Balancing Shared and Distributed Heaps on NUMA Architectures

Malak Aljabri[1](✉), Hans-Wolfgang Loidl[2], and Phil Trinder[1]

[1] School of Computing Science, University of Glasgow, Glasgow G12 8QQ, Scotland, UK
m.aljabri.1@research.gla.ac.uk
[2] School of Mathematical and Computer Sciences, Heriot-Watt University, Riccarton, Edinburgh EH14 4AS, Scotland, UK

Abstract. Due to the varying latencies between memory banks, efficient shared memory access is challenging on modern NUMA architectures. This has a major impact on the shared memory performance of parallel programs, particularly those written in languages with automatic memory management.

This paper presents a performance evaluation of distributed and shared heap implementations of parallel Haskell on a state-of-the-art physical shared memory NUMA machine. The evaluation exposes bottlenecks in the shared-memory management, which results in limits to scalability beyond 25 out of the 48 cores.

We demonstrate that a hybrid system, GUMSMP, that combines both distributed and shared heap abstractions consistently outperforms the shared memory GHC implementation on seven benchmarks by a factor of 3.3 on average. Specifically, we show that the best results are obtained when sharing memory only within a single NUMA region, and using distributed memory system abstractions across the regions.

1 Introduction

Current high-end servers offer 48 or 64 cores with a NUMA (non-uniform memory access) architecture that supports shared memory access across the address space. On such architectures, reduced synchronisation costs are bought at the price of memory latencies, which vary by a factor of up to 3, depending on the NUMA region in which the memory bank is located (Sec 2.2). Even more problematic, for those applications that require frequent memory access, the memory bus can become a major bottleneck, degrading access times far below the values measured on an idle machine. These architectures pose a challenge to parallel languages, especially in cases where they make very dynamic use of memory.

We study the impact of state-of-the-art NUMA architectures on the parallel performance of languages with automated memory management. We explore a range of systems from purely shared-memory, hybrid shared/distributed memory to purely distributed memory. The vehicle for our study is a suite of implementations of the Glasgow parallel Haskell (GpH) extension of Haskell. The underlying

© Springer International Publishing Switzerland 2015
J. Hage and J. McCarthy (Eds.): TFP 2014, LNCS 8843, pp. 1–17, 2015.
DOI: 10.1007/978-3-319-14675-1_1

compiled parallel graph reduction execution model induces both frequent and highly random memory access, aggravating the impact of the NUMA memory architecture. Hence GpH programs are excellent test cases for exploring the impact of NUMA memory management. Our results, however, are not restricted to Haskell: the issues we explore impact all languages with automated memory management on NUMA architectures.

We demonstrate that the scalability of the shared memory GpH implementation is limited by heap contention due to synchronisation and locking overheads of the stop-the-world, parallel garbage collector (Section 3.3). This limits the number of cores, that can be usefully exploited, to well below the 48 physical cores available on our AMD Opteron measurement platform. In contrast, our hybrid shared/distributed system can effectively exploit several distributed heaps on a physical shared memory machine, to reduce both memory contention and heap locking.

We quantify the impact of heap contention on state-of-the-art NUMA servers for memory intensive languages like GpH, and hence identify how parallel Haskell applications can best exploit emerging shared memory hardware architectures. This paper makes the following contributions:

- We investigate the scalability limits of the shared heap implementation of a memory intensive language (the GHC-SMP implementation of GpH) on a recent NUMA architecture (Section 3.2).
- We analyse the memory usage profile of the applications and find that a hybrid shared/distributed memory implementation of a memory intensive language (the GUMSMP implementation of GpH) exhibits significantly smaller GC overheads than the shared memory implementation GHC-SMP.
- For a range of shared and distributed heap configurations, the hybrid GUMSMP approach improves performance by a factor of 3.3 on average for 7 benchmarks (Section 3.3).
- We investigate how to optimise the number of cores per node of a distributed memory multicore cluster (Section 4).

Our measurements in Section 3.3 demonstrate a drop in runtime by up to a factor of 4.5 when using the hybrid GUMSMP, over the specialised shared memory GHC-SMP system. Moreover, we achieve the best results when using the shared memory system on a single NUMA region, while using the distributed memory system across the regions, effectively matching the number of heaps to the number of NUMA regions on the hardware platform. We conjecture that this latter configuration represents a sound decision for other languages with automated memory management, which will be similarly affected by the varying memory latencies.

2 Background

2.1 Parallel Haskell Implementations

This section provides a brief overview of the three existing parallel Haskell implementations: GHC-GUM, using distributed heaps with a virtual shared heap

abstraction to hide the distribution from the programmer; GHC-SMP, using a shared heap implementation; and GHC-GUMSMP, or just GUMSMP, representing a hybrid of both memory management models.

The Distributed Memory GHC-GUM Implementation: GHC-GUM (Graph Reduction for a Unified Machine Model) [19] is our research platform for distributed memory parallelism based on GHC, which represents a portable implementation of an abstract graph reduction machine, based on explicit message passing and implementing a virtual shared heap. It implements the Glasgow Parallel Haskell (GpH) extension. GHC-GUM was built as an extension to the runtime environment (RTE) of the Glasgow Haskell Compiler (GHC) [13]. Parallelism is introduced by the `par` primitive, indicating that the evaluation of an expression is potentially parallel, and exploited by reducing separate sub-graphs in parallel [15].

