Improving GHC Haskell NUMA Profiling

Ruairidh MacGregor
Phil Trinder
University of Glasgow
Glasgow, United Kingdom
Phil.Trinder@glasgow.ac.uk

Hans-Wolfgang Loidl
Heriot-Watt University
Edinburgh, United Kingdom
H.W.Loidl@hw.ac.uk

Abstract

As the number of cores increases, Non-Uniform Memory Access (NUMA) is becoming increasingly prevalent in general purpose machines. Effectively exploiting NUMA can significantly reduce memory access latency and thus runtime by 10-20%, and profiling provides information on how to optimise. Language-level NUMA profilers are rare, and mostly profile conventional languages executing on Virtual Machines (VMs). Here we profile, and develop new NUMA profilers for, a functional language executing on a runtime system (RTS).

We start by using existing OS and language level tools to systematically profile 8 benchmarks from the GHC Haskell nofib suite on a typical NUMA server (8 regions, 64 cores). We propose a new metric: *NUMA access rate* that allows us to compare the load placed on the memory system by different programs, and use it to contrast the benchmarks. We demonstrate significant differences in NUMA usage between computational and data-intensive benchmarks, *e.g.* local memory access rates of 23% and 30% respectively. We show that small changes to coordination behaviour can significantly alter NUMA usage, and for the first time quantify the effectiveness of the GHC 8.2 NUMA adaption.

We identify information not available from existing profilers and *extend* both the *numaprof* profiler, and the GHC runtime system to obtain three new NUMA profiles: *OS thread allocation locality*, *GC count (per region and generation)* and *GC thread locality*. The new profiles not only provide a deeper understanding of program memory usage, they also suggest ways that GHC can be adapted to better exploit NUMA architectures.

CCS Concepts: • **Software and its engineering → Parallel programming languages**; **Functional languages**.

FHPNC'21, August, 2021, Virtual
© 2021 Association for Computing Machinery.
ACM ISBN 978-x-xxxx-xxxx-x/YY/MM. . . $15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

Keywords: Haskell, GHC, NUMA, Profiling

ACM Reference Format:
Ruairidh MacGregor, Phil Trinder, and Hans-Wolfgang Loidl. 2021. Improving GHC Haskell NUMA Profiling. In *Proceedings of Functional High-Performance and Numerical Computing (FHPNC'21).* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 Introduction

Non-uniform memory access (NUMA) architectures provide performance scalability for many-core machines by partitioning the memory into regions, each associated with several cores. Every core has access to all memory regions, but access to the memory in the local region is both faster (currently 2× to 4×) and higher bandwidth (currently 2×) than accessing memory in a remote region. As the number of cores in an architecture rises above 16, NUMA is the best way to provide fast access to shared memory. Although NUMA architectures have been around for more than two decades they have historically been restricted to specific applications like Data Analytics. More recently, as the number of cores in many commodity architectures increases, they are becoming the dominant architecture for general purpose platforms, *e.g.* clusters of small (4 or 8 region) NUMAs are very common server architectures, and they are coming to laptops, for example with the AMD Ryzen processors.

In languages with explicit memory management, like C/C++, the programmer herself lays out memory, and may go to great lengths to exploit NUMA. However this is additional programming effort, and must be repeated for each NUMA configuration.

Languages with automatic memory management such as Haskell, Java or Python give the programmer far less control over memory usage: allocation, layout, and locality. For these languages the *challenge* is to minimise the cost of non-local memory access [6]. Effective use of NUMA is necessary as studies show that optimisation can reduce runtimes of applications in conventional languages by 10–20%, *e.g.* [14, 18]. The performance gains are likely even greater for functional languages, such as Haskell, with their massive memory residencies and high (de)allocation rates. Automatic memory management does, however, provide *opportunities* to benefit from NUMA as the implementation is free to move data around to better exploit locality, and doing so will be invisible in the program. Moreover the program will not require refactoring for new NUMA configurations.

Informative profiling of NUMA usage is essential to enable effective optimisation, and is available from multiple layers in the system stack. At the lowest levels there are hardware performance counters and associated tools like *Intel VTune*[1] and *AMD μprof*[2]. At the OS level are profilers like Linux *perf tools* and *numaprof* [20]. Profilers for languages with automatic memory management are not common, and *PerfUtil* [14] for the JVM is a notable exception.

This paper makes the following research contributions.

The systematic profiling of NUMA memory usage of parallel GHC Haskell programs using existing compiler & OS tools (Section 4). Eight benchmarks from the GHC *nofib* suite are profiled, and three are selected for presentation: two versions of the data intensive sumeuler, and the compute intensive prsa encryption engine. The study uses the GHC heap profiler and the OS-level *numaprof* tool to provide *the first ever quantification of the effectiveness of GHC NUMA adaption.* That is, the policy of local allocation is effective in increasing local access from a notional 12.5% to around 30% for data intensive benchmarks, and to around 23% for a compute intensive benchmark. We record and analyse two standard metrics: memory *residency* and *allocation*; and two NUMA-specific metrics: *OS thread access locality* within NUMA regions, and *OS thread NUMA region counts*. We propose a new metric: **NUMA access rate** that allows us to compare the load placed on the memory system by different programs, and use it to contrast the benchmarks. The study reveals key information about GHC Haskell's NUMA usage not available from current profilers, and motivates the following contributions.

For a thread executing in some region current NUMA profilers record *access* to memory in each region, but not the *allocation* of memory into each region. We **design, implement, and demonstrate an extension to *numaprof* to record *OS thread allocation locality*** (Section 5). The extensions record the NUMA region of the calling thread and the bytes allocated into each region. The output is integrated with the *numaprof* UI to show a heat map of allocations between regions. The profiles allow developers to detect, and potentially control, remote allocation. They also provide evidence, not available from existing profilers, to explain why some regions have very high access rates. The extension to *numaprof* is freely available.

The design, implementation, and demonstration of extensions to GHC garbage collection (GC) to profile NUMA usage (Section 6). While GC in the current GHC RTS reports global statistics like GC counts, and GC runtimes, it provides no information on NUMA usage. We extend GHC's GC to record *GC count per region* and *GC count per generation in each region* enabling the programmer to identify regions with high memory residency or high allocation rates. We also extend GHC's GC to record *GC thread locality*, analogous to *access locality*, using the *numaprof* UI to produce a region × region heat map that identifies opportunities to improve locality during GC by reallocating objects between regions. The extension to the GHC RTS is freely available.

2 Related Work

2.1 NUMA Architectures

The technology of *Non-uniform memory access* (NUMA) machines was developed concurrently by several companies in the 1990s for super computers. NUMA architectures aim to provide performance scalability for shared memory machines by physically partitioning the memory of the system into *regions* each associated with several cores, as depicted in Figure 1. Every core has access to all memory regions, but access to the memory in the local region is via a bus and is both faster (currently 2× to 4×) and higher bandwidth (currently 2×) than accessing memory in a remote region where the memory access must go via an interconnect [9].

