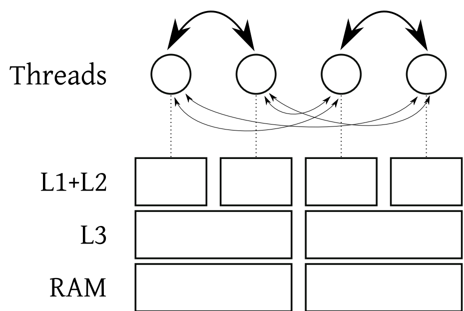

Fig. 1. A hypothetical graph of four threads is with the amount of write-to-read communication depicted via edge thickness. Shown below is a mapping to a two-node, four-core NUMA machine that maximises the amount of communication via low-latency caches and RAM.

Processors with NUMA architectures are, today, a *de-facto* standard for building large high-performance systems. In these architectures, there is a hierarchical grouping of the physical processor cores, based on the *distance* between them, where the distance is measured as a latency in communication (communication being reads/writes to the same data). In such tiered cache architectures, a write to a cacheline by one core followed by a read to that same cacheline by a different core, involves a certain amount of high-latency communication. The amount of communication will depend on the placement of the data in RAM and what state various caches hold that line in. When the two cores share a close cache, that cache may be able to service the read without involving higher-latency caches; and if the two cores share local RAM some inter-package communication can be avoided. Communication via low-latency caches and local-RAM has better latency, throughput and energy properties than farther caches and remote-RAM respectively. Therefore, threads in a parallel application that share a lot of data should always ideally be placed on the cores of a system that share as many memory resources as possible (i.e. the cores which are close-by in the hierarchical core grouping). On the other hand, if the two threads do not share much data, it is perfectly acceptable to place them on distant cores that do not share fast cache, or even that share no cache or RAM at all. Correct placement of the threads, especially in memory-intensive applications, can significantly impact both performance and energy consumption.

To motivate our technique, consider the example application shown in Figure 1. Lines with arrows show communication between threads, with the thickness of the line indicating volume of *communication*. In this example, we can see that the communication between Threads 1 and 2 on one side, and 3 and 4 on the other is most intensive. Therefore, in placing these threads, we should ideally put Threads 1 and 2 (and also 3 and 4) to the cores that share a cache as possible, hoping that most of the communication will go through this cache, or else will be served by the local RAM. This is demonstrated on the same figure. There is still communication (e.g. between Threads 1 and 3 and 1 and 4) that goes over the NUMA boundary, but the volume of this communication is smaller than for Threads 1 and 3. The default Linux scheduler cannot guess how much the threads in a parallel application communicate with each other, and therefore does not base the thread placement decisions on their communication profile. The goal of our work is to show an automated method to optimise the amount of communication happening via local RAM and low-latency caches without requiring any modification of the application.

III. COMPROF: Communication Profiling

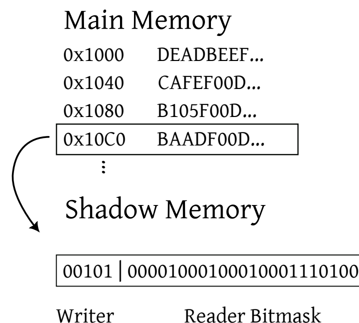

Fig. 2. Overview of COMPROF. For each cacheline-sized chunk of application memory address a 64bit shadow word is allocated. This word contains the provenance of the value in main memory (the writer thread) as well as a set of observers of the value (packed into a bitfield). These data are used to construct the final communication graph.

In this section we introduce COMPROF, a communication profiling tool which operates on unmodified Linux applications. COMPROF is first phase of our thread placement scheme, COMPLACE, which is described in Section IV. We will also demonstrate in Section V how COMPROF can be used in isolation to gain valuable black-box insight into an multi-threaded application. COMPROF uses Dynamic Binary Instrumentation to profile shared-memory *communication*. For our purposes communication is a write by one thread to an area of memory (a cacheline) followed by the first read from different thread which witnesses the new value. We use this definition to approximate the cache-coherence messages required to propagate a value through a hierarchy of caches. COMPROF estimates only the logical communication between application threads, it does not record any hardware-specific details such as cache behaviour. In our testing COMPROF causes, approximately, a 20x slowdown of the application. Architectural simulation, which underpins previous work[6, 2], is known to be orders of magnitude slower.

Intel's PIN framework[19] is used to perform dynamic instrumentation on the target application. Algorithm 1 shows high-level pseudocode for the instrumentation routines. The approach is spiritually similar to *addresssantisier*[24] and *valgrind*[23] in that we allocate and maintain shadow state for blocks of application memory. However, the information we track (last writer and the set of readers) is used to estimate hardware communication operations and not to detect undefined behaviour. In our implementation, the writer field

Algorithm 1 Pseudocode for Communication Profiling

```
 1: procedure INITIALISE
 2:     for all cacheline ← Process Memory Map do
 3:         last_writer_cacheline ← -1
 4:         readers_cacheline ← {}
 5:     end for
 6:     for all i ← Thread IDs do
 7:         for all j ← Thread IDs do
 8:             comm_i,j ← 0
 9:         end for
10:     end for
11: end procedure
12:
13: procedure INSTRUMENTREAD(threadid, cacheline)
14:     atomically do
15:         readers_cacheline ← readers_cacheline|{threadid}
16:     end atomically
17: end procedure
18:
19: procedure INSTRUMENTWRITE(threadid, cacheline)
20:     atomically do
21:         w ← last_writer_cacheline
22:         rs ← readers_cacheline
23:         last_writer_cacheline ← threadId
24:         readers_cacheline ← {}
25:     end atomically
26:     for all r ← rs do
27:         ATOMICINCREMENT(comm_w,r)
28:     end for
29: end procedure
```