A key concept integrated into the GHC-GUM design is the virtual shared heap, as shown in Figure 1, where the graph representing the program to be evaluated in parallel is stored, and is implemented on top of a distributed memory model. Another key characteristic is the dynamic and adaptive management of both work and data. This enables the runtime environment to adjust the dynamic behaviour of an application to the hardware characteristics and to the dynamic behaviour of the program.

Memory Management: The parallel program is represented as a graph in a (flat) virtual shared memory and can be evaluated in parallel using the available processors. Each Processing Element (PE) has local memory integrated into the global distributed heap, and a two level addressing scheme; one for local addresses (LAs), and one for global addresses (GAs), which is used to reference values in the shared heap. GAs enable each PE to garbage collect locally, without the need to synchronise with other PEs.

A Global Address (GA) is a globally unique identifier for a closure, which is created as a result of sending work from one PE to another in response to a work-request message. After a thunk, representing work, is sent to the requesting PE, the original thunk is overwritten with a `FetchMe` closure, a global indirection, containing the global address of the new copy of the thunk at the destination. The purpose of overwriting a thunk with `FetchMe` is to indicate that it is being evaluated in another PE, and to indicate its new location, should the result be needed subsequently by the original PE. The GA consists of a locally unique identifier, the PE identifier of the destination and a weight, as discussed below.

A Global Indirection Table (GIT) is maintained within each PE to map global identifiers to the local address of the corresponding heap closure. The GIT acts as a source of roots for local garbage collection. This design enables each PE to garbage collect independently, provided that the GIT is adjusted after each garbage collection to reflect the new locations of the local heap closures.

Global addresses are garbage collected using standard distributed weighted reference counting algorithm [10]. When a GA is created, it has an initial weight

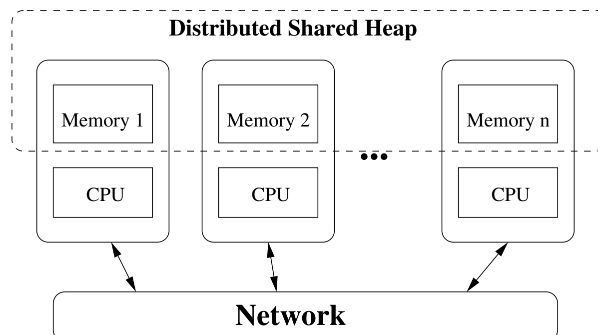

Fig. 1. Structure of a Virtual Shared Heap

that is split whenever the reference is shared. This mechanism aims to minimise the synchronisation needed among referrers to one closure. When a global object is locally garbage collected, the associated reference weight is returned to the owning PE.

A mapping of global to local addresses is required to ascertain whether a copy of a newly imported graph structure already exists on that PE, and to avoid duplication of data and work. If a newly imported graph structure does exist, the version of the graph, which has been evaluated less, will be subsumed by the more evaluated version. The details of this design are discussed in more detail in the virtual shared memory instance of PAEAN [6].

The Shared Memory GHC-SMP Implementation: GHC-SMP is an optimised shared memory implementation of GpH, integrated into the standard distribution of GHC [8,12]. It assumes physical shared memory and uses mutexes for synchronisation between local threads. GHC-SMP excels at the efficient handling of light weight threads. Millions of light weight threads are supported by the GHC runtime environment, which also supports concurrency. In order to achieve high thread management performance, the threads are multiplexed onto a handful of operating system threads, approximately one for each physical CPU. A thread is represented by a thread state object (TSO), and a heap allocated structure, which maintains the state of the Haskell thread including its stack. The structure of the TSO is the same as in GHC-GUM. A small set of operating system threads (worker threads, one worker thread per core) execute the Haskell threads. One Haskell Execution Context (HEC) is maintained for each core, owing to the fact that the worker thread may vary frequently.

Memory Management: The memory management is based on the concept of a block-structured heap [12,14,20]. The shared heap is divided into non-contiguous, fixed-sized blocks. A block allocator manages these blocks, which can be singly allocated, and linked together into lists to form an allocation area, to be provided to each HEC to allocate fresh objects. They can also be linked in contiguous

groups to allocate large objects with sizes greater than a block size. The operating system provides the block allocator with memory initially and when it has none remaining.

The garbage collector implemented in GHC-SMP is a generational, copying garbage collector based on dividing the shared heap into generations of fixed-size blocks. Generations are numbered from `0` to `n`, with `0` being the youngest. To collect generation `n`, all younger generations from `0` to `n` must be collected. A remembered set is maintained to keep track of all pointers referenced from mutable objects in the older generation to younger ones. The youngest generation, where new objects are allocated, is frequently garbage collected. Objects are promoted from generation `n` to the older generation `n+1`, which is collected less frequently, after they have survived a specific number of collections.

Each generation is collected using a copying collection, where the promotion of objects takes place by evacuating all reachable objects from the root pointers or remembered sets of older generations. Then, the scavenge phase operates on each evacuated object and in turn evacuates each pointer in the object.

This garbage collection is parallel and *stop-the-world* so it is initiated by a HEC with an exhausted allocation area, and takes place when all the HECs *synchronise* to start the garbage collection. For parallel copying GC, it is important to evacuate or scavenge each object using different processors. Each GC thread synchronises to get a private (*to-space*) allocation block. Local per-HEC remembered sets are maintained to avoid synchronisation costs and to improve data locality, as the TSOs that have been executed on a given core, with the data they refer to, are likely to be present in the core cache, and therefore traversed by the garbage collector on the same core.

