NUMA Optimizations for Algorithmic Skeletons

Citation for published version:
Metzger, P, Cole, M & Fensch, C 2018, NUMA Optimizations for Algorithmic Skeletons. in 24th International European Conference on Parallel and Distributed Computing. Springer, Cham, Turin, Italy, pp. 590-602, 24th International European Conference on Parallel and Distributed Computing, Torino, Italy, 27/08/18. DOI: 10.1007/978-3-319-96983-1_42

Digital Object Identifier (DOI):
10.1007/978-3-319-96983-1_42

Link:
Link to publication record in Edinburgh Research Explorer

Document Version:
Peer reviewed version

Published In:
24th International European Conference on Parallel and Distributed Computing

NUMA Optimizations for Algorithmic Skeletons

Paul Metzger[1], Murray Cole[1], and Christian Fensch[2]

[1] University of Edinburgh - School of Informatics - Edinburgh, EH8 9AB, UK
{paul.metzger, m.cole}@inf.ed.ac.uk
[2] Heriot-Watt University - MACS - Edinburgh, EH14 4AS, UK c.fensch@hw.ac.uk

Abstract. To address NUMA performance anomalies, programmers often resort to application specific optimizations that are not transferable to other programs, or to generic optimizations that do not perform well in all cases. Skeleton based programming models allow NUMA optimizations to be abstracted on a pattern-by-pattern basis, freeing programmers from this complexity. As a case study, we investigate computations that can be implemented with stencil skeletons. We present an analysis of the behavior of a range of simple and complex stencil programs from the NAS and Rodinia benchmark suites, under state-of-the-art NUMA aware page placement (PP) schemes. We show that even though an application (or skeleton) may have implemented the correct, intuitive scheduling of data and work to threads, the resulting performance can be disrupted by an inappropriate PP scheme. In contrast, we show that a NUMA PP-aware stencil implementation scheme can achieve speed ups of up to 12x over a similar scheme which uses the Linux default PP, and that this works across a set of complex stencil applications. Furthermore, we show that a supposed PP performance optimization in the Linux kernel never improves and in some cases degrades stencil performance by up to 0.27x and should therefore be deactivated by stencil skeleton implementations. Finally, we show that further speed ups of up to 1.1x can be achieved by addressing a work imbalance issue caused by poor conventional understanding of NUMA PP.

1 Introduction

Modern systems have complex and non-uniform memory organizations to meet the high bandwidth requirements of increasing core counts. For example, multisocket systems feature multiple memory controllers that are spread over sockets (see Fig. 1). CPUs can access memory that is attached to a remote memory controller via interconnects. The downside of this is that memory accesses are non-uniform in terms of latency and bandwidth. Thus, great care must be taken when choosing the right location for a memory page at a given time during program execution. These complexities in memory systems of NUMA machines cause hard to predict performance anomalies [1,2].
NUMA aware program optimizations that address this problem are at the extremes of a spectrum. At one end are generic NUMA page placement (PP) schemes, such as First-Touch, Interleaved, and the Linux automatic NUMA Balancing feature which are known to exhibit pathological behavior in hard to predict situations [3,4]. At the other end of the spectrum are application specific memory optimizations such as shared variable privatization. However, transferring these to other applications is a labor-intensive

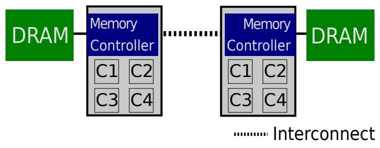


Fig. 1: Illustration of a NUMA system with two NUMA nodes.