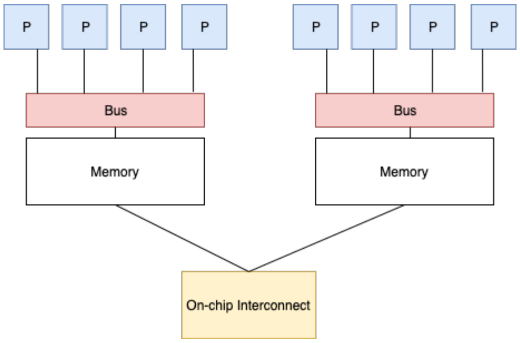


Figure 1. A NUMA architecture showing 2 regions, 4 cores per region. Cores access local memory via a bus and access remote regions via the on-chip interconnect.

NUMAs come in varying sizes. Clusters of small NUMAs (4 or 8 regions) are already the dominant server architecture, and it is widely predicted that in the next 7 years mid-size (32 – 256 cores, 2 – 16 regions) NUMA architectures will become increasingly common in servers and desktop/workstations. Some laptops already use AMD Ryzen NUMA architectures.

2.2 Challenges of Automatically Managing NUMA

Languages with explicit memory management, *e.g.* C or C++, offer full control over the NUMA memory usage of the application. A great deal of programming effort is required to efficiently exploit NUMA. This may introduce bugs, especially if the language is not memory safe [8, 17]. In addition

[1] https://software.intel.com/content/www/us/en/develop/tools/oneapi/components/vtune-profiler.html
[2] https://developer.amd.com/amd-uprof/

to the increased programming effort, the code generally isn't portable and must be re-written for NUMAs of a different size. Therefore, automating the management of NUMA usage is highly desirable.

Languages with automatic memory management such as Haskell and Java provide far less control over the NUMA usage of the application. The burden of managing memory allocation, layout and locality is left to the RTS implementation. A recent study suggests that NUMA is an issue for languages with automatic memory management [14], and are identified as: (1) NUMA aware garbage collection; (2) scheduling threads onto the region where the data used resides; (3) avoiding high memory controller utilisation.

Although here we focus on NUMA usage at the language level it is worth mentioning that Operating System (OS) or Hypervisor tools can allocate memory to exploit NUMA [22]. Here a common heuristic is *first touch*, that allocates memory into the region where the first thread to request it is executing. The OS can also pin processes to regions to assist with moving processes (OS threads) close to the data. While this may improve performance, it may also lead to load imbalance and hence it is often better to allocate in a nearby remote region [12].

2.3 Opportunities Provided by Automatically Managing NUMA

Automating the management of NUMA memory provides opportunities for portable performance across NUMA platforms. The garbage collector (GC) is free to move data across regions. For example, if the system detects that a lot of remote accesses are made to a region, then the GC can move the data closer to threads that use it.

Automating the memory management also massively reduces programming effort, reducing development time by simplifying code and improving maintainability. Moreover the code will not require refactoring for a new NUMA.

In the context of the OpenMP RTS, Broquedis *et al* [4, 16] use live profiling information and a graph-based compute model to efficiently reduce data transfers between NUMA regions. They report average performance improvements of 1.12× on a 288-core shared-memory system.

2.4 Memory Management in Parallel Haskell

Haskell is a purely functional, non-strict programming language, and ideal for a study of automatic NUMA memory management as Haskell applications typically have much higher memory requirements than imperative languages. This is due to two main factors. (1) State is immutable and the only way to modify a data structure is to allocate a new one with updated values. This leads to a lot of temporary, short-lived data. (2) Lazy evaluation results in the continuous allocation and de-allocation of closures, *i.e.* thunks representing Haskell expressions. The

Memory management in the RTS of the Glasgow Haskell Compiler (GHC) uses a generational policy, by default with 2 generations (for young and old objects) and a separate heap area for data (static objects) from top level constant expressions (CAFs) that reside outside the generational structure. The RTS maintains a *Haskell Execution Context (HEC)* for each processor. This contains all the data required for a sparked thread to execute its share of the program code. Depending on the parallel Haskell dialect, most of the parallel aspects of the execution is implicitly handled by the RTS. In the Glasgow parallel Haskell (GpH) dialect, constructs on source code level indicate potential parallel evaluation of an expression, which is represented as a *spark* in the RTS. These are turned into *sparked threads* on-demand, *i.e.* when the RTS perceives the need for more parallelism. Since GHC 8.2 sparked threads are pinned to the HEC's region. In contrast, the main thread, which is not generated from a spark, is unpinned, and thus its location (region) may vary between runs of a program. An abstraction layer of evaluation strategies helps to orchestrate the (parallel) execution of the program and the forcing of data structures.

In Haskell NUMA memory is managed by the RTS, and identifying where allocations occur in the code can be difficult in a lazy language. In most other languages, such as Java, allocations are more easily linked to lines in the source code. This creates the need for strong NUMA profiling in Haskell.

2.5 Adapting languages with automatic memory management to NUMA

There has been a significant body of research aiming to adapt Java implementations to NUMA. Recent studies focus on characterising the garbage collector's scalability bottlenecks or on introducing various NUMA-aware thread scheduling and memory management policies, as follows.

Gidra's GiC garbage collector (GC) [7] for Big Data applications executing on the JVM, uses a mostly distributed design, utilising message passing techniques. GC threads normally only collect within the region that they reside in. Despite the additional overhead, NumaGiC achieves end-to-end performance improvements of up to 45%, and reduces GC time by up to 5.4×, on mid-size (*e.g.* 48-core 8-region) AMD and Intel machines.

Alnowaiser shows that in the Java Virtual Machine (JVM), on average, 80% of the vertices in a garbage collection *reference graph* all reside in the same region [2]. *Reference graphs* are directed graphs, composed by tracing references from a root object to all objects reachable from the root. Alnowaiser & Singer incorporate this knowledge into the JVM garbage collector [3], by pinning a GC thread to the region of a root in the reference graph, demonstrating speedups by up to 2.5×, relative to not adding region locality.

The GUMSMP Haskell RTS is designed for NUMA and clusters of NUMA [1]. It has a mostly distributed design that utilises shared-memory within a region and message passing between regions. On NUMA GUMSMP delivers better performance compared with the standard GHC RTS that uses only shared memory parallelism.

2.6 NUMA Profiling Levels

Hardware Performance Counters are at the lowest level of the system stack. They typically capture compute cycles, L1/L2 cache misses, and NUMA local/remote access counts. Applications can interact with hardware counters via sampling. Relevant NUMA tools using these counters include the *numastat*/*numatop*[3] tools for Linux on Intel Xeon. *Intel VTune* and *AMD μProf* provides native support for Intel and AMD architectures respectively, offering rich graphical interfaces and data export capabilities. The *PAPI* interface [19] provides a uniform API across different hardware vendors to allow programs to access hardware counter information. The Linux *perf tools*[4] provide access to hardware counters but are currently restricted to Intel-based architectures.

OS Level Profiling Tools build on hardware counters, providing more abstraction and adding flexibility. For example to track not only the memory location accessed, but also the region of the executing thread, making it possible to discover the *access & allocation locality* of OS threads.