and reader sets are packed as a bitset into a single 64-bit word, facilitating the use of ordinary locked instructions to implement the atomic sections (see Figure 2). [1] A 64bit shadow word is needed for every cacheline (512bits on most platforms) leading to an overhead of 12.5%. COMPROF detects memory map changes by hooking the *mmap* and *brk* syscalls as well as using callbacks from PIN's image loader. A shadow mapping for the stack is created by reading */proc/self/maps* and over-allocating to allow the stack to automatically grow. Special care must be taken not to instrument code in kernel-provided code, such as the *vdso*. COMPROF is conservative in maintaining a reader set for each write to the memory location. An alternate implementation could track a single reader and writer set for each memory location. This risks over-counting communication but this may not be an issue in practice. Per-location read and write sets could also be the basis for an online profiler that estimates these sets via sampling memory performance counters. We assume that subsequent reads to a shared variable, with no intervening write, can be served without any communication related overhead, so COMPROF does not count them. COMPROF also has a configuration where it adds additional edges between writers of a new value and readers of the previous value to simulate flush/invalidation traffic in the cache hierarchy; although we do not currently use this data for mapping decisions. Note that COMPROF does not consider how closely in time the read follows the write. If there are many operations in-between, the value may be evicted from the cache hierarchy in the meantime causing some potential benefit to be lost.

[1] COMPROF simulates sequentially consistent memory. Its results may be inaccurate under weak memory models, or programs with data races. We expect the results to still be a good estimate for performance profiling purposes.

IV. COMPLACE: AUTOMATED THREAD PLACEMENT

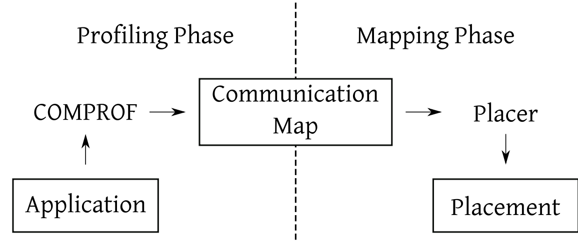

Fig. 3. COMPLACE: Automatic Thread Placement

In this section, we describe COMPLACE, our automatic thread placement scheme which optimises memory and cache coherence traffic. COMPLACE comprises two phases, a *profiling* phase (described in Section III), and a *mapping* phase, where we give implementation details in Section IV-A. We give a high level overview of the two phases here:

1) **Profiling Phase** COMPROF is used to profile the application and create a weighted graph of estimated inter-thread *communication*. Here *communication* refers to a write to an area of memory, followed by a later read of that area by a different thread. The cacheline size of the target machine is used to size these areas. The output of this phase is a mapping from each thread-pair to the amount of communication between them.
2) **Mapping Phase** During the application launch, the host hardware is inspected and its cache topology determined. A graph-matching algorithm is then applied to the two topologies to create the thread/core mapping. Finally, the application is launched and some heuristic is used to match operating system threads with their corresponding node in the measured communication graph.

The profiling phase is performed offline (in a separate test run) only once, but is redone if parameters of the application (i.e. the number of threads) change. There is no requirement for the profiling and mapping phases to be run on the same hardware or even the same architecture. COMPROF may also be executed on smaller problem instances, so long as the resulting communication behaviour is representative of larger sizes. In other implementations of this concept a profiling phase is run online (during the main run), where it may be repeated to detect or respond to phase changes in the application's behaviour [1].

The effectiveness of the mapping phase depends on the graph structure of both the application's communication and the target hardware. Ideally the communication graph features sets of tightly-coupled threads of suitable size for mapping to processor nodes/caches. This also requires the target platform

to have heterogeneous communication between sets of cores, so that there are efficient communication paths to take advantage of. Examples of this heterogeneity include NUMA and SMT systems. Applications with clique-like communication will see little benefit from the mapping phase, since there is no mapping that favours one communication path over another. Applications that do not have a deterministic communication pattern (including many graph algorithms and applications with dynamic task assignment) will also see no benefit since the placer uses the previous, profiled data to inform its future placement decisions. In these cases performance may in fact be reduced by preventing the scheduler's built-in balancing logic from moving threads and thus sharing the load. An online profiler could address many of these issues and allow per-phase mappings for applications with phase changes at the cost of some runtime overhead. Our implementation, which maps threads at application start has no overhead once initialised.

A. Thread Placement Implementation

Algorithm 2 Pseudocode for Thread Mapping

```
1: function MAPTHREADS(threadGraph, coreGraph)
2:     while not EMPTY(threadGraph) do
3:         thread ← REMOVEANYNODE(threadGraph)
4:         core ← REMOVEANYNODE(hardwareGraph)
5:         mapping_thread ← core
6:         loop
7:             pendingEdges ← edges in threadGraph
8:                 connecting an unmapped and a mapped
9:                 thread
10:            if EMPTY(pendingEdges) then
11:                break
12:            end if
13:            threadEdge ← maximum weighted edge in
14:                pendingEdges
15:            start_thread ← mapped end of threadEdge
16:            end_thread ← unmapped end of threadEdge
17:            start_core ← mapping_start_thread
18:            coreEdge ← maximum weighted edge in
19:                coreGraph connecting start_core
20:                to an unmapped core
21:            end_core ← unmapped end of coreEdge
22:            mapping_end_thread ← end_core
23:        end loop
24:    end while
25:    return mapping
26: end function
```

In COMPLACE, we achieve thread mapping using a naïve greedy algorithm. Algorithm 2 gives the high-level pseudocode. At each iteration a mapping is made so that a suitable maximum-weight communication edge will be mapped to a maximum-weight hardware edge. The hardware graph is constructed by inspecting *procfs* to determine the lowest latency cache each pair of cores share, the level of that cache determines the edge weight through a simple map. For our default parameters, we assign values of decreasing orders of magnitude per cache memory: L1 sharing is assigned 10000, L2 1000 etc. Our thread mapping was designed for high performance applications that assign a single worker thread to each core. It does not handle the case of more threads than cores. [2] Fewer threads than cores can lead to poor performance on SMT systems, since the technique prefers to place tightly coupled threads on adjacent hardware threads so they can communicate via the L1 data cache. Using SMT threads when there are free cores can have a significant performance hit.