Load balancing of the GC is achieved with work stealing queues. When the GC begins, each HEC already has a lot of data in its cache. Therefore, the GC thread takes blocks to scavenge from its own queue in preference to stealing, starting with blocks from the oldest generations. If no work is available in its own queue, then it will try to steal work from the queues of other HECs. This design improves the locality and reduces the contention of a single, global work queue, which was originally implemented in GHC-SMP. In fact, stealing work from the queues of other HECs in order to balance the load, is to be avoided with minor collections as it has a detrimental effect on locality [12].

Lock Contentions: During parallel garbage collection, synchronisation is required for the following parts:

1. One global lock in the block allocator to obtain a new block for a GC thread: Each GC thread needs a private block into which objects can be copied when they are evacuated. Contention to this lock is reduced by allocating multiple blocks at a time and by keeping the spare ones on a private partly-free-list associated with the thread. When a GC thread wants a fresh allocation block, it first searches in its partly-free-list to reduce synchronisation.
2. One lock per step in the large-object lists: Large objects with sizes greater than a block size are allocated into a block group of contiguous blocks.

A linked list of large objects is maintained for each step of each generation. During the garbage collection, those large objects are not copied, but are, instead, moved by re-linking them from one linked list to another, and therefore require a lock.

3. The per-object evacuation lock: To prevent multiple GC threads from copying the same object, an atomic instruction is required for synchronisation. This synchronisation represents the major source of overhead for the parallel copying GC with up to 30% of the GC time [14]. In improvements to the original design, this contention was reduced by relaxing the lock when copying immutable objects, resulting in a 7% improvement. Since the rate of actual collisions is very low, the space wasted by duplicate copying is negligible [12].

The Hybrid Shared/Distributed Memory GUMSMP Implementation: GUMSMP is our integration of GHC-SMP and GHC-GUM functionality in one system. It is designed to be multilevel, using different, tailored technologies on the small scale, physical shared memory level (multicores) and also on the large scale, distributed memory level (clusters). The design was built based on the successful technologies that already exist at both levels. In particular, it combines a mechanism of work stealing for passive load distribution, with an adaptive, dynamic mechanism for automatically distributing work and data on a cluster. Technically, this design was achieved by integrating the functionalities of the existing GHC-SMP and GHC-GUM implementations of the RTE for GHC. The main design objectives for GUMSMP and the implementation details can be found in [1].

2.2 NUMA Architectures

One of the main trends in hardware design is the use of a NUMA (Non-Uniform Memory Access) model for physical shared memory machines [9]. The design goal is to provide performance scalability for manycore machines with large main memory. In this model, the main memory is partitioned into several NUMA regions, each of which is associated with several cores. Access to the memory within one region is fast, while remote access must pass through an on-chip network to access a different memory bank, and is much slower. This performance asymmetry intensifies as when the number of cores in a single region increases, thus negatively affecting uniformity [17].

For manycore processors, the NUMA design of the memory sub-system requires awareness of the differences in latency by the system or the algorithm to avoid scaling issues. Both effective memory bandwidth and latency to different regions on the processor can be negatively impacted by problems with hardware [4].

Traditionally the term NUMA is mainly used to characterise the structure of the memory sub-system. However, in general, other resources, such as I/O, are also impacted by the asymmetry of NUMA architectures. This can result in substantial fluctuation in I/O performance relative to latency and bandwidth,

where remote I/O access generates a higher latency and usually a lower bandwidth for data transfer, as shown in [17]. This paper, however, mainly focuses on the asymmetry in memory latencies.

3 Performance on NUMA Architecture

The measurements are made on a 48-core NUMA machine, provided by four AMD Opteron-based processors, one per socket. Each processor contains two NUMA regions, and each region has six 2.8GHz cores. The total RAM is 512 GB, evenly distributed as 64 GB for each region. A 2 MB L2 cache is shared between every 2 cores in each region, and a 6 MB L3 cache is shared between all the 6 cores within the same region. The machine runs x86_64 Linux CentOS 6.5. Memory latencies[1] on this NUMA architecture vary by a factor of 2.2.

The RTE of the parallel Haskell implementations are based on GHC 6.12.2, using GCC 4.4.7, and PVM 3.4.5 for message passing. For GHC-SMP, the performance of GHC 7.6.3 was tested, delivering similar results. In our experiments, we choose 40 cores to evenly partition the machine into 2, 5, 6, and 8 regions.

3.1 Setup and Programs

We used the following benchmarks that exhibit a range of parallel patterns:

- `parfib` is a divide-and-conquer program, which computes for a given value, the Fibonacci number using a depth threshold.
- `coins` is a divide-and-conquer program, which computes the number of ways to pay a given value from a fixed set of coins.
- `sumEuler` is a data-parallel program, which computes the sum of the Euler totient function on the list interval.
- `worpitzky` is a divide-and-conquer program, which checks the Worpitzky property over Stirling numbers.
- `maze` is a nested data-parallel AI application for finding the path through a fixed maze using a parallelism threshold.
- `mandelbrot` is a data-parallel application for computing a Mandelbrot set over a given window size, and number of iterations.
- `blackscholes` is a data-parallel application, which represents implementation of the Black-Scholes algorithm for modelling financial contracts by providing a number of options, and granularity.

3.2 Scalability Limits

Table 1 compares runtimes using the GHC-SMP shared memory with the GHC-GUM distributed memory system. These numbers show a significant degradation in performance for the shared memory GHC-SMP system, beyond 15 to 25 cores, while the distributed memory GHC-GUM implementation continues

[1] Measured using `numactl -H`.

to scale. The program with the lowest heap allocation rate, `worpitzky`, scales best, achieving the lowest runtime in an GHC-SMP setting at 35 cores, but even this program has a lower performance on 40 cores (on an 48-core machine). Meanwhile, `coins` has the highest allocation rate; it represents the one with the lowest scalability, as performance starts to decrease after 15 cores.