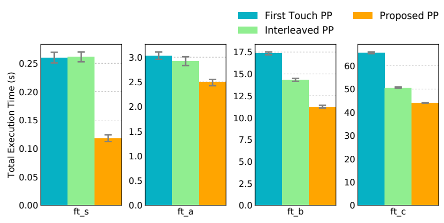
process. Skeleton-based programming systems [5–7] have the potential to support a compromise position: NUMA aware optimizations that are transparently applicable across the class of computations captured by each skeleton. In support of this hypothesis, we present a case study for *stencil* computations. We conduct an analysis of the behavior of stencil applications from the NAS-PB and Rodinia benchmark suites, comparing their performance under state-of-the-art NUMA placement schemes with performance under a stencil-skeleton-aware NUMA scheme, and its extension with a novel work distribution heuristic. We show that

- the stencil-skeleton-aware NUMA PP scheme has good applicability across a wide range of stencil computations, well beyond the simple Jacobi-style stencils which motivate it, offering speed-ups of up to 12x over state-of-the-art schemes.
- automatic NUMA Balancing, a generic optimization in the Linux kernel, is actively disruptive of stencil performance, diminishing performance by up to 0.27x, and so should be disabled by stencil skeletons.
- our novel work distribution approach further speeds up applications by 1.1x.

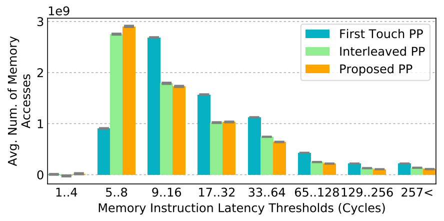
The remainder of this paper is structured as follows: Section 2 provides a motivating example that demonstrates the possible performance benefits of stencil aware PP. Section 3 introduces stencil computations and standard NUMA PP schemes. Section 4 motivates and describes our stencil aware PP and work distribution scheme, and provides an overview of the experimental program which informs and evaluates it. Section 5 describes the experimental set up and section 6 presents experimental results. Finally, sections 7 and 8 discuss related work and conclusions.

2 Motivating Example

As has previously been demonstrated for individual applications [8–11], this section provides an example which confirms that performance improvements over state of the art schemes can be achieved by adding application awareness to the page placement (PP) process. We use the NAS-PB Fourier Transformation (ft) benchmark as a case-study. Figure 2a shows execution times of ft with different PP schemes. Stencil Aware PP performs significantly better than the other schemes in all cases and the maximum speed up is 57%. We sampled the number of memory accesses that fall into set latency ranges to better understand the performance benefits (see Fig. 2b). The results indicate that Stencil Aware and Interleaved PP take pressure from interconnects and memory controllers compared to First-Touch PP as they use all interconnects and memory controllers evenly. Stencil Aware PP also minimizes the number of high latency remote memory accesses and, therefore, performs better than Interleaved PP.

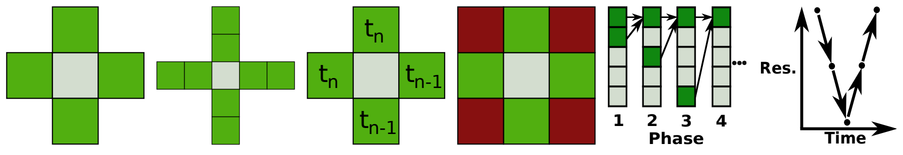


(a)



(b)

Fig. 2: (a) Execution times of the NAS-PB ft benchmark with different page placement (PP) schemes. The letters *s, a, b, c* indicate the standard problem set sizes in ascending order. (b) Access latency histogram of ft with the largest input data set.



(a) (b) (c) (d) (e) (f)

Fig. 3: Jacobi stencils (a + b), a Gauss-Seidel stencil (c), a stencil with a dynamic neighborhood (d), a butterfly divide-and-conquer stencil (e) and the multigrid method (f).

3 Background

3.1 Stencil Computations

Stencil computations update elements in a buffer based on the values in the elements' neighborhoods. Figure 3a illustrates this for a single element (grey) and its neigborhood (green). Updates are performed in a single sweep or multiple iterations. The remainder of this subsection discusses different types of stencil computations.
Jacobi stencil computations are conceptually the simplest stencils as their neighbourhood and input grid have a fixed size and shape. Shape and dimensionality of the neighborhood can vary across Jacobi stencils (see, for example, Fig. 3a and 3b). *Gauss-Seidel* stencils use values from the current and the previous iteration. In Fig. 3c elements at the top and the left-hand side are from the current and the elements at the bottom and right-hand side are from the previous iteration. Some stencils have a *variable* neighborhood that changes depending on the input data. The stencil in Fig. 3d uses either the green or the green and the dark red elements as input. The *red black* method arranges the elements in the input buffer like a checker board. Black elements are updated based on values of neighboring red elements and vice versa. The *Butterfly divide-and-conquer* method works in phases and changes the size of the stencil in each phase. Fig. 3e illustrates this based on the computation of one element. Arrows indicate which elements of