B. Thread Identification

As the application is launched a small daemon monitors the *procfs* directory for the application. It notices threads as they are created and sets their affinity. Once all the threads are mapped it sleeps until the application exits. In our implementation, threads are identified by the order that they are spawned in. We found this heuristic works well for the applications we tested. However, more involved options are possible: threads could be identified also by the parent thread which issued the spawn call, creating a tree of threads; or the behaviour of new threads could be monitored for a short time (e.g. using the ptrace API) to create a behaviour-based identifier.

V. CASE STUDY

In this section, we attempt to demonstrate the *profiling* and *mapping* phases of COMPLACE on the *Ocean* application from the SPLASH2X benchmark suite, distributed as part of the PARSEC3 suite. As part of our evaluation, we use the *CP* variant of the *Ocean* benchmark, which has false aliasing, in order to show our profiler in action. In Section VI we give a full performance evaluation of COMPLACE across the NAS benchmark suite. In this section, we use a 56-thread shared-memory NUMA machine, as described in Table II.

A. Phase 1: Profiling

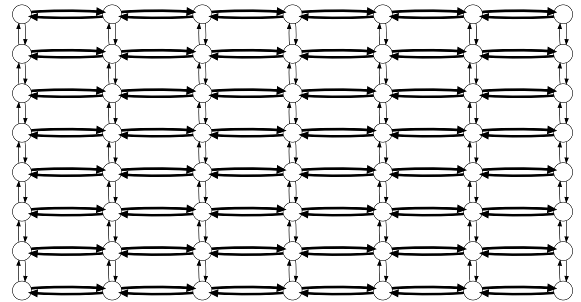

Fig. 4. Profiling results for the *Ocean* benchmark. Nodes show threads, and edge width shows total communication measured over the run. Edges with a weight below 10% of the maximum weight have been removed.

As mentioned in Section IV, the first phase of COMPLACE comprises a profiling stage. Here we use COMPROF to profile *Ocean* on a 56 thread NUMA machine. This profiling step requires no input or interaction from the user, and the profiling phase treats the application executable as a black-box. Figure 4

[2]One possibility for supporting this is to introduce penalties for placing a thread edge on a congested hardware edge. The greedy algorithm would then optimise over some objective which includes these penalties.

shows the resulting communication graph, where each node of the graph represents one of the 56 threads. The edges between the nodes represent the communication measured over the run: thicker lines between two nodes represent more communication between those two nodes than two nodes with a thinner line connecting them. Figure 4 illustrates, for example, that there is more communication between the top-leftmost thread and its horizontal neighbour than with thread and its vertical neighbour. The figure also indicates that the pattern of communication in the *Ocean* benchmark resembles a 2D grid structure. This is not surprising, as, indeed, *Ocean* is a simulation that assigns 2D grid partitions to worker threads and data communication happens along grid edges after each iteration. It seems as though significantly more communication is happening along one of the grid axes, suggesting a cache line aliasing issue[3]. For example, it could be that the data communicated between horizontal neighbours is well-packed into the machine's cachelines, whereas the apparent communication between vertical neighbours is an artefact of unrelated variables having been placed in the same cacheline. We believe that COMPROF is useful as a standalone tool for visualisation multi-threaded behaviour and diagnosing performance issues caused by communication overhead.

B. Phrase 2: Mapping

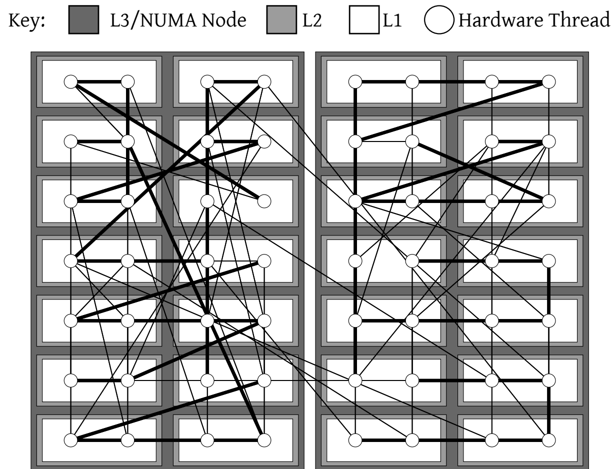

Fig. 5. Automatic thread placement of the ocean benchmark on a 2-node NUMA system. Physical SMT cores are drawn as white circles and the enclosing rectangles represents cache levels. Profiled communication is drawn by connecting lines, the thickness of which is proportional to the amount of measured communication.

The second phase of COMPLACE is to uses the profiler's output from Phase 1 to generate thread placement. Figure 5 shows a 56-thread instance after mapping to a two-node NUMA system. The figure illustrates the two NUMA nodes, represented by dark-gray rectangles, comprising cores (white rectangles); each core comprises two threads (circles); thread communication is represented by edges between the circles (the thicker the line the more communication between two threads). The figure shows that the mapper has succeeded in reducing the amount of inter-node memory traffic, with minimal communication traffic across the NUMA boundary.

[3]We are deliberately using the *CP* variant of the benchmark, which does have false aliasing, to show COMPROF in action.