The latency for accessing different regions in a NUMA architecture varies, and Figure 2 shows the relative (access) latencies between regions on Togian (the machine used in our experiments, Section 3.2). We use *numactl* to access this information: a command-line front-end that accesses hardware architecture information through the *hwloc* library from the Linux kernel. On AMD architectures *numaprof* [20] records *OS access thread locality* using the *Intel Pin Tool* [11], and is the tool we use in the experiments in Section 4 and Section 5.

$$\begin{bmatrix}
10 & 16 & 16 & 22 & 16 & 22 & 16 & 22 \\
16 & 10 & 22 & 16 & 16 & 22 & 22 & 16 \\
16 & 22 & 10 & 16 & 16 & 16 & 16 & 16 \\
22 & 16 & 16 & 10 & 16 & 16 & 22 & 22 \\
16 & 16 & 16 & 16 & 10 & 16 & 16 & 22 \\
22 & 22 & 16 & 16 & 16 & 10 & 22 & 16 \\
16 & 22 & 16 & 22 & 16 & 22 & 10 & 16 \\
22 & 16 & 16 & 22 & 22 & 16 & 16 & 10
\end{bmatrix}$$

Figure 2. Relative access latencies (distances) between regions on Togian, the 8 region NUMA used. Accessing a remote region increases latency from 10 to 16 or 22 (data from *numactl*).

Other OS level profilers include the following. Linux *perf tools*[4] captures some events at OS level. *MemProf* [10] is a higher level tool that focuses on NUMA access patterns using a kernel module and AMD's Instruction Based Sampling [5] to reduce overhead. *NumaPerf* [23] tracks sharing patterns between threads, instead of remote accesses, to identify load imbalances and potential thread migrations.

Language Implementation Tools. The VM or RTS for a language with automatic memory management has more information about memory usage patterns of an application than the OS. The state of all objects is tracked within the language implementation and potentially provides scope for identifying hotspots and making optimisations.

Very few languages with automatic memory management provide NUMA profiling, and the work that has been done mostly uses the VMs for managed languages. For example *PerfUtil* [14] provides NUMA specific metrics for the JVM, and was measured on a small 2-region NUMA. GHC Haskell currently has no NUMA profiling, and in Section 6 we describe extensions to record *per region GC counts* and *GC thread locality* measures. The new RTS-level NUMA profiling is novel both in profiling a non-strict functional language that places massive demands on the memory system, and in profiling programs executing under a RTS rather than a VM.

3 NUMA Metrics and Experiment Setup

This section defines the NUMA metrics used and describes the GHC Haskell experiment methodology.

3.1 Metrics

We start by defining two standard and five NUMA specific memory usage metrics.

Residency: is the number of bytes of live (non-garbage) data in the heap during program execution. Residency is commonly plotted against execution time, and the maximum residency is a key statistic.

Total Allocation: is the product of memory allocated and execution time, typically measured as bytes*s. In GHC allocation is attributed to cost centres in the program source, and allocation is plotted as a graph of bytes against execution time. The slope of these graphs is the **Allocation Rate** typically measured as bytes/s

OS thread access locality: is an access matrix, where entry (i, j) is the total number of accesses made by threads in region i to data in region j. In *numaprof* the data is normalised with respect to all accesses and visualised as a heat-map.

OS thread NUMA region counts: is the number of accesses made by each thread at different access latencies (10, 16 or 22 on Togian as in Figure 2), recorded by *numaprof.*

OS thread allocation locality: is an allocation matrix where entry (i, j) is the total number of bytes allocated by OS threads in region i to region j. We extend *numaprof* to record this, and present the results as a heat map.

GHC GC count per region & generation: is the number of times a GC occurs in a particular region and generation. Here we extend GHC's GC to record the information, and

[3] https://man7.org/linux/man-pages/man8/numastat.8.html
[4] https://perf.wiki.kernel.org/index.php/Main_Page

present the results as a table. This metric may highlight load imbalance if some regions have far more GCs.

GHC GC thread locality: is a matrix where entry (i, j) records the number of times a GC thread in region i processes data in region j. Here we extend the GHC’s GC to record the information, and present the results as a table. This metric identified what percentage of objects processed by the GC are local, and what percentage are remote.

3.2 Experimental Setup

All measurements were made on a typical NUMA server, and specifically the **Togian** Linux server at the School of Computing Science at University of Glasgow. Togian features 64 cores, using the AMD™ Opteron Processor 6366 HE; 64GB RAM, 8 NUMA regions so 8GB per region; 8 cores per region; running CentOS Linux 7 (Core). All applications are compiled with GHC version 8.4.3, although this version is relatively old there have been no NUMA adaptions since its release. To account for non-deterministic scheduling and other factors: (1) reported runtimes are the median of 5 measurements, and (2) *all profiles are collected from the same program execution* with the exception of the GHC heap profiles.

3.3 Benchmarks

A set of eight benchmarks from the parallel *nofib* [15] suite have been profiled, and the profiles are publicly available[5]. In the remaining sections we use three versions of two benchmarks as running examples.

The sumeuler benchmark is data intensive: creating and traversing huge lists. The computation maps the Euler totient function (ϕ) over the list, and then computes the sum of the results. We use two implementations of this benchmark using two different evaluation strategies. Namely, *divide & conquer* (DNC) and a *data parallel* (data parallel) method, with a focus on earlier evaluation. These two methods of computation were chosen from a number of *nofib* implementations as the DNC has the greatest runtime and data parallel the least. The runtimes and speedups with different inputs are listed in Table 1 and Table 2, and other runtimes are available online [5]. The relatively low parallel efficiencies, *i.e.* speedups of 33.2 and 37.7 on 64 cores for DNC and data parallel versions, are typical of GHC on large NUMA [1]. The DNC implementation is not part of the *nofib* suite however, and uses the divConq algorithmic skeleton from [13]. The input size is set to 100K to provide good speedups while minimising profiling time. Already *numaprof* imposes a 40-100× slowdown, and the GHC profiling extensions raise this to a 500× slowdown.

The prsa benchmark is compute intensive, and drawn from the GHC *nofib* suite. It is a parallel implementation of the RSA public key encryption algorithm. The input used is 300K, and the runtimes on 1 core and 64 cores are 2.43s and 0.45s respectively. For the purposes of profiling the modest parallel performance isn’t an issue: often programs are profiled to improve their performance.

Input Size	Runtime (s) (1 core)	Runtime (s) (64 cores)	Speedup
50K	80.97	3.85	21.0
100K	341.03	13.91	24.5
150K	787.34	29.70	26.5
200K	1428.23	43.01	33.2

Table 1. Divide-and-Conquer (DNC) sumeuler runtimes & speedups.

Input Size	Runtime (s) (1 core)	Runtime (s) (64 cores)	Speedup
50K	81.28	2.55	31.9
100K	341.89	9.56	35.8
150K	787.39	21.23	37.1
200K	1435.31	38.04	37.7

Table 2. Data parallel (DP) sumeuler runtimes & speedups.

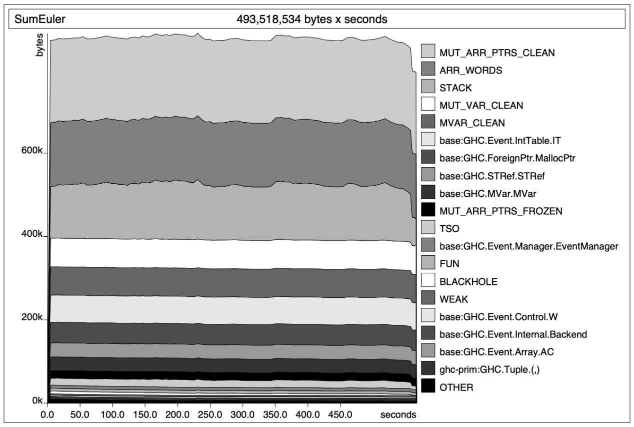


Figure 3. DNC sumeuler Heap Profile (GHC).