Table 1. Runtimes for GHC-SMP and GUM with increasing core numbers (lowest RTs for GHC-SMP highlighted)

	Runtimes													
Cores	1		15		20		25		30		35		40	
Implementation	SMP	GUM	SMP	GUM	SMP	GUM	SMP	GUM	SMP	GUM	SMP	GUM	SMP	GUM
`parfib`	6004.2	6644.7	741.8	573.4	746.7	406.1	**666.7**	350.4	740.9	296.3	711.9	307.7	752.67	276.31
`coins`	5155.7	5690.7	**829.2**	485.0	857.5	432.9	834.8	384.3	940.4	340.4	1137.5	340.7	1095.1	318.3
`sumEuler`	1507.9	1552.0	199.9	102.8	197.9	94.2	**182.3**	77.9	194.3	81.9	226.1	81.5	222.0	79.0
`worpitzky`	1842.3	1818.3	217.3	173.1	204.9	135.9	187.0	116.5	185.2	111.5	**169.9**	105.4	178.6	108.8
`maze`	3181.9	3289.4	1472.5	675.8	1424.4	505.5	**1404.3**	467.9	1553.9	419.3	1650.9	403.2	1527.9	348.7
`mandelbrot`	4226.9	3772.6	1163.1	420.0	**631.5**	327.9	801.2	294.9	779.8	303.5	821.6	313.9	882.4	315.4
`blackscholes`	5133.1	5996.3	542.5	396.3	463.32	326.3	431.8	265.1	**406.9**	245.4	491.6	235.4	596.9	200.42

While GHC-GUM starts with higher execution times on 1 PE, it typically outperforms GHC-SMP from ca. 10–15 cores onwards. In consideration of this trend, the remainder of the paper is based on a study that assumes there is an intermediate point with even higher performance in the range of the extremes of shared heap GHC-SMP, and distributed heap GHC-GUM.

3.3 Benefits of a Distributed Heap

The GUMSMP implementation of parallel Haskell combines the heap models for both GHC-SMP and GHC-GUM. It provides parameters for selecting the number of cores to be used, inherited from GHC-SMP, and for selecting the number of PEs (independent instances of the Haskell runtime environment), inherited from GHC-GUM.

The figures and tables in this section explore a range of configurations, from a purely shared heap to purely distributed heaps, using the GUMSMP implementation and a total of 40 cores. The columns in Table 2 show configurations in the form PE/N, indicating that PE instances of the runtime system, each with its own heap, are spawned, with N cores used in each instance, all accessing the same shared heap. Our goal is to establish a balance between PE instances and per PE core numbers that achieve the best results for this set of test programs.

Our main results, the runtimes in Figure 2, Table 2 (lowest runtimes highlighted), and the speedups in Figure 3 show that for all programs a hybrid of distributed and shared heaps achieves the best performance. Typically, it is best to use up to 5 of the 40 physical cores, resulting in at least 8 separate PEs running simultaneously. For the more data intensive `mandelbrot` application (see Figure 6), we observed a further, but minor, improvement when using 5 cores. Notably, the improvement relative to the pure shared memory execution (GHC-SMP) is most pronounced for `maze` (a data intensive program) and `coins`

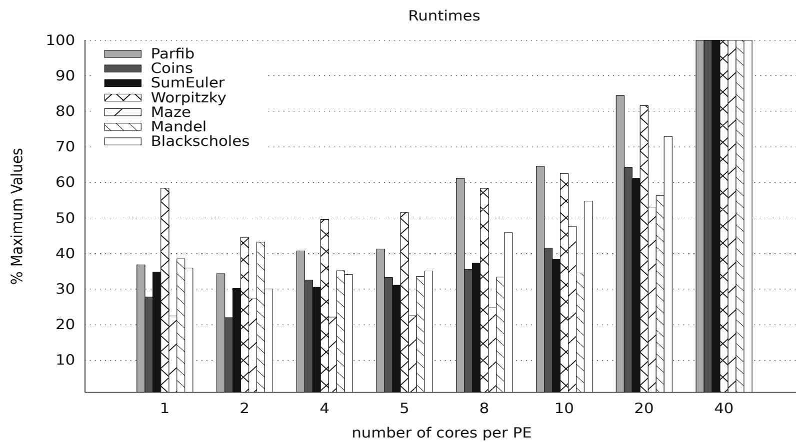

Fig. 2. Runtimes (normalised w.r.t. maximum runtime) for GUMSMP with increasing numbers of cores per PE. Note that in each case a total of 40 cores is used, and the difference is only in the number of cores that are used per PE.