C. Ocean Results

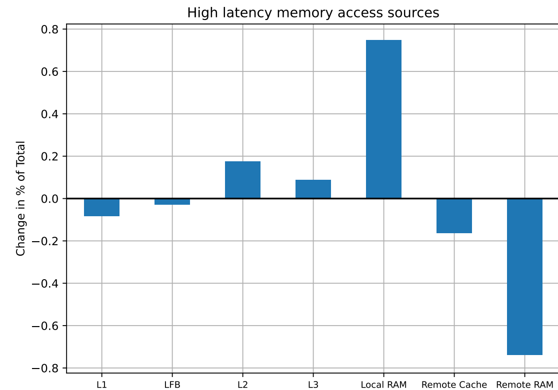

Fig. 6. Change in High-latency memory read sources for ocean_cp after applying COMPLACE. Results are sampled from Intel's "Load Latency Performance Monitoring" counters with the latency threshold set to 30 cycles. The percentage of samples from each source was computed for baseline and COMPLACE separately and the difference between the two percentages is shown. Uncached and Unknown L3 misses were removed from the data.

We ran this configuration against a baseline of default Linux thread-placement (without thread pinning) algorithm and measured time as well as energy consumption as reported by RAPL (Running Average Power Limit) [9]. We take the average of 5 runs for each measured value. Table I shows the averaged results, where the original configuration (baseline) runtime reports 25.95 seconds versus our optimised configuration produced by COMPLACE of 22.49 seconds. The total package energy for the baseline configuration was 5220 joules, versus 4642 joules for the COMPLACE configuration. This shows approximately a 15% performance and 12% energy improvement using our technique, validating our approach to reducing communication overhead. The table also reports counts of sampled memory reads obtained during the run, hitting each of the L1, LFB, L2, L3, RAM, local cache and RAM memories. The samples were obtained from the memory latency performance monitoring facility of the Broadwell chips we used. The sampling was configured to measure loads taking 30 or more cycles. We can clearly see that the number sampled reads for the COMPLACE configuration of L1 has decreased by 9,144.8, L2 by 161,719.2 and L3 by 14,870.4, and the total Remote RAM read has decreased by 27,037. Figure 6 illustrates how the source of high-latency memory load operations changed between the two configurations. It is clear that COMPLACE's mapping has significantly reduced inter-node NUMA traffic, where the decrease in remote-RAM traffic is matched by an equal-sized increase in local-RAM traffic. These accesses have favourable latency, throughput and energy properties compared to an access that traverses an inter-node link. We believe that the effectiveness is primarily due to

TABLE I
RESULTS USING OCEAN_CP WITH AND WITHOUT OUR TECHNIQUE.

Configuration	time (s)	Pkg Energy(J)	RAM Energy (J)	L1	LFB	L2	L3	Local RAM	Remote Cache	Remote RAM
Baseline	25.953	5220.19	1042.28	113176.60	2898601.00	150913.00	322234.80	74155.00	10157.80	34668.20
COMPLACE	22.248	4642.71	926.69	104031.80	2736881.80	148534.80	307364.40	95564.20	4048.20	7631.20

reduced inter-node NUMA traffic, though the technique is not NUMA specific and could, on other platforms, reduce traffic to high-latency caches. We expect the gains to be more modest in these cases. However, as these savings come without any user intervention, recompilation of the application, or need to alter the software's toolchain or build system, even a modest improvement is worthwhile.

VI. RESULTS

TABLE II
HARDWARE SETUP

CPU Part	Intel Xeon E5-2690
Sockets	2
Cores per socket	14
Threads per core	2
L1d cache	32Kib
L1i cache	32Kib
L2 cache	256Kib
L3 cache	35840Kib
RAM per Socket	128 Gib
RAM type	DDR4 @ 2133 MT/s
RAM Configuration	4 $\times$ 32 Gib DIMMs per socket
Operating System	Scientific Linux 7.9
Kernel Version	3.10.0

In this section we evaluate the performance of COMPLACE on the NAS Parallel Benchmarks (NPB) suite. Since our technique is for shared-memory systems we use the OpenMP implementations that come with the benchmarks. No modifications were made to the sources, compiler, or build system; we used problem sizes D and E. We note that some benchmarks did not build or run correctly at size E, and these were removed from our results set. We measured total execution time and RAPL energy measurements together for five iterations and present the arithmetic mean and standard deviation for our results in Table III. The target hardware is a two-node, Broadwell generation NUMA system, described in Table II. We used the default scheduler and frequency governor configuration for all runs.

Table III gives the averaged results for NPB suite, categorised by benchmark sizes D and E, where we report up to 12% performance improvement and up to 10% improvement in energy for a number of benchmarks. Figure 7 shows the COMPLACE performance at size D normalised to baseline performance. Two benchmarks, *EP* and *IS*, show no performance improvement. We also note that no benchmark is showing significant degradation. We give three possible reasons for the observed speedup, which we discuss in turn.

- **NUMA Data Placement** On a NUMA system there is separate RAM attached to each node. The OS decides which pages of application memory are served by which node. Our results use Linux's *preferred* NUMA allocation policy which attempts to place memory on the first node that accessed it. This works well except in cases where the memory is shared. COMPLACE mitigates the problem of sharing by co-locating threads that communicate (i.e. share memory) onto a single NUMA node.
- **Communication via Caches** By Placing threads that communicate such that they share caches, inter-cache communication traffic is reduced.
- **Working-Set Size** Placing threads with similar behaviour close to each other reduces the overall working set size for each cache, increasing their hit rates.