4 NUMA Profiling GHC with Existing Tools

4.1 Heap Profiles

Figure 3 shows the memory profile for the divide-and-conquer (DNC) sumeuler recorded by the GHC memory profiler. Heap residency is recorded against execution time, and is attributed to cost centres in the program. The maximum residency is less than 900KB, and the total allocation is 493MBs. Heap residency is fairly consistent throughout execution. In the DNC implementation, each of the sparked threads generates an interval list to process, and the allocation and residency

[5] https://github.com/ruairidhm98/Profiles

likely represents the interval lists being generated, and consumed.

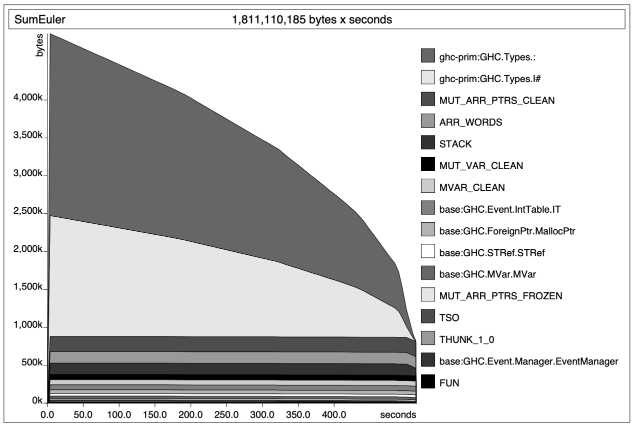


Figure 4. Data parallel sumeuler Heap Profile (GHC).

Figure 4 shows the memory profile for data parallel sumeuler. Compared to the DNC profile the maximum residency is more than 5× higher at around 5MB, and total allocation is 3× higher: 1811MBs. In this version of the program the main thread creates the entire list before sparking tasks to process intervals of it. This is reflected in the profile where in the first 1 or 2s the program allocates a large amount of memory (5MB), thereafter heap residency declines throughout execution as the sparked threads consume the list. So in contrast to the DNC version, little of the allocation is performed by the sparked threads.

Figure 5 shows the memory profile for prsa. For this compute intensive program the maximum residency is fairly low at 1.7MB, and total allocation is very low at 8.45MBs. Residency shows a different pattern again: it steadily increases during execution, and the increase is fairly uniformly attributed to cost centres.

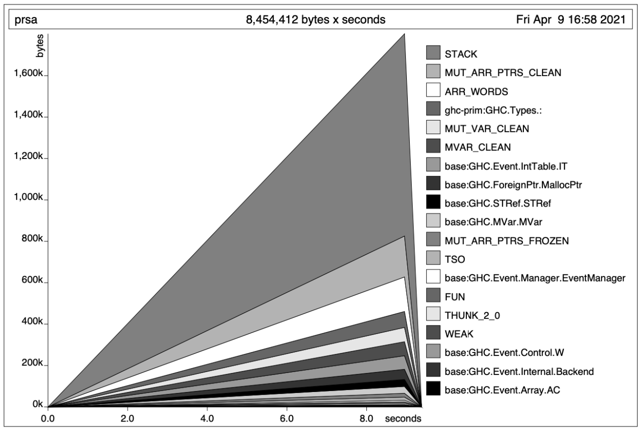


Figure 5. prsa Heap Profile (GHC).

GHC overall heap profiles are oblivious to NUMA, and hence lack key information about parallel execution on NUMA. It is not apparent what memory regions hold the resident objects, which threads allocate and access those objects, nor what region those threads execute in. Fortunately some OS level tools can provide some additional information.

4.2 OS Thread Access Locality

Figure 6a shows the OS Thread Access Locality heat-map for DNC sumeuler produced by *numaprof*. Here the colour of the cells should be interpreted as follows: grey means zero so no accesses from one region to the other, blue is a small number of accesses, purple is more accesses, and red is a large number of accesses. So the purple diagonal shows significant local access within each region: 28.7% of accesses. These are likely data that GHC allocates locally, so for sumeuler the list of integers being searched by each thread. However, without information about thread memory allocation we can only speculate, and this *motivates our extension of numaprof* in Section 5. The red of column 7 shows that some 82% of all memory accesses are to this region (7).

Figure 6b shows the OS Thread Access Locality heat-map for the data parallel sumeuler. As in Figure 6a it has a purple diagonal showing slightly more local accesses (29.6% of accesses), and a hot column, in this case in region 3 (81% of accesses).

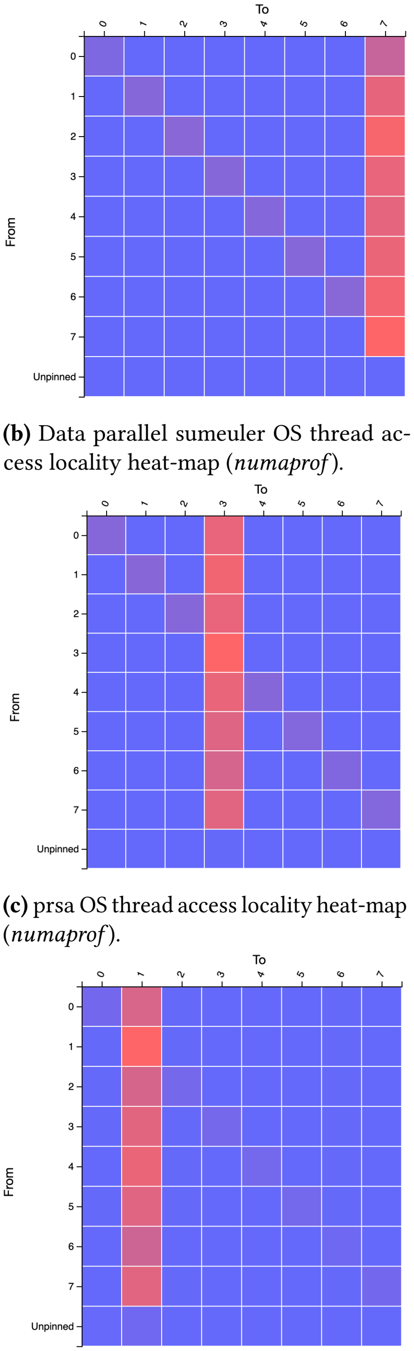
Figure 6c shows the OS Thread Access Locality heat-map for prsa. As in the sumeuler benchmarks the purple diagonal shows some local accesses, but slightly fewer (22.9% of accesses). The hot region (1) accounts for 89.4% of accesses. This accords with our expectations: in a compute intensive program like prsa sparked threads do fewer memory accesses, and likely fewer allocations: a conjecture that is suggested by the total allocation (Figure 5), and that our new profiler in Section 6 allows us to confirm: the greatest amount of allocation within a single region is from an unpinned thread, and is just 9444 bytes.