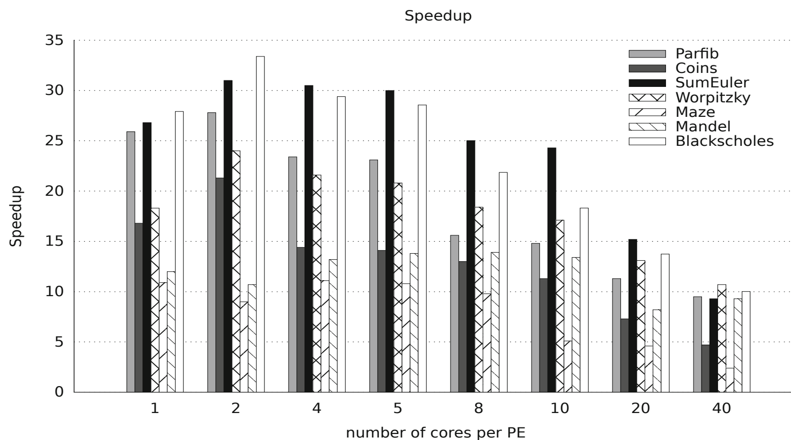

Fig. 3. Speedup for GUMSMP with increasing numbers of cores per PE (always using 40 cores in total)

(a divide-and-conquer program) with runtime improvements up to a factor of 4.5; whereas, improvements for other programs are between 2.2 and 3.3.

To quantify the garbage collection (GC) overhead, we measure the percentage of GC time relative to the total execution time in Figure 4. There is a strong correlation between this GC percentage, and the runtime, indicating a loss in performance for high core numbers, which is mainly due to memory management overheads. Part of this overhead is inherent to the parallel nature of the execution. All the programs typically generate a large number of threads;

Fig. 4. Average GC percentage (normalised w.r.t. maximum average GC percentage) for the hybrid GUMSMP system on 40 cores

especially in the case of the shared heap implementation. Each thread defines a set of live heap cells, which need to be retained following garbage collection. The large amount of live data, which shows up as significantly higher values of memory residency in Figure 6, translates into the need for a (currently NUMA-agnostic) garbage collection to perform more work, which represents one major source of overhead. The other major sources of overhead, which are harder to quantify, are the synchronisation to perform stop-the-world GC, and the per-object locking that is required to prevent multiple threads from duplicating mutable objects when copying, as discussed in Section 2.1.

It should be noted, that we always use the default minimum heap size of 0 for each PE; thus, there is no gain in the size of the initial heap when increasing the number of PEs. When increasing the minimum heap size, we observe a significant drop in runtime for GHC-SMP, as expected, because garbage collections are less

Table 2. Runtimes (in sec.) for the hybrid GUMSMP system on 40 cores (lowest RTs highlighted)

Configuration	**GUM**	**GUMSMP**							**SMP**	$\frac{SMP\ RT}{GUMSMP\ RT}$
PE/Cores	PE 40	40/1	20/2	10/4	8/5	5/8	4/10	2/20	N 40	
`parfib`	276.31	277.1	**258.5**	306.6	310.6	460.2	485.7	635.3	752.67	2.9
`coins`	318.3	304.4	**240.9**	356.8	364.5	388.7	455.3	702.9	1095.1	**4.5**
`sumEuler`	79.0	77.3	**66.9**	67.8	69.1	82.9	85.06	135.9	222.02	3.3
`worpitzky`	108.8	104.2	**79.6**	88.5	91.9	104.2	111.6	145.7	178.6	2.2
`maze`	348.7	343.1	375.6	**338.3**	344.0	378.3	728.36	810.7	1527.9	**4.5**
`mandelbrot`	315.4	332.0	372.5	303.3	**289.4**	**288.0**	297.9	485.1	882.4	3.0
`blackscholes`	200.42	214.4	**179.2**	203.6	209.5	273.7	326.8	435.4	596.9	3.3
Geom Means.										**3.3**

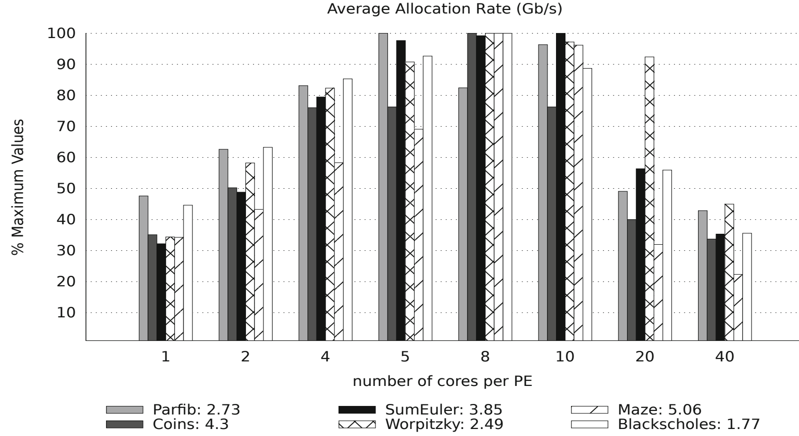

Fig. 5. Average allocation rate (normalised w.r.t. maximum average allocation rate) for the hybrid GUMSMP system on 40 cores (in GB/s)

frequent. However, the GHC-SMP runtimes are still substantially higher than the GUMSMP runtimes.

Figure 5 measures the amount of allocation per second with increasing core numbers per PE. We explain the serious degradation in allocation rate as an indirect consequence of the locking during GC, as discussed above.

While the synchronisation overhead for stop-the-world parallel GC is largely independent of the live data set, the per-object locking overhead increases with both higher core numbers and larger live data set. As a combination of both overheads, the garbage collection phase becomes the constraining factor in the allocation performance. This behaviour is indicated by the consistent drop in the allocation rate beyond ca. 8–10 cores per PE.

Notably, the memory residency shown in Figure 6 matches the profile of the GC percentage shown in Figure 4. This match underlines the fact that the majority of additional work done during the GC for high core numbers was required due to the size of the live data set in these configurations.