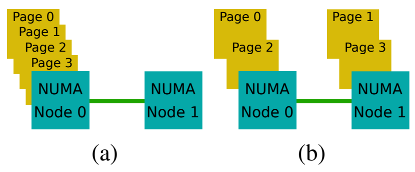


Fig. 4: First-Touch (a) and Interleaved (b) page placement.

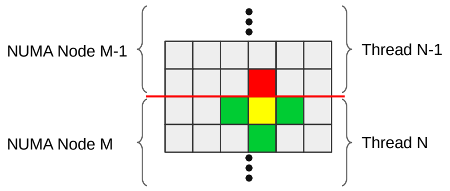


Fig. 5: Illustration of parallelization, collocation and remote memory access (red) when stencil aware PP is used. Elements above the red line are placed on NUMA node M-1 and elements below are placed on node M.

the input buffer are read in each phase. The *multigrid method* changes the resolution of the in- and output data dynamically (see Fig. 3f).

3.2 Page Placement Schemes

This section presents state of the art PP schemes. *First-Touch Page Placement* allocates pages on the same NUMA node as cores that first access them and is the default scheme of Linux. Figure 4a illustrates this PP policy. All pages are placed on node zero if the thread that runs on this node accesses them first. First-Touch PP optimises for data locality if pages are mostly accessed by threads that access them first. *Interleaved Page Placement* places pages on NUMA nodes in a round robin fashion and can be used as an alternative to First-Touch PP (see Fig. 4b). This scheme distributes memory access equally across memory controllers and interconnects but fails to optimise for data locality. *Automatic NUMA Balancing* migrates pages and threads across NUMA nodes, informed by run-time memory access statistics, to increase data locality. This is known to cause page thrashing and an extension called *Pseudo-Interleaving* has been proposed to address this [3]. Automatic NUMA Balancing is activated by default on Linux systems (i.e. in addition to First-Touch).

4 Stencil Aware Page Placement and Work Distribution for NUMA Systems

This section describes our stencil aware page placement (PP) and work distribution scheme and provides an overview of the experimental program which informs and evaluates it. We first describe a basic stencil aware NUMA PP scheme, as motivated in Sec. 2 and explain how this may be vulnerable to disruption by LinuxNUMA, a phenomenon which we will evaluate in Sec. 6. We then explain why the basic stencil aware PP scheme may experience performance degradation due to uneven distribution of remote accesses, and propose a novel work distribution technique which addresses this. The new PP and work distribution scheme are evaluated in Sec. 6.

A Basic Stencil Aware Page Placement Scheme Motivated by previously reported ad-hoc PP experiments, this scheme places pages on NUMA nodes that access them most frequently to improve data locality prior to a computation. Figure 5 illustrates this with a simple 2D Jacobi stencil. Stencil aware PP collocates thread N with the Nth memory block on NUMA node M, and so on. Note that in doing so we are going beyond conventional stencil-skeleton wisdom of simply associating threads with specific data partitions (and hence work), in order to ensure that this allocation is also respected by the underlying PP scheme. We also investigate whether this can be achieved for more involved types of stencil computations than simple Jacobi stencils (see Sec. 3.1).

Performance Degradation through automatic NUMA Balancing NUMA Balancing is known to cause page thrashing if multiple NUMA nodes access the same pages in an alternating fashion [4]. NUMA Balancing then migrates pages back and forth between these nodes. The stencil access pattern causes some pages to be shared between two NUMA nodes in each iteration of the stencil computation. In our experiments we investigate whether this effect degrades performance predictably for stencil computations.