4.3 OS Thread NUMA Access Latencies

The different NUMA access latencies are outlined in Figure 2, and are 10, 16 and 22 on Togian. Figure 7 shows the NUMA access latency counts for both DNC & data parallel sumeuler recorded by *numaprof*. For DNC sumeuler 28.7% of memory accesses are in the same region with latency 10, and also reported in the access locality heat map. The majority of accesses (41.6%) are at a latency of 16, and some 29.7% are at latency 22. For data parallel sumeuler 29.6% of memory accesses are at latency 10. As for DNC most of the accesses are at latency 16 (41.3%), with 29.1% at latency 22. Figure 8 shows the NUMA access counts for prsa. 22.9% of memory accesses are at latency 10. Most of the accesses are at latency 16 (45.3%), with 30.8% at latency 22.

From the access latency matrix Figure 2, we see that entirely random accesses between regions would give 12.5% latency 10 accesses (i.e. 1 out of 8 regions), 50% latency 16 accesses (*i.e.* 4 out of 8 regions), and 37.5% latency 22 accesses

(a) DNC sumeuler OS thread access locality heat-map (*numaprof*).



(b) Data parallel sumeuler OS thread access locality heat-map (*numaprof*).

(c) prsa OS thread access locality heat-map (*numaprof*).

Figure 6. OS thread access locality heat-maps

(*i.e.* 3 out of 8 regions). We conclude that *the existing GHC policy of preferring local allocation is effective in increasing local access* from the notional 12.5% to around 30% for data intensive benchmarks, and to around 23% for a compute-intensive benchmark. The increased number of local accesses reduce access to remote regions at latencies 16 and 22 approximately equally. These profiles, together with the results in Table 3, are *the first ever quantification of the effectiveness of GHC NUMA adaption.*

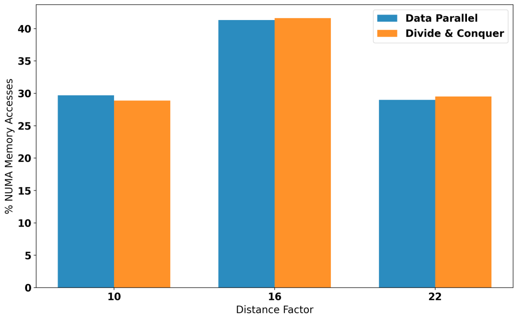


Figure 7. OS thread NUMA region counts for sumeuler (*numaprof*).

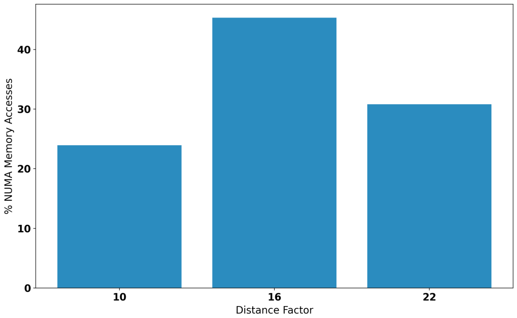


Figure 8. OS thread NUMA region counts for prsa (*numaprof*).

We propose **a new metric to characterise the load a program places on a NUMA memory system**. A program's **NUMA access rate** is the sum of memory accesses weighted by their NUMA latency and divided by program runtime, with unit object accesses × latency/s. So on a NUMA with access latencies or distances d_i (10, 16 and 22 on Togian), a program that makes a_i accesses at each distance in runtime t seconds has

$$\text{NUMA access rate} = \frac{\Sigma_i d_i * a_i}{t}$$

The unit is informally abbreviated to (weighted) accesses/s, and the metric allows us to *compare the load placed on the memory system by different programs.*

The NUMA access rate for divide-and-conquer sumeuler is $1223 \times 10^9/13.91s = 87.9 * 10^9$ accesses/s. In comparison the NUMA access rate for data parallel sumeuler is twice as high: $1591 \times 10^9/9.56s = 166.4 \times 10^9$ accesses/s. Clearly the initial list allocation by the main thread forces sparked threads in other regions threads to make many more remote accesses, placing a higher load on the memory system. The NUMA access rate for prsa is $255 \times 10^9/0.45s = 566.7 \times 10^9$ accesses/s showing that the large percentage of remote accesses (Figure 8) places the highest demand on the memory system.

Another potential metric is NUMA memory *allocation* rate We don't report this as allocation costs are comparatively low, despite the fact that allocation determines much larger the NUMA access rate.

Benchmark	% Local	% Hot Column	NUMA Speedup
blackscholes	11.8	60.2	1.07
DNC sumeuler	28.7	69.0	1.11
DP sumeuler	29.6	70.0	1.03
matmult	26.9	50.9	1.01
partak	38.0	60.0	1.23
prsa	22.6	76.8	1.21
queens	19.1	68.0	1.13
transclos	14.9	83.7	1.03
Geo. Mean Speedup			1.10

Table 3. Percentage local accesses, most highly accessed region, and speedup due to NUMA optimisation for 8 *nofib* benchmarks.

Multiple Benchmarks. Table 3 shows the percentage of local accesses, and the maximum percentage of remote accesses to a single region, e.g. in Figure 6a this would be the sum of $(0, 7)$ to $(6, 7)$ divided by the total number of accesses. The rightmost column shows the speedup obtained by turning on GHC 8.2 NUMA adaption to keep allocation local.

Existing Profilers Summary. Existing profilers identify patterns of GHC NUMA memory access for the 3 benchmarks. There is some locality of access, varying from 30% for data intensive sumeuler to 23% for compute intensive prsa. In each benchmark there is a single *hot* region with a high percentage of accesses identified as a red column in the heat maps. The hot region changes between executions of the same program. However, from existing profiles we can only speculate as to what is causing the *hot* region. Perhaps the imbalance is caused by how memory is allocated? We design and implement our first new NUMA profiler to explore this possibility.

5 Profiling GHC NUMA Allocation Locality with *numaprof*

In the implementations of many languages with automatic memory management the access pattern during program execution is determined by where objects are *allocated*. Once an object is allocated into some region, all accesses are directed to that region. This is similar to the OS level first-touch policy (Section 2.2). Moreover the RTS can potentially change NUMA access patterns by choosing where to allocate new objects.

No profiler was found to provide allocation locality information. The information could be obtained by extending the GHC RTS or *numaprof*. We elected to extend *numaprof* as it provides a malloc tracker in the form of the MALT tool [21]. MALT tracks the region where the data was allocated. Thus, the only extensions that were required were to: (1) track the region of the calling thread; (2) store this in an access matrix at runtime; and (3) integrate it into the *numaprof* web based UI. The implementation is freely available [6].

Figure 9a shows the new OS thread allocation locality heat map for DNC sumeuler. The red diagonal shows that all allocations made by GHC sparked threads are local, as conjectured from the access locality heat map (Figure 6a), and due to the GHC 8.2 NUMA locality optimisation. The bottom row of the heat map shows the allocations made by the main thread in GHC: the only thread that is not pinned to a specific region. The purple entry at $(7, 8)$ reveals that 97% of the bytes allocated by the unpinned thread (54334 bytes) are allocated into region 7.

Figure 9b shows the OS thread allocation locality heatmap for data parallel sumeuler. As for DNC sumeuler all allocations made by GHC sparked threads are local, and the purple entry at $(3, 8)$ reveals 97% (5.0MB) of the unpinned allocated bytes are in a single region.