In summary, the combination of global synchronisation for GC and locking overheads in the parallel, copying GC account for a significant bottleneck in heavily allocating programs. This overhead, which becomes dominating with larger live data sets, is the main reason for the drop in performance observed in Table 1 and Figure 2. Notably, programs with a low allocation rate, such as `worpitzky`, exhibit the smallest runtime improvement over pure shared memory versions.

4 Performance on Multicore Clusters

GUMSMP was designed for clusters of multicore; whereby, the system can use a shared heap on one node and distributed heaps across nodes. In our previous

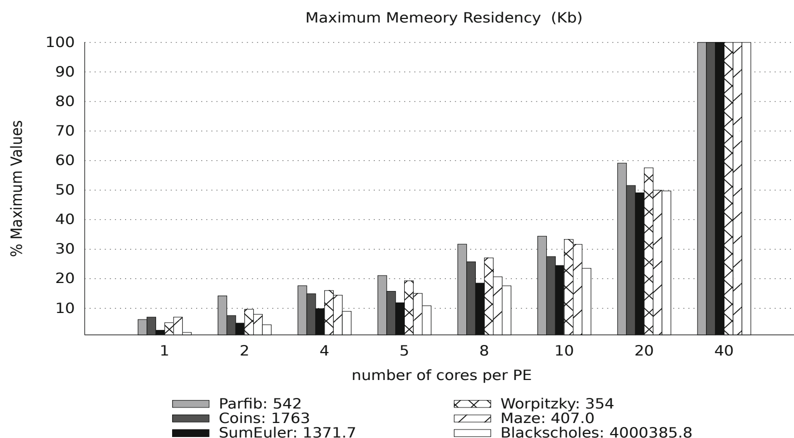

Fig. 6. Maximum memory residency (normalised w.r.t. maximum memory residency) for the hybrid GUMSMP system on 40 cores

results [1], we evaluated the performance of GUMSMP on a cluster of up to 100 cores with fixed number of cores per PE.

In this section, we systematically investigate how to optimise the number of cores per PE in a *distributed memory cluster*. In particular, we fix the total number of cores to be 84 and test the possible combinations of cores per PE and their effect on performance. This serves as guidance for our ongoing work to optimise the performance of GUMSMP on clusters of multicores. These measurements are made on a homogeneous Beowulf cluster of multicores, where each node is an 8-core CPU (2 quad-core Xeon E5506 2.13GHz, with 256kB L2 and 4MB shared L3 cache). All 32 nodes are connected via a non-specialised Gigabit ethernet connection. All machines are running Linux CentOS 6.4. The implementation of the GHC-SMP RTE is based on GHC 6.12.2, using GCC 4.4.7, and PVM 3.4.5 for message passing.

As indicated in Figure 7, using GUMSMP with 3 cores per PE instance consistently performs better with divide-and-conquer programs, with a speedup of up to 68 on 84 cores. Data-parallel programs still perform better using GUMSMP with larger cores per PE, and achieve the best performance using 4 cores per PE instance for `sumEuler` and 6 cores per PE instance for `mandelbrot` with a speedup of 65 and 21 respectively on 84 cores.

However, with increasing number of cores per PE instances, the performance of divide-and-conquer programs degrades, as a consequence of the shared memory management discussed in the previous section.

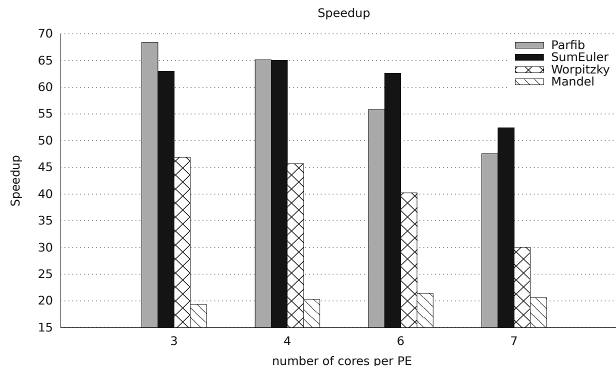

Fig. 7. Speedup for GUMSMP on the cluster of multicore with increasing numbers of cores per PE. Note that in each case a total of 84 cores is used.

5 Related Work

The impact of non-uniform memory latencies on parallel performance has recently been studied in several contexts. A comparative empirical study by Bergstrom [5], running low-level benchmarks in the modified C language STREAM, summarises that Intel Xeon architectures provide larger cross-processor bandwidth and suffer less from NUMA penalties compared to the widely used AMD Opteron architecture. This underlines the importance of NUMA on our measurements, using the latter architecture.

The baseline for our work is Marlow et al.'s [14] implementation of parallel, generational GC in GHC-SMP, which is the technology used in the main branch of the GHC runtime system (as discussed in Section 2.1). This work was extended to concurrent GC in [11]: in this implementation a GC thread runs concurrently with mutator threads, avoiding the need for a stop-the-world GC. The implementation features local heaps, parallel GC, in which each core has its own private heap, collected independently of others. There is also a shared heap, which is collected less frequently, using the parallel stop-the-world GC; thereby leading to less synchronisation. While this design is desirable and the new parallel GC achieves good performance improvements on up to 24 cores, scalability is lower than expected, and the implementation is significantly more complex than the current GC, which is of the parallel stop-the-world variety. Therefore, these modifications have not been merged into the mainline GHC.