Bad Work Distribution and our NUMA Aware Scheme The intuitive work distribution scheme for stencils allocates an equal share of grid points to each thread. However, this fails to consider the potential for unequal NUMA memory accesses to impact upon the time it takes to complete the corresponding work. Fig. 5 illustrates this for one element with a simple 2D Jacobi stencil. Meanwhile, some threads are not penalized by remote memory accesses and so complete their iteration sooner. These threads must wait on a barrier after each iteration, potentially creating a significant imbalance in waiting time and signifying a wasted resource. Our experiments investigate the extent to which this phenomenon occurs, and motivate the improved scheme described in the next section.
We propose and evaluate a novel work distribution scheme which aims to reduce the idle waiting time of threads that do not access remote memory (see previous section). This work distribution reflects the different access latencies in NUMA systems. Threads that are penalized by high latency remote memory accesses are assigned smaller chunks of input data than threads that access only local memory. Our experiments evaluate the impact of this new scheme.

5 Experimental Setup

To reduce the complexity of our experiments, we did not use a skeleton library but implemented the Stencil Aware PP and work distribution schemes, which could be implemented by a skeleton library, by hand. To enact the basic PP policy on top of the default OS First-Touch policy we introduce OpenMP code that creates and fills the stencil buffers with initial values. This parallel code imitates the memory access patterns of subsequent stencil computations, and so places pages on NUMA nodes that subsequently access them. In the srad benchmarks, buffers for precalculated indices are interleaved when Stencil Aware PP is used as the entire buffers are accessed by all threads. Stencil Aware PP can only work if the OS cannot migrate threads to another NUMA node, since otherwise, pages that these migrated threads access would then be

Table 1: Hardware details of the test machines.

Machine Name	Machine A	Machine B
CPU Model	Xeon L7555	Xeon E5-2697 v2
Sockets	4	2
Cores / Socket	8	12
LLC / Socket	18MB	30MB
Mem. Contr. / Socket	1	1
QPI Band. / Link	5.86GT/s	8GT/s
Hyperthr.	Deactivated	Deactivated
Prefetchers	active	active
Linux Kernel	4.4.36	3.10.0

Table 2: Benchmark application details. The letter s to c and numbers 64 to 8192 indicate standard input sizes.

App.	Stencil Type	Source	Memory Consumption
Srad v1	Jacobi	Rod	16MB
Srad v2	Jacobi	Rod	122MB
Hotspot	Jacobi	Rod	64: 10MB; 128: 12MB; 256: 11MB; 512: 15MB; 1024: 19MB; 2048: 54MG; 4096: 202MB; 8192: 800MB
MG	Multigrid	NPB	S: 10MB; A: 619MB; B: 620MB; C: 4,736MB
FT	Butterfly D&C	NPB	S: 21MB; A: 450MB; B: 1,760MB; C: 6,897MB

on a remote NUMA node and so we use thread pinning. Finally, the stencil iterations are implemented with an OpenMP parallel for region, using the static scheduling.
Table 1 lists details of the test systems. Machine A's kernel uses Pseudo-Interleaving (see Sec. 3.2) [3]. Benchmarks are taken from the Rodinia [12] and NPB-PB [13] suites (see Table 2) and are compiled with ICC 17.0.4 and the -O2 flag. The standard inputs of the benchmark applications are used except for the largest hotspot input due to very long execution time and the iterations that the Rodinia benchmarks perform are higher to reduce noise. Five samples are taken in the access latency experiment in Sec. 2 and at least ten samples are taken in each of the other experiments. Our timing experiments are reported with 95% confidence intervals. Our speed ups are reported as the ratio of the means of the relevant measurements. Spinning time and access latency related experiments were conducted with Intel V-Tune XE 2017.