Investigating the performance improvement, a single run was

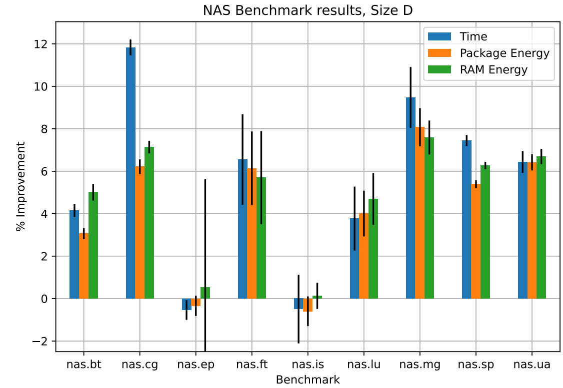

Fig. 7. Relative improvements from running COMPLACE on the NAS Benchmark suite, data from Table III, Input Size D. The mean of five runs was used to compute improvements. Error bars show the computed standard deviation, assuming results are normally distributed.

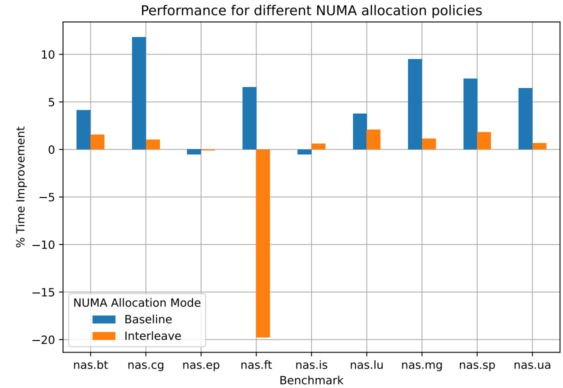

Fig. 8. Relative time improvement from running COMPLACE on the NAS Benchmark suite (size D) for different NUMA data placement policies.

TABLE III
COMPLACE RESULTS ON THE OPENMP VERSION OF THE NAS PARALLEL BENCHMARKS. VALUES ARE REPORTED NEXT TO THEIR STANDARD DEVIATION (SHOWN AS A PERCENTAGE). ORDERED BY TIME IMPROVEMENT.

Benchmark	Time (s)					Package power (J)					RAM Power (J)				
	Baseline		COMPLACE		change	Baseline		COMPLACE		change	Baseline		COMPLACE		change
	mean	std	mean	std	%	mean	std	mean	std	%	mean	std	mean	std	%
Size D															
nas.cg	385	0.4%	340	0.1%	**-11.83%**	83550	0.3%	78359	0.1%	**-6.21%**	13184	0.3%	12243	0.1%	**-7.13%**
nas.mg	105	1.4%	95	0.7%	**-9.48%**	24347	0.9%	22380	0.3%	**-8.08%**	4799	0.8%	4435	0.3%	**-7.59%**
nas.sp	788	0.3%	730	0.0%	**-7.45%**	188933	0.1%	178739	0.1%	**-5.40%**	34060	0.2%	31924	0.1%	**-6.27%**
nas.ft	528	1.3%	493	1.9%	**-6.55%**	123504	1.0%	115924	1.6%	**-6.14%**	23985	1.1%	22619	2.1%	**-5.70%**
nas.ua	702	0.4%	657	0.4%	**-6.43%**	165876	0.3%	155224	0.2%	**-6.42%**	29111	0.3%	27162	0.2%	**-6.70%**
nas.bt	755	0.3%	723	0.2%	**-4.15%**	196013	0.2%	190017	0.1%	**-3.06%**	23119	0.4%	21960	0.2%	**-5.01%**
nas.lu	541	0.6%	521	1.5%	**-3.77%**	140341	0.4%	134721	1.0%	**-4.00%**	18786	0.4%	17905	1.2%	**-4.69%**
nas.is	31	1.5%	31	0.7%	**+0.49%**	6169	0.2%	6205	0.7%	**+0.60%**	1174	0.4%	1172	0.5%	**-0.13%**
nas.ep	37	0.2%	38	0.4%	**+0.54%**	9514	0.3%	9547	0.4%	**+0.34%**	774	3.0%	770	4.1%	**-0.52%**
Size E															
nas.mg	884	1.4%	775	0.2%	**-12.33%**	202789	0.9%	184820	0.1%	**-8.86%**	39328	0.9%	35812	0.2%	**-8.94%**
nas.sp	14572	1.7%	12943	0.2%	**-11.18%**	3365211	1.3%	3087903	0.1%	**-8.24%**	585699	1.2%	529313	0.1%	**-9.63%**
nas.cg	10479	10.4%	9570	1.1%	**-8.67%**	2113666	6.7%	2002333	0.8%	**-5.27%**	399307	6.2%	379590	0.9%	**-4.94%**
nas.bt	17442	0.8%	17094	1.9%	**-1.99%**	4148724	0.8%	4046591	1.0%	**-2.46%**	653591	1.2%	634519	2.8%	**-2.92%**
nas.lu	9130	0.5%	9103	0.7%	**-0.30%**	2303784	0.5%	2282313	0.7%	**-0.93%**	347081	1.6%	346060	1.3%	**-0.29%**
nas.ep	598	0.4%	598	0.3%	**+0.00%**	152545	0.3%	152371	0.3%	**-0.11%**	12249	3.0%	12223	4.7%	**-0.21%**