Figure 9c shows the OS thread allocation locality heat-map for prsa. All allocations by GHC sparked threads are local. The diagonal is purple as the compute-intensive threads allocate less memory. The red entry in $(1, 8)$ shows that the main thread allocates a high percentage of the memory, although it's only 9.1KB.

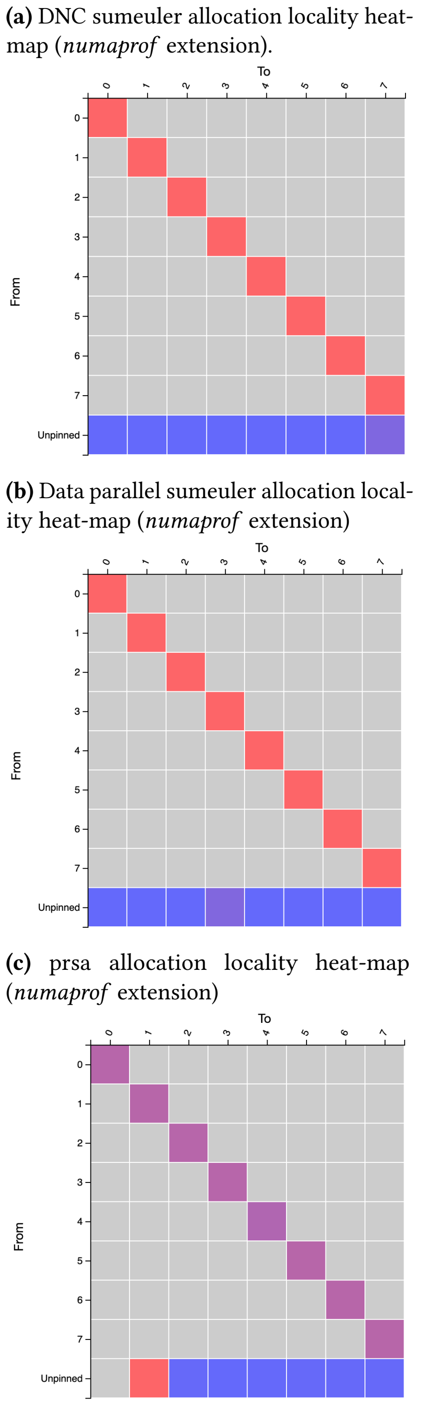
The new profiler explains the hot regions. In all three of the allocation heat maps the purple (unpinned) main thread *allocation* region (regions 8, 3 and 1 respectively) accounts for the high *access* rates seen in the corresponding access heat maps: Figure 6a, Figure 6b, and Figure 6c. Moreover we see that in prsa the unpinned main thread does far more allocation than the computationally intensive sparked threads. In both cases the new allocation profiles provide new information to confirm speculations based on access profiles.

6 Profiling NUMA In The GHC GC

6.1 Per Region GC Count

GHC garbage collection provides global statistics reporting bytes moved/copied, GC counts, GC runtimes (overall & per cycle) etc. However no information about NUMA usage is recorded. Here we extend GHC memory profiling to record *GC counts per region* and *GC counts per generation in each region*. By identifying regions where GC is occurring most frequently, these metrics allow the programmer to identify

[6] https://github.com/ruairidhm98/numaprof in branch AllocationLocality



(a) DNC sumeuler allocation locality heat-map (*numaprof* extension).

(b) Data parallel sumeuler allocation locality heat-map (*numaprof* extension)

(c) prsa allocation locality heat-map (*numaprof* extension)

Figure 9. OS thread allocation locality

imbalances in memory usage between regions, *e.g.* to identify regions with high memory residency or high allocation.

To implement the *GC counts*, the counts for each generation are stored in each Haskell Execution Context (HEC). GHC's GC is typically local to a HEC, and uses a stop-the-world policy. When a Generation 1 GC starts in a HEC, and hence in a region, a new Generation 1 counter is incremented in the HEC, and similarly for a Generation 2 GC. Storing the counts within each HEC means that no synchronisation is required as there is only one GC thread per HEC. At the end of the execution the counts for all HECs are output, and the HEC counts associated with each region are aggregated.

The second and third lines of Table 4 show the GC counts per region & generation for the DNC sumeuler, using boldface to highlight the hot region as in Figures 6a to 6c. There are a total of 3983 Generation 1 collections, and just 18 Generation 2 collections. Both Generation 1 and 2 collections are evenly spread between the regions, and this concurs with the heap profile in Figure 3 that shows constant heap residency throughout the execution. The high total allocation is confirmed in Table 5 that shows the numbers of objects accessed by each benchmark in static, Generation 1, and Generation 2 collections. It shows that DNC sumeuler accesses much more memory from Generation 1, *i.e.* newly allocated objects.

The fourth and fifth lines of Table 4 show the GC counts per region & generation for the data parallel sumeuler. There are a total of 2634 Generation 1 collections, and 49 Generation 2 collections. The increased number of Generation 2 collections compared to the DNC sumeuler is almost certainly because most allocations are done early in the program execution (Figure 4), leading to objects with long lifetimes that are promoted into Generation 2. Interestingly, Table 5 shows that GC accesses only a small number of Generation 2 objects, 1.87M compared with 3.63M for DNC. This indicates that in this memory intensive program, only a small fraction of the old objects are live, and that the amount of allocation is high enough to more frequently trigger major collections involving Generation 2.

The sixth and seventh lines of Table 4 show the GC counts per region & generation for prsa. There are a total of 647 Generation 1 collections, and 190 Generation 2 collections. Table 5 shows 0.88M Generation 1 objects accessed during GC compared with 2.51M Generation 2 objects. The small number of Generation 1 collections, and the small number of objects accessed reflects the expectation that compute intensive programs do less allocation than data intensive ones. The large number of Generation 2 collections, and greater number of objects accessed, shows that the allocated data is long-lived, and concurs with the heap profile (Figure 5).

The emboldened columns in the table correspond to the *hot* region for each benchmark execution, but the GC counts don't provide information to help understand these regions.

6.2 GHC GC Thread Locality

We also extend GHC's GC memory profiling to record *GC thread locality*, analogous to *thread access locality* (Section 4.2). For each Generation 1 or Generation 2 GC in a given region we record the location of each object accessed. As for *access*

Benchmark	Gen	Rg 0	Rg 1	Rg 2	Rg 3	Rg 4	Rg 5	Rg 6	Rg 7
DNC	1	493	643	461	509	499	408	509	**461**
sumeuler	2	4	1	2	3	2	1	2	**3**
data parallel	1	390	165	175	**432**	345	288	488	351
sumeuler	2	11	7	6	**4**	6	6	5	4
prsa	1	153	**81**	80	90	68	70	35	70
	2	49	**24**	23	22	23	20	10	19

Table 4. Garbage Collection counts per region & generation for DNC sumeuler, data parallel sumeuler, and prsa (GHC RTS extension).

Area	DNC sumeuler	Data Parallel sumeuler	prsa
Static	1.45	1.12	3.38
Generation 1	4.60	0.89	0.88
Generation 2	3.63	1.87	2.51

Table 5. Objects accessed during garbage collections (Millions) by area/generation for DNC sumeuler, data parallel sumeuler, and prsa (GHC RTS extension).

locality the results produce a region × region heat map that reveals access locality properties. Crucially, the information identifies opportunities to improve locality during GC by reallocating objects between regions, a common technique for adapting language implementations for NUMA, *e.g.* in [3, 7]. The GHC RTS extensions are freely available [7].