6 Evaluation

6.1 Stencil Aware Page Placement

To assess the performance of *Stencil Aware* PP, we compare it in Fig. 6 against two state of the art PP schemes available on Linux: *First-Touch* and *Interleaved* PP. In most cases, Stencil Aware PP either matches or improves performance over state of the art PP schemes. A maximum speed up of 12x has been achieved with the hotspot benchmark

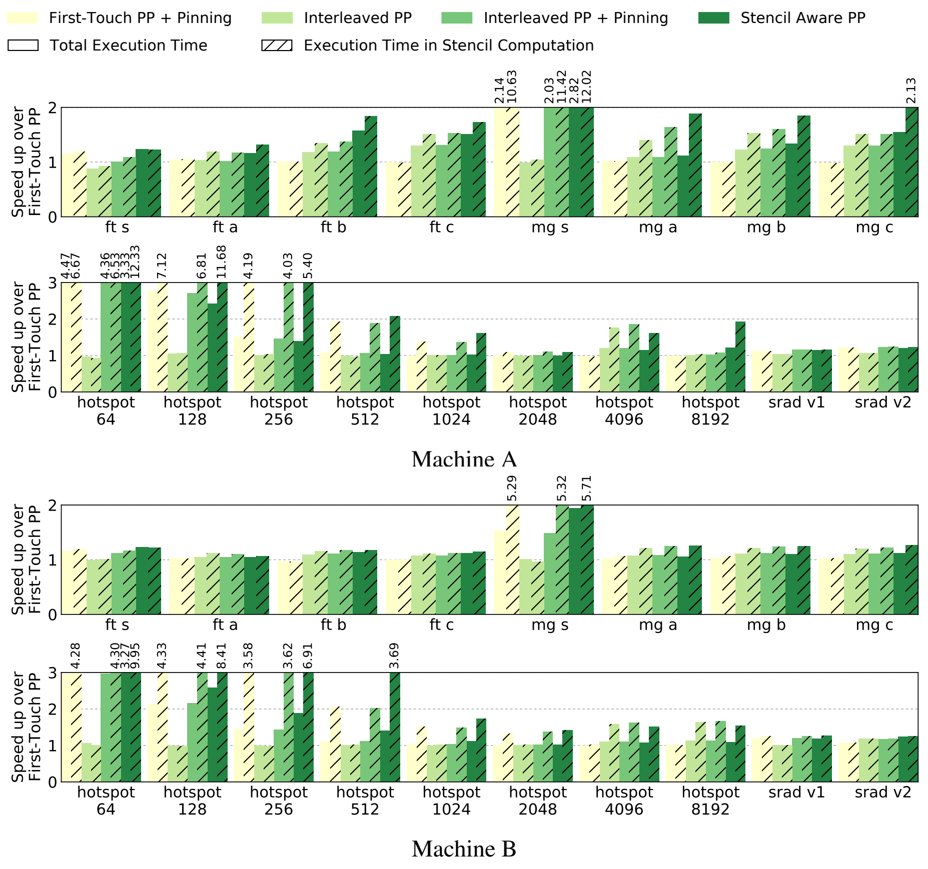


Fig. 6: Speed ups over standard implementations without pinning and the standard Linux First-Touch PP scheme with the total execution time and the execution time spent in the stencil iterations.

on machine A. Large speed ups can be observed for very small problem sizes i.e. mg with problem size s and hotspot with problem sizes 64 to 512. In only a small number of cases do the standard schemes perform slightly better. The standard PP schemes perform better than Stencil Aware PP when we measure the total execution time with hotspot 64 to 256 on machine A. However, Stencil Aware PP performs better in these cases when we measure the execution time spent in the stencil iterations. This indicates that Stencil Aware PP still improves the performance of the stencil computations but that the overhead of the page placement outweighs the performance benefits of Stencil Aware PP in these few instances. Lastly, results with small problem sizes and the standard PP schemes show that pinning significantly benefits performance. However, Stencil Aware PP still improves performance, and pinning has a very small, and in some cases no influence on performance for larger problem sizes. In summary, our re-