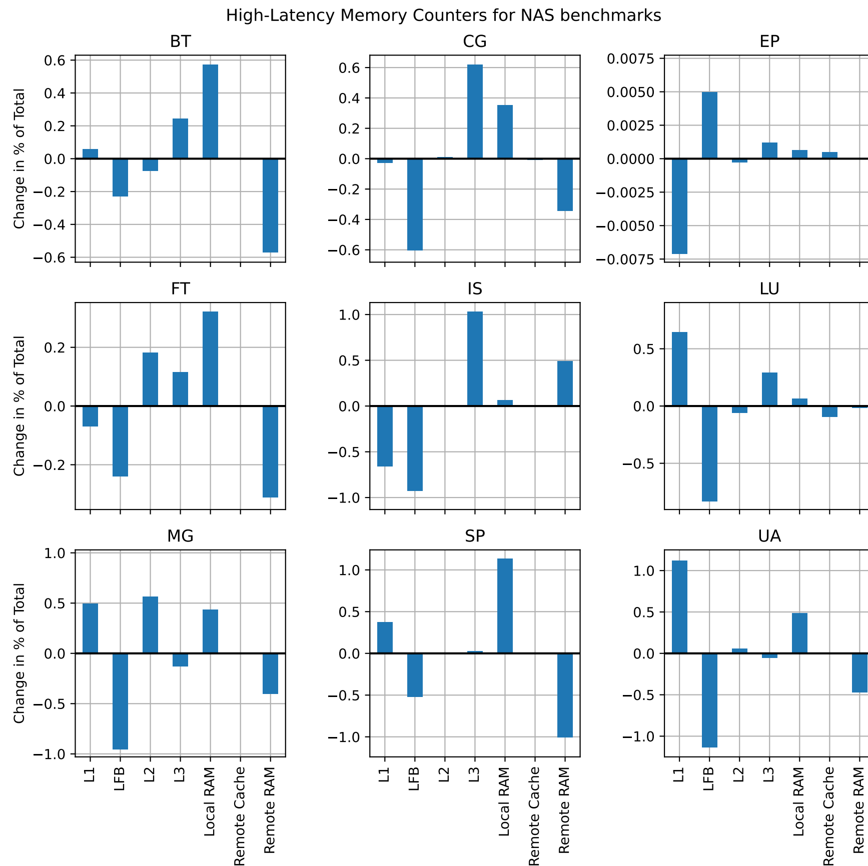

Fig. 9. Change in High-latency memory read sources for the NAS benchmark suite (size D) after applying COMPLACE. Results are sampled from Intel's "Load Latency Performance Monitoring" counters with the latency threshold set to 30 cycles. The % of samples from each source was computed for baseline and COMPLACE separately, and the difference between the two percentages is shown. Uncached and Unknown L3 misses were removed from the data.

performed with the NUMA memory allocation set to *Interleaved*. The *interleaved* policy does not use the location of requesting cores when placing memory, removing any benefit of COMPLACE due to data placement. Figure 8 shows the corresponding time improvement in each case. The benefit of our technique seems to primarily come from aiding the *preferred* data placement policy. There is a small performance improvement in the interleaved case, although for *FT*, interleaving seems to drastically decrease the performance. Figure 9 shows sampled high-latency loads using the same technique described in Section V (Figure 6). For each benchmark, the proportion of loads from each source was computed in both the COMPLACE and non-COMPLACE cases and the difference plotted. For all benchmarks except *EP* and *IS* (the two that showed no speedup), there is a reduction in the fraction of loads served by remote RAM or remote caches (caches on another NUMA node). The pattern of load traffic moving to faster RAM or caches appears responsible for the improvements seen in time and energy. It is difficult, in general, to determine the relative importance of communication and working set size on the improvement. *LU*, *MG*, and *UA* all show a significant increase in L1 hits. In these benchmarks the bulk of communication happens between immediate thread neighbours (Figure 10). COMPLACE attempts to place these neighbours on adjacent SMT threads; the increased L1 traffic is suggestive of communication happening via L1 data cache. Figure 10 shows the profiled communication pattern for for each benchmark. Clear diagonal lines indicate localised communication ideal for COMPLACE. *IS* shows non-deterministic communication after several executions, explaining its lack of improvement with COMPLACE. The similarly unresponsive *EP* benchmark has a broadcast-like communication structure that cannot be placed to exploit efficient communication paths (since all placements would involve the same number of each type of path). *CG* shows a clique-like structure but still has a strong local element. *FT* has some, limited local structure. However, Figure 9 shows that *FT* derives its benefit solely from data placement. A more robust account of this behaviour would require a change to the profiler to track memory sharing rather than communication (see Section III).

A. Measuring Energy

In all our experiments, we used RAPL (Running Average Power Limit) [9] to measure energy consumption. RAPL uses internal event counters to produce a rolling estimate of the current being drawn by the processor package and, in some models, the RAM. It appears that our technique does not lead to an out-sized improvement of energy consumption, which is surprising. Intuition tells us that different memory paths should have different energy properties – i.e. more transistor activity would be required to serve a remote memory request than a local one. In our testing, RAPL reliably reported different power (energy divided by time) values for processors executing at different manually-set processor P-states (the mechanism to run the processor at different voltage and/or frequency settings), but reported very similar power values for a broad range of workloads at a fixed P-state. In one experiment, we compared batches of 64-bit load instructions with address offsets select to create hits in L1, L2, L3 and RAM respectively[4]. As higher-latency caches were hit, RAPL reported an average power *decrease* of the processor package despite measurements at the wall socket showing a notable power increase. Consequently, we believe RAPL is not suitable for quantifying energy of communication.