As for GC counts the thread locality information is recorded in each HEC to minimise synchronisation. We record the location of the GC thread (known as it is pinned to the HEC's region), and the location of each object processed by the GC thread, and use the *numaprof* visualisation technology. The profiling currently has a high overhead as it uses expensive OS system calls to discover the object locations. The overheads could be reduced by caching the object locations.

Figure 10 – Figure 12 shows the GC thread locality heat maps for the DNC sumeuler. Figure 10 is for static objects, *i.e.* top-level Constant Applicative Forms (CAFs) in the program, and shows no locality: the GC threads access all regions. This is likely due to the lazy evaluation of CAFs, as demanded by expressions in sparked threads. As sparked threads execute in all regions, the CAF objects are associated with all regions. It is likely that locality could be improved if the GC moved CAF objects into the region with most accesses.

In contrast to the lack of locality for CAFs, Generation 1 and Generation 2 collections show good locality. Figure 11 is for Generation 1 objects and shows strong locality with 86.4% of objects processed locally. Figure 12 is for Generation 2 objects and also shows strong locality (93.2%). Good locality arises as the sparked threads generate and process lists locally, and corresponds to the strong allocation locality identified in Figure 9a.

Figure 13 – Figure 15 shows the GC thread locality heat maps for the data parallel sumeuler. Figure 13 is for static objects (CAFs) and again shows no locality, only 13.2% of static objects are processed locally. Figure 14 is for Generation 1 objects and shows good locality, with 61.7% of objects processed locally, and no access to many remote regions (grey entries). Figure 15 is for Generation 2 objects and shows good locality (57.7%). Remote GC accesses likely arise in the data parallel implementation as fragments of the list may reside in a different region from the HEC that uses it. Table 5 shows

Figure 10. DNC sumeuler GC thread locality: Static objects (CAFs)

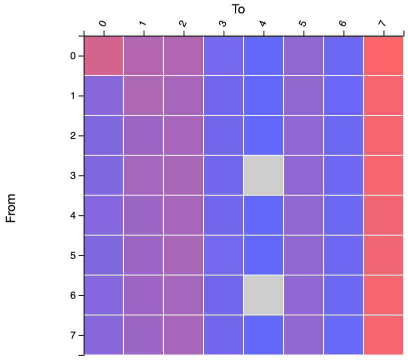


Figure 11. DNC sumeuler GC thread locality: Generation 1 objects

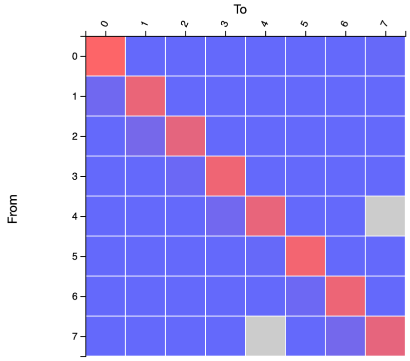
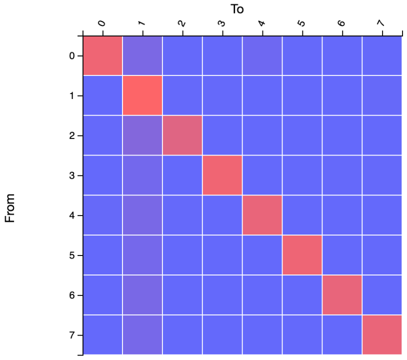


Figure 12. DNC sumeuler GC thread locality: Generation 2 objects



[7] https://github.com/ruairidhm98/ghc

that, compared with DNC the GC visits far fewer Gen 1 objects (0.89M vs 4.60M), and fewer Gen 2 objects (1.87M vs 3.63M).

The GC thread locality heat maps for prsa are similar to those for the sumeuler benchmarks. There is little locality for static objects (CAFs): only 13.3% of static objects are processed locally. Both Generation 1 and Generation 2 have good locality, with 61.8% and 73.2% of objects processed locally respectively. So both short and long lived objects are processed locally.

7 Conclusion

NUMA architectures are increasingly important as general purpose multicore architectures. Since release 8.2 GHC provides some simple adaption for NUMA: sparked threads allocate objects into the region where they execute. This paper is one of the few studies of NUMA usage in a functional language, and reports a systematic study of GHC Haskell on a typical NUMA server (8 regions, 64 cores).

We report **the systematic profiling of NUMA memory usage by GHC 8.4 parallel Haskell programs using existing compiler & OS tools.** The study uses the GHC heap profiler and the OS-level *numaprof* tool to profile eight GHC *nofib* benchmarks (Table 3), and three are presented in detail: two versions of the data intensive sumeuler, and the compute intensive prsa encryption engine. We record and analyse two standard metrics: (1) memory *residency*; and (2) *total allocation*, and the profiles reveal significant differences in the memory management required by the benchmarks.

We also record and analyse two NUMA-specific metrics. (3) *OS thread access locality* reveals some locality of access to NUMA regions, with local access varying from 30% for data intensive sumeuler to 23% for compute intensive prsa. Moreover in each benchmark there is a single *hot* region with a high percentage of accesses. From existing profiles we can only speculate as to what is causing the *hot* region. (4) *OS thread NUMA region counts* from *numaprof* provide *the first ever quantification of the effectiveness of GHC NUMA adaption.* That is, the policy of local allocation is effective in increasing local access from a notional 12.5% to around 30% for data intensive benchmarks, and to around 23% for a compute-intensive benchmark (Section 4.3).

We propose a new metric: **NUMA access rate** that allows us to compare the load placed on the memory system by different programs. We show that the data parallel sumeuler generates twice the load of the DNC version, and that prsa generates the highest load. The study exposes key information about GHC Haskell NUMA usage not available from current profilers, and motivates the development of new NUMA profiling (Section 4).

We **design, implement, and demonstrate an extension to *numaprof* to record *OS thread allocation locality*.** The openly available extension records the NUMA region of the calling thread and the bytes allocated into each region. The output is integrated into the *numaprof* UI to show a heat map of allocations between regions. The profiles reveal 100% allocation locality for sparked threads, and that the GHC main thread allocates a significant amount of memory into a single region. Hence the profiles provide new information to confirm speculations based on access profiles (Section 5).

We **design, implement, and demonstrate extensions to profile NUMA usage during GHC garbage collection**. We extend GHC's global GC statistics to record *GC count per region* and *GC count per generation in each region.* These counts reveal that in the constantly allocating DNC sumeuler Generation 1 and 2 collections are evenly spread between the regions. For the data parallel sumeuler there are far more Generation 2 collections compared to the DNC sumeuler as most allocations are done early and long-lived objects are promoted into Generation 2. For the compute intensive prsa there are fewer Generation 1 collections representing the low allocation, but many Generation 2 collections showing that the data that is allocated is long-lived. We also extend GHC's GC to record *GC thread locality*, again using the *numaprof* UI to produce a region × region heat map. For all benchmarks the heat maps reveal no locality for static objects: the GC threads access all regions to collect GHC Constant Applicative Forms (CAFs), but excellent locality for Generation 1 and Generation 2 collections, between 60% and 75% local accesses (Section 6).