Efficient automatic memory management on NUMA architectures is a challenge for aggressively allocating languages, like declarative ones. One notable system that tackles these challenges is the Manticore system for parallel ML, with the garbage collector implemented by Auhagen et al. [3]. It combines a split heap design with a three phase, semi-generational GC maximises locality and minimises global synchronisation. This was demonstrated to scale effectively

with good utilisation and improved performance over all available cores for a 48-core AMD Opteron, and a 32-core Intel Xeon machines.

A similar trend can be observed in modern Java implementations. The measurements by Gidra et al. [7] of several DaCapo benchmark programs implemented in OpenJDK7 mirror our observations made for (shared-memory) parallel Haskell programs: scalability is poor on a 48-core NUMA architecture, with a stop-the-world collector representing the main bottleneck. They provided more detailed measurements on the sources of overhead than we did, they identified the scanning and copying phases of remote objects, i.e. objects in remote NUMA regions, as the main overhead during GC, linking it to specific NUMA features.

ffect the performance. ects the scalability.

ffer) which is local to one GC thread to avoid locking for every in-copy object. ffer (TLAB), a fraction of young generation space allocated to every application thread for lock-free object allocation in the .However, there is no guarantee that the physical memory for the GCLAB in use by a GC thread comes from the local memory node. like remote scanning, lack of object aNffinity between GC thread and in-copy object also causes this.

With a similar interest to our paper, Alnowaiser [2] studied locality characteristics in two Java benchmarks is. This paper evaluated and analysed the locality characteristics of a rooted sub-graph for NUMA GC using two DaCapo and SPECjbb2005 benchmarks. While data locality is generally high, on average more than 80% of objects are co-located with the root, large, distributed graphs suffer from being exposed to load balancing techniques that diminish data locality. The author suggests modifications to the GC heuristic, using the root location as locality heuristic for GC, and ensuring that GC is structured to process the roots on the same memory node in one phase.

While the above papers mainly present observations on performance and scalability, several authors have developed concrete improvements inside the RTE. In particular, Terboven et al. [18] offers concrete recipes for the parallel programmer to enhance performance of OpenMP programs with task-level parallelism. These recipes are designed to improve data-locality under several different workloads, and are based on extensive measurements of different task-level OpenMP implementations, using a range of benchmark programs.

While the above paper achieves performance improvements through changes on program level, Yi Su et al. [16] developed NUMA-aware, thread placement algorithms inside an RTE for OpenMP, considering the critical path when addressing NUMA latencies. They used on-line profiling of information obtained from hardware counters to direct thread placement; thereby improving performance by minimising the critical path of OpenMP parallel regions. These algorithms have been evaluated using four NPB OpenMP applications, achieving between an 8% to 26% improvement over the default Linux thread placement algorithm.

6 Conclusions

We have investigated the impact of a NUMA memory model on the parallel performance of languages with automated memory management using 7 Glasgow

parallel Haskell benchmarks on a state-of-the-art platform. We show that beyond 10 NUMA cores it is beneficial to use distributed heaps, and specifically one heap per NUMA region. Hence better performance is obtained for all benchmark programs with the hybrid shared/distributed memory model provided by our GUMSMP implementation.

We report the following as the main findings.

- GUMSMP's performance, with a maximum of 5 cores per PE is consistently better than a pure GHC-SMP execution, by a factor of up to 4. This configuration amounts to using a single shared heap for each NUMA region.
- For large core numbers GC overheads in the shared-memory GHC-SMP increase drastically, primarily due to the larger live heap set.
- The allocation rate of GHC-SMP is typically much smaller than that for GUMSMP. We conjecture that this a combination of synchronisation overhead in the stop-the-world parallel GC and locking overhead incurred to prevent multiple GC threads from accidentally duplicating mutable objects during parallel copying.

We observe best performance when using one shared heap per NUMA region, which means in our measurements using 5 cores per PE in a configuration of 8 PEs, running on a hardware with 40 cores. These improvements occurred, despite the fact that the RTE is not NUMA-aware, by simply structuring the heap into several distributed heaps and relying on the operating system for the concrete mapping. Further improvements should be possible with a tighter integration of the RTE into the underlying operating system. It should also be noted that, graph reduction based execution models, such as the one used in these systems, incur frequent and unstructured memory access. Therefore, the relative impact of different memory latencies is likely to be higher in our systems, and so, this study can be seen as a stress test for modern RTEs in the presence of NUMA architectures, contributing to studies of NUMA performance of languages with highly dynamic memory usage, as outlined in the related work.

In future work, we plan to study ways to make the RTE NUMA-aware, initially by directly mapping an RTE heap to a particular NUMA region. In the longer term a more fine-grained mechanism would be desirable, where segments, or partitions, of the heap can be assigned to specific parts of the shared memory. This paper, together with the source code of the benchmarks, and the data set is available online at: http://www.macs.hw.ac.uk/~dsg/projects/gph/papers/abstracts/tfp14.html.

Acknowledgements. This work has been supported by the European Union grant IST-2011-287510 "RELEASE: A High-Level Paradigm for Reliable Large-scale Server Software", by the UK's EPSRC grant EP/G055181/1 "HPC-GAP: High Performance Computational Algebra and Discrete Mathematics", by Saudi Arabian Ministry of Higher Education, and by Umm Al-Qura University.