VII. Related Work

There are a number of well-established techniques to collect communication profiles for message-passing applications [16, 18, 17, 4], which generally rely on intercepting MPI library calls to trace message passing and collective operations. Shared-memory communication profiling is less well-established. Some existing techniques rely on architectural simulation [6, 2]. [1] attempts to use performance monitoring counters to infer communication. The results are less precise as the counters only show a subset of the events of interest and do not include source and destination nodes. [13] uses page fault information to construct a communication profile, the resulting data is far coarser in space and time than competing techniques. [7] suggests using the Translation Lookaside Buffer (TLB) contents but their work uses features not present in contemporary architectures. State-of-the-art shared-memory communication profiling requires architectural simulation despite the massive overheads involved. Dynamic binary instrumentation approaches (such as ours) can improve on this both in performance and precision. Some work has attempted to characterise application communication, such as [5, 2, 25]. [15] also defines a number of communication-based metrics to understand applications. [12] uses “meta-models” to predict the impact of placement decisions. In [14], Diener et al. classify affinity-based mapping mechanisms for shared memory systems. A survey of scheduling techniques for addressing shared resources in multi-core processors is given in [27]. [15, 6] use communication data to optimise thread and data-mapping. We offer a simpler, greedy placement algorithm that handles thread and (implicitly) data placement. [20] undertakes the task of manually modifying applications to improve communication properties. We hope that our profiling technique can be used by developers to expedite such work in future. A good overview of both hardware mechanisms and software tools for controlling energy consumption, with a special focus on their use in scientific applications, is given in [8]. Nornir [11] is a C++-based framework for self-adaptation, that allows implementation of dynamic policies for enforcing performance and power consumption of parallel applications De Sensi in [10] proposes a method, based on multiple linear regression, for predicting power consumption and performance of applications based on the number of cores used the frequency of these cores. On the software

[4]All processors simultaneously executed an equivalent batch of loads, but to unique cachelines. The batch size was selected to ensure the instructions fit in both the processor’s ICache and Uop Cache, loop counts were set $\approx$0.5ms delay between expensive timing and synchronisation operations.

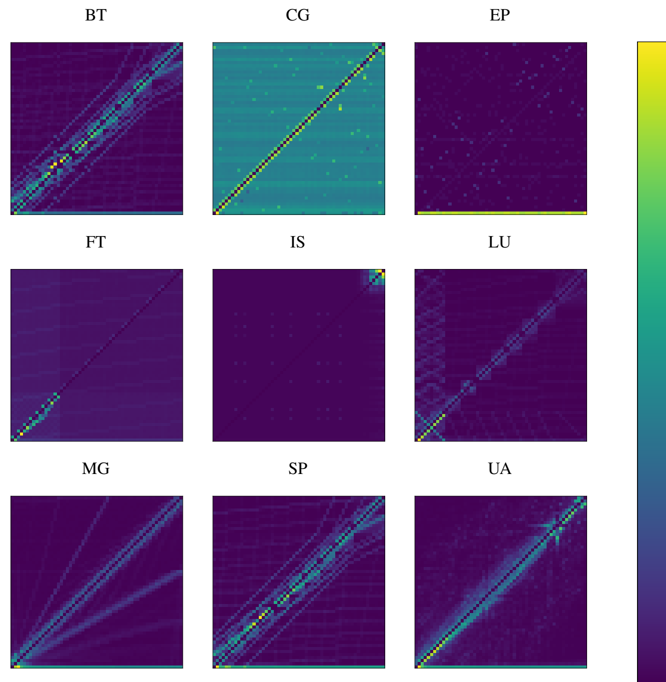

Fig. 10. Communication profiling results for the NAS benchmarks for 56 threads. Profiling was done on the smallest problem instance. The writing thread is plotted on the Y axis and reading thread on the X axis. Colour shows the amount of communication between the two threads relative to the maximum communication between any two threads for that benchmark; zero communication (purple) to maximum communication (yellow).

side, Torquati et al. [26] propose a technique for controlling power consumption based on switching between blocking and non-blocking concurrency protocols for parallel patterns based on queues (such as parallel pipeline). Melot et al. [22] propose a method for resource allocation, task mapping and dynamic voltage and frequency scaling (DVFS) for streaming applications, in order to achieve energy-efficient execution within given performance constraints. In [21], static scheduling for embarrassingly parallel applications is considered, that takes into account cores grouping for core allocation, mapping and DVFS, allowing tighter mapping of tasks to cores and allowing turning off of unused or idle cores.

VIII. Conclusions and Future Work

In this paper, we presented COMPLACE, a novel profiling and thread placement technique for shared-memory systems. Our COMPLACE tool comprises two phases: i) a novel profiling tool, COMPROF, for understanding patterns of communication between threads in a parallel application; and ii) a runtime optimisation algorithm that uses the communication graph to map the application threads to the cores of a system. We demonstrated our COMPLACE tool on the *Ocean* benchmark from the SPLASH2X benchmnark suite from Parsec, and also on the NAS benchmark suite. With our COMPLACE tool, we were able to achieve improvements of up to 12% in the execution time and up to 10% in the energy consumption. Our technique does not require any modification to the source code of the application, or any recompiling of the application code and has no runtime overhead. There are a number of future directions for this work. COMPROF is currently limited to x86 processors, extending the support to new architectures would to improve ease-of-use for users running non-x86 workloads (note, however, that the profiling phase does not have to be run on the same platform as the main running phase). We aim to gain a better understanding of the obtained RAPL energy measurements and the relation between communication and energy-efficiency in HPC workloads. COMPROF could be extended with a profiling mode designed for memory-sharing rather than inter-thread communication to drive explicit data-placement decisions. We want to evaluate our technique on other platforms and test on graph-based benchmarks. We are also missing a robust comparison of offline communication profiling techniques, such as ours, to online ones based on memory performance counters such as [1].

Acknowledgements

This work was generously supported by UK EPSRC *Energise*, grant number EP/V006290/1.

References

[1] Reza Azimi et al. "Enhancing operating system support for multicore processors by using hardware performance monitoring". In: *ACM SIGOPS Operating Systems Review* 43.2 (2009), pp. 56–65.