Future Work. There are two main avenues for future work. The first is to provide additional NUMA profiling for GHC parallel Haskell. Integrating NUMA latencies into the *GC thread locality* heat-maps would provide more information on the NUMA access costs of GC. The per region GC count profiler could record GC cycle runtimes per region to identify regions with high GC overheads. It would also be desirable to record the memory bandwidth consumption per region, as in *Intel VTune* (no such tool for Togian's AMD Opteron processors). Optimising the profiler tools to reduce runtime overheads is also desirable, *e.g.* as outlined in Section 6.

The NUMA profiles reveal areas where the GHC RTS could better exploit NUMA, and the second line of future work is to explore these possibilities. For example *GC thread locality* identifies opportunities to move remote data closer to the thread(s) that access it during garbage collection. It may also be possible to adapt the unpinned GHC main thread to move CAFs to the region with the most accesses, as identified by the *OS thread access locality* profiles.

Acknowlegements This work was supported by EPSRC grants MaRIONet (EP/P006434) and STARDUST (EP/T014628).

Figure 13. Data parallel sumeuler GC thread locality: Static objects (CAFs)

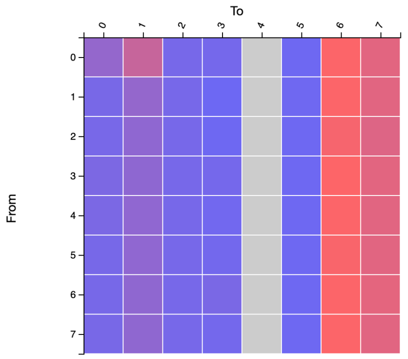


Figure 14. Data parallel sumeuler GC thread locality: Generation 1 objects

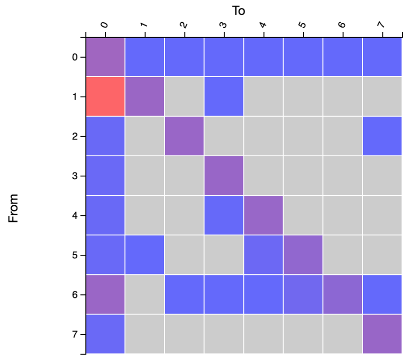
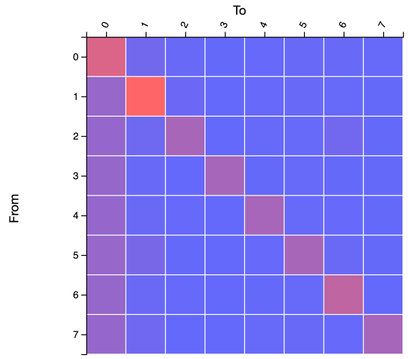


Figure 15. Data parallel sumeuler GC thread locality: Generation 2 objects



References

[1] M. S. Aljabri *et al.* Balancing shared and distributed heaps on NUMA architectures. In *TFP14: Intl Symp on Trends in Functional Programming*, LNCS 8843, pages 1–17. Springer, 2014.

[2] K. Alnowaiser. A study of connected object locality in NUMA heaps. In *MSPC14: Workshop on Memory Systems Performance and Correctness*, pages 1:1–1:9. ACM, 2014.

[3] K. Alnowaiser and J. Singer. Topology-aware parallelism for NUMA copying collectors. In *LCPC15: Workshop on Languages and Compilers for Parallel Computing*, LNCS 9519, pages 191–205. Springer, 2015.

[4] F. Broquedis *et al.* ForestGOMP: An Efficient OpenMP Environment for NUMA Architectures. *Intl. J. Parallel Program.*, 38(5-6):418–439, 2010.

[5] P. J. Drongowski. Instruction-based sampling: A new performance analysis technique for AMD family 10h processors. Technical report, Advanced Micro Devices, 2007.

[6] F. Gaud *et al.* Challenges of Memory Management on Modern NUMA Systems. *ACM Queue*, 13(8), Dec. 2015.

[7] L. Gidra *et al.* NumaGiC: a Garbage Collector for Big Data on Big NUMA Machines. In *ASPLOS15: Intl Conf on Architectural Support for Programming Langs and O.S.*, pages 661–673. ACM, 2015.

[8] S. Kell. Some were meant for C: the endurance of an unmanageable language. In *Onward! Intl Symp on New Ideas, New Paradigms, and Reflections on Programming and Software*, pages 229–245. ACM, 2017.

[9] R. La Rowe *et al.* Evaluation of NUMA Memory Management Through Modeling and Measurements. *IEEE Trans. Parallel Distributed Syst.*, 3(6):686–701, 1992.

[10] R. Lachaize *et al.* Memprof: A memory profiler for NUMA multicore systems. In *USENIX*, pages 53–64, 2012.

[11] C. Luk *et al.* Pin: building customized program analysis tools with dynamic instrumentation. In *PLDI05: Programming Languages Design and Implementation*, pages 190–200. ACM, 2005.

[12] Z. Majo and T. R. Gross. Memory management in NUMA multicore systems: trapped between cache contention and interconnect overhead. In *ISMM11: Intl Symp on Memory Mgmt*, pages 11–20. ACM, 2011.

[13] S. Marlow *et al.* Seq no more: better strategies for parallel Haskell. In *Haskell’10: Intl Symposium on Haskell*, pages 91–102. ACM, 2010.

[14] O. Papadakis *et al.* You can’t hide you can’t run: a performance assessment of managed applications on a NUMA machine. In *MPLR ’20: 17th International Conference on MPLR*, pages 80–88. ACM, 2020.

[15] W. Partain. The nofib benchmark suite of Haskell programs. In *GlaFP’92: Glasgow Functional Programming Workshop*, pages 195–202. Springer, 1992.

[16] I. Sánchez Barrera *et al.* Reducing data movement on large shared memory systems by exploiting computation dependencies. In *ICS’18: Intl Conf on Supercomputing*, pages 207–217, Beijing, China, 2018. ACM.

[17] J. Shapiro. Programming language challenges in systems codes: why systems programmers still use C, and what to do about it. In *ASPLOS’06: Architectural Support for Programming Languages and Operating System*, page 9. ACM, Oct 2006.

[18] L. Tang *et al.* Optimizing Google’s warehouse scale computers: The NUMA experience. In *HPCA’13: Intl Symp on High Performance Computer Architecture*, pages 188–197. IEEE Computer Society, 2013.

[19] D. Terpstra *et al.* Collecting performance data with PAPI-C. In *IWPTHPC’09: Tools for High Performance Computing*, pages 157–173. Springer, 2009.

[20] S. Valat and O. Bouizi. Numaprof, A NUMA memory profiler. In *EuroPar’18*, LNCS 11339, pages 159–170. Springer, 2018.

[21] S. Valat *et al.* MALT: a malloc tracker. In *SEPS’17: Intl Workshop on Software Engineering for Parallel Systems*, pages 1–10. ACM, Oct 2017.

[22] G. Voron *et al.* An Interface to Implement NUMA Policies in the Xen Hypervisor. In *European Conference on Computer Systems (EuroSys’17)*, pages 453–467. ACM, 2017.

[23] X. Zhao *et al.* NumaPerf: predictive and full NUMA profiling. *CoRR*, abs/2102.05204, 2021.