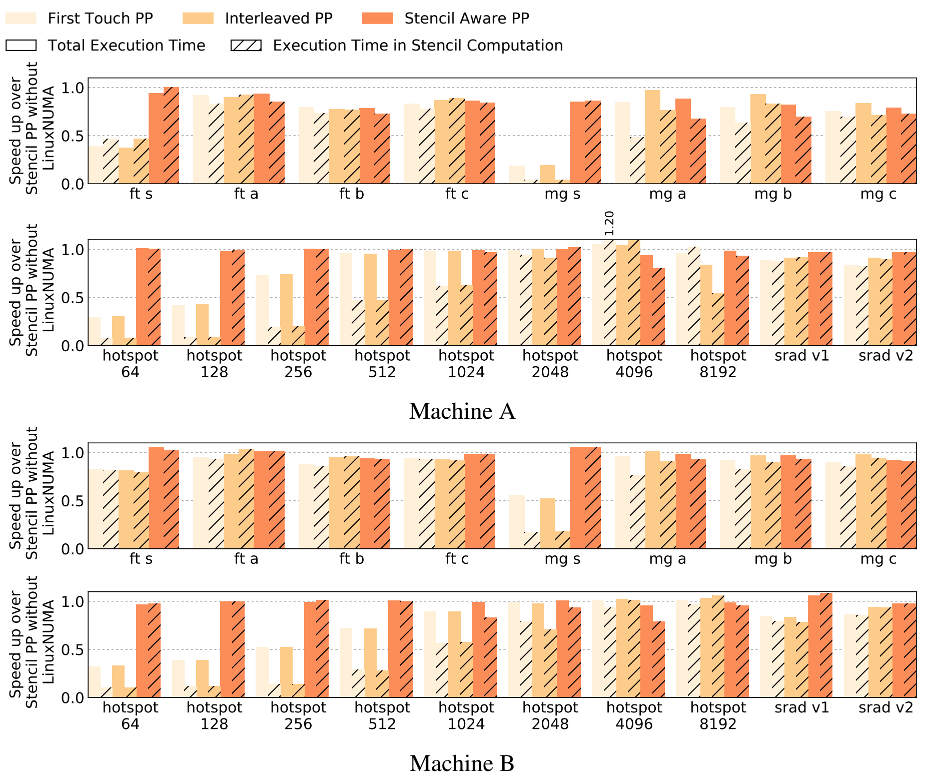
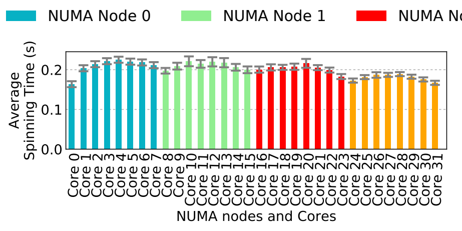


Fig. 7: Speed up with LinuxNUMA over our Stencil Aware PP without LinuxNUMA. Most speed ups are below 1.0 i.e. are slow downs indicating the superiority of our scheme.

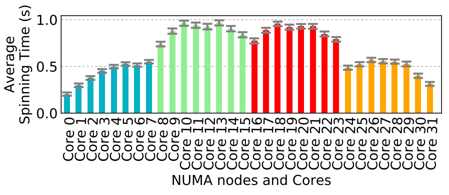
sults show that Stencil Aware PP is a viable alternative to the current built-in schemes of Linux and should be used instead.

6.2 Performance Degradation through NUMA Balancing

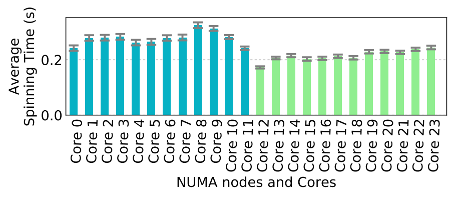
To assess the impact of automatic NUMA Balancing (from now referred to as *"LinuxNUMA"*) we compare Stencil Aware PP without LinuxNUMA against all PP schemes with LinuxNUMA in Figure 7. Most of the applications perform worse with LinuxNUMA than Stencil Aware PP without LinuxNUMA. In the few cases where LinuxNUMA is beneficial the differences are small (max: 0.09x) and present on only one machine, or the PP schemes already perform slightly better than our scheme even without LinuxNUMA (see hotspot with input size 4096 and 8192 in Fig. 6.1) and continue to do so with LinuxNUMA. It is important to note that LinuxNUMA degrades the performance of Stencil Aware PP in all cases, by up to 0.27x with mg and input c on machine A. Thus, in addition to using Stencil Aware PP, LinuxNUMA should be deactivated for stencil computations (for example, in stencil skeleton libraries).