[2] Nick Barrow-Williams, Christian Fensch, and Simon Moore. "A communication characterisation of splash-2 and parsec". In: *2009 IEEE international symposium on workload characterization (IISWC)*. IEEE. 2009, pp. 86–97.

[3] E Barszcz et al. “The NAS parallel benchmarks”. In: *The International Journal of Supercomputer Applications*. Citeseer. 1991.

[4] Hu Chen et al. “MPIPP: an automatic profile-guided parallel process placement toolset for SMP clusters and multiclusters”. In: *Proceedings of the 20th annual international conference on Supercomputing*. 2006, pp. 353–360.

[5] Sucheta Chodnekar et al. “Towards a communication characterization methodology for parallel applications”. In: *Proceedings Third International Symposium on High-Performance Computer Architecture*. IEEE. 1997, pp. 310–319.

[6] Eduardo Henrique Molina da Cruz et al. “Using memory access traces to map threads and data on hierarchical multi-core platforms”. In: *2011 IEEE International Symposium on Parallel and Distributed Processing Workshops and Phd Forum*. IEEE. 2011, pp. 551–558.

[7] Eduardo HM Cruz, Matthias Diener, and Philippe OA Navaux. “Using the translation lookaside buffer to map threads in parallel applications based on shared memory”. In: *2012 IEEE 26th International Parallel and Distributed Processing Symposium*. IEEE. 2012, pp. 532–543.

[8] Daniele D’Agostino et al. “Hardware and software solutions for energy-efficient computing in scientific programming”. In: *Scientific Programming* 2021 (2021).

[9] Howard David et al. “RAPL: Memory power estimation and capping”. In: *2010 ACM/IEEE International Symposium on Low-Power Electronics and Design (ISLPED)*. IEEE. 2010, pp. 189–194.

[10] Daniele De Sensi. “Predicting performance and power consumption of parallel applications”. In: *2016 24th Euromicro International Conference on Parallel, Distributed, and Network-Based Processing (PDP)*. IEEE. 2016, pp. 200–207.

[11] Daniele De Sensi, Tiziano De Matteis, and Marco Danelutto. “Simplifying self-adaptive and power-aware computing with Nornir”. In: *Future Generation Computer Systems* 87 (2018), pp. 136–151.

[12] Nicolas Denoyelle et al. “Data and thread placement in numa architectures: A statistical learning approach”. In: *Proceedings of the 48th International Conference on Parallel Processing*. 2019, pp. 1–10.

[13] Matthias Diener, Eduardo HM Cruz, and Philippe OA Navaux. “Communication-based mapping using shared pages”. In: *2013 IEEE 27th International Symposium on Parallel and Distributed Processing*. IEEE. 2013, pp. 700–711.

[14] Matthias Diener et al. “Affinity-based thread and data mapping in shared memory systems”. In: *ACM Computing Surveys (CSUR)* 49.4 (2016), pp. 1–38.

[15] Matthias Diener et al. “Characterizing communication and page usage of parallel applications for thread and data mapping”. In: *Performance Evaluation* 88 (2015), pp. 18–36.

[16] Ahmad Faraj and Xin Yuan. “Communication characteristics in the NAS parallel benchmarks”. In: *IASTED PDCS*. 2002, pp. 724–729.

[17] JunSeong Kim and David J Lilja. “Characterization of communication patterns in message-passing parallel scientific application programs”. In: *International Workshop on Communication, Architecture, and Applications for Network-Based Parallel Computing*. Springer. 1998, pp. 202–216.

[18] Ingyu Lee. “Characterizing communication patterns of NAS-MPI benchmark programs”. In: *IEEE Southeastcon 2009*. IEEE. 2009, pp. 158–163.

[19] Chi-Keung Luk et al. “Pin: building customized program analysis tools with dynamic instrumentation”. In: *Acm sigplan notices* 40.6 (2005), pp. 190–200.

[20] Zoltan Majo and Thomas R Gross. “(Mis) understanding the NUMA memory system performance of multithreaded workloads”. In: *2013 IEEE International Symposium on Workload Characterization (IISWC)*. IEEE. 2013, pp. 11–22.

[21] Nicolas Melot, Christoph Kessler, and Jörg Keller. “Improving energy-efficiency of static schedules by core consolidation and switching off unused cores”. In: *Parallel Computing: On the Road to Exascale*. IOS Press, 2016, pp. 285–294.

[22] Nicolas Melot, Christoph Kessler, and Jorg Keller. “Voltage island-aware energy-efficient scheduling of parallel streaming tasks on many-core CPUs”. In: *2020 28th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP)*. IEEE. 2020, pp. 157–161.

[23] Nicholas Nethercote and Julian Seward. “Valgrind: a framework for heavyweight dynamic binary instrumentation”. In: *ACM Sigplan notices* 42.6 (2007), pp. 89–100.

[24] Konstantin Serebryany et al. “{AddressSanitizer}: A Fast Address Sanity Checker”. In: *2012 USENIX Annual Technical Conference (USENIX ATC 12)*. 2012, pp. 309–318.

[25] Jaswinder Pal Singh, Edward Rothberg, and Anoop Gupta. “Modeling communication in parallel algorithms: A fruitful interaction between theory and systems?” In: *Proceedings of the sixth annual ACM symposium on Parallel algorithms and architectures*. 1994, pp. 189–199.

[26] Massimo Torquati et al. “Power-aware pipelining with automatic concurrency control”. In: *Concurrency and Computation: Practice and Experience* 31.5 (2019), e4652.

[27] Sergey Zhuravlev et al. “Survey of scheduling techniques for addressing shared resources in multicore processors”. In: *ACM Computing Surveys (CSUR)* 45.1 (2012), pp. 1–28.