References

1. Aljabri, M., Loidl, H.-W., Trinder, P.W.: The design and implementation of GUMSMP: a multilevel parallel haskell implementation. In: Proceedings of the 25th ACM SIGPLAN Symposium on Implementation and Application of Functional Languages, IFL 2013. ACM, Nijmegen (2013). http://dx.doi.org/10.1145/2620678.2620682
2. Alnowaiser, K.: A study of connected object locality in numa heaps. In: Proceedings of the 2014 ACM SIGPLAN Workshop on Memory Systems Performance and Correctness, MSPC 2014, pp. 1:1–1:9. ACM, New York (2014). http://doi.acm.org/10.1145/2618128.2618132
3. Auhagen, S., Bergstrom, L., Fluet, M., Reppy, J.: Garbage collection for multicore NUMA machines. In: Proceedings of the 2011 ACM SIGPLAN Workshop on Memory Systems Performance and Correctness, MSPC 2011, pp. 51–57. ACM, New York (2011). http://doi.acm.org/10.1145/1988915.1988929
4. Benner, R., Echeverria, V.T.E., Onunkwo, U., Patel, J., Zage, D.: Harnessing manycore processors for scalable, highly efficient, and adaptable firewall solutions. In: 2013 International Conference on Computing, Networking and Communications (ICNC), pp. 637–641, January 2013. http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=6504161&isnumber=6504039
5. Bergstrom, L.: Measuring numa effects with the stream benchmark. CoRR, abs/1103.3225 (2011). http://dblp.uni-trier.de/db/journals/corr/corr1103.html#abs-1103-3225
6. Berthold, J., Loidl, H.-W., Hammond, K.: PAEAN: Portable Runtime Support for Physically-Shared-Nothing Architectures in Parallel Haskell Dialects. Journal of Functional Programming (2015). To appear in Special Issue on Runtime-environments
7. Gidra, L., Thomas, G., Sopena, J., Shapiro, M.: Assessing the scalability of garbage collectors on many cores. In: Proceedings of the 6th Workshop on Programming Languages and Operating Systems, PLOS 2011, pp. 7:1–7:5. ACM, New York (2011). http://doi.acm.org/10.1145/2039239.2039249
8. Jones Jr., D., Marlow, S., Singh, S.: Parallel performance tuning for haskell. In: Proceedings of the 2nd ACM SIGPLAN Symposium on Haskell, Haskell 2009, pp. 81–92. ACM, New York (2009). http://doi.acm.org/10.1145/1596638.1596649
9. Lameter, C.: NUMA (Non-Uniform Memory Access): An Overview. Queue **11**(7), 40:40–40:51 (2013). http://doi.acm.org/10.1145/2508834.2513149
10. Lester, D.: An efficient distributed garbage collection algorithm. In: Odijk, E., Rem, M., Syre, J.-C. (eds.) PARLE 1989. LNCS, vol. 365, pp. 207–223. Springer, Heidelberg (1989). http://dx.doi.org/10.1007/3540512845_41
11. Marlow, S., Peyton Jones, S.L.: Multicore garbage collection with local heaps. In: Proceedings of the International Symposium on Memory Management, ISMM 2011, pp. 21–32. ACM, New York (2011). http://doi.acm.org/10.1145/1993478.1993482
12. Marlow, S., Peyton Jones, S.L., Singh, S.: Runtime support for multicore haskell. In: Proceedings of the 14th ACM SIGPLAN International Conference on Functional Programming, ICFP 2009, pp. 65–78. ACM, New York (2009). http://doi.acm.org/10.1145/1596550.1596563
13. Marlow, S., Peyton Jones, S.L.: The Glasgow Haskell Compiler. In: The Architecture of Open Source Applications, vol. 2. lulu.com (2012). http://www.aosabook.org/en/ghc.html

14. Marlow, S., Harris, T., James, R.P., Peyton Jones, S.L.: Parallel generational-copying garbage collection with a block-structured heap. In: Proceedings of the 7th International Symposium on Memory Management, ISMM 2008, pp. 11–20. ACM, New York (2008). http://doi.acm.org/10.1145/1375634.1375637
15. Peyton Jones, S.L.: Parallel Implementations of Functional Programming Languages. Comput. J. **32**, 175–186 (1989). http://portal.acm.org/citation.cfm?id=63410.63418
16. Su, C., Li, D., Nikolopoulos, D.S., Grove, M., Cameron, K., de Supinski, B.R.: Critical Path-based Thread Placement for NUMA Systems. SIGMETRICS Perform. Eval. Rev. **40**(2), 106–112 (2012). http://doi.acm.org/10.1145/2381056.2381079
17. Tan, L., Yufei, R., Dantong, Y., Shudong, J., Robertazzi, T.: Characterization of input/output bandwidth performance models in NUMA architecture for data intensive applications. In: 2013 42nd International Conference on Parallel Processing (ICPP), pp. 369–378, October 2013. http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=6687370&isnumber=6687321
18. Terboven, C., Schmidl, D., Cramer, T., an Mey, D.: Assessing OpenMP tasking implementations on NUMA architectures. In: Chapman, B.M., Müller, M.S., Massaioli, F., Rorro, M. (eds.) IWOMP 2012. LNCS, vol. 7312, pp. 182–195. Springer, Heidelberg (2012). http://link.springer.com/chapter/10.1007%2F978-3-642-30961-8_14
19. Trinder, P., Hammond, K., Mattson Jr., J.S., Partridge, A.S., Peyton Jones, S.L.: GUM: a portable parallel implementation of haskell. In: Programming Languages Design and Implementation, PLDI 1996, Philadelphia, PA, USA, pp. 79–88, May 1996. http://dx.doi.org/10.1145/231379.231392
20. Yang, E.Z.: The GHC Runtime System, July 2013. http://ezyang.com/jfp-ghc-rts-draft.pdf