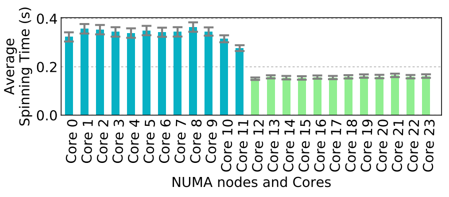
(b) Parallel section 1



(c) Parallel section 2



(d) Parallel section 1



(e) Parallel section 2

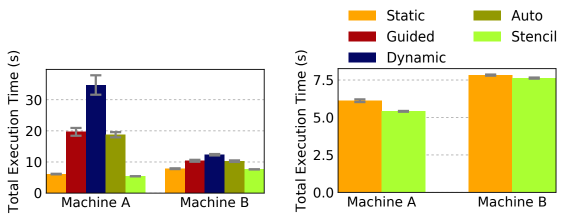
Fig. 8: Uneven idle times with srad v2 on Machine A (a + b) and B (c + d).

6.3 Bad Work Distribution and Stencil Aware Work Distribution

The uneven idle time effect is present to varying degrees across our benchmark suite and machines. The expected variation in idle times occurs with the srad v2 benchmark on Machine A and B as shown in Fig. 8. Adjacent cores in Fig. 8 share data and cores that share data with a remote NUMA node, like core 7 and 8 on machine A have the expected, consistently lower spinning times which indicate that they are slowed down by remote memory accesses as discussed in Sec. 4. The expected variations are also visible for the NAS-PB mg benchmark on Machine A with input size a and b. However, the idle time differences are very small and statistically insignificant.
In contrast, the effect is not visible in other cases for the following reasons. For the NAS-PB ft benchmark implementation the stencil has a 2D neighborhood and the input buffer is 3D. The computation is parallelized over the third dimension of the input buffer and, therefore, the cores do not share data, and so the idle time imbalance is not present. The data set of srad v1 fits into the combined LLCs which causes the latency differences between local and remote memory accesses to the LLCs to be very small and so the variations of the idle times become too small to report.

6.4 NUMA and Stencil Aware Work Distribution

We compare our work distribution scheme with a range of alternatives, all of which are extensions of our basic pinned, Stencil Aware PP scheme as introduced in Sec. 4. These are created by selecting different OpenMP schedules for the stencil iterations. Figure 9 shows execution times with OpenMP's schedules and the stencil aware work distribution. "Static" corresponds to our basic stencil aware PP scheme from Sec. 4. Our new scheme achieves improved performance of up to 1.1x for srad v2. In contrast,



(a) (b)

Fig. 9: Execution times of srad v2 with OpenMP schedules and with our NUMA and stencil aware work distribution (a). A direct comparison of the best OpenMP schedule with the stencil aware work distribution (b). *Static*, *guided*, *dynamic* and *auto* are OpenMP schedules. The stencil aware work distribution reflects the uneven idle times discussed in Sec. 6.3.

none of the standard OpenMP schedules can mitigate the negative effects of the uneven idle time distribution. This shows that our scheme addresses the work imbalance caused by variable NUMA memory latencies. To make this applicable within a generic stencil scheme it would be important to predict the stencil instances for which an improvement is achievable.

7 Related Work

We first review state of the art page placement schemes, then NUMA stencil optimizations and, lastly, work on NUMA aware schedulers and work distribution.
Carrefour reduces congestion on interconnects and memory controllers via page collocation, replication and interleaving [14]. It has to monitor memory accesses to inform the usage of these techniques and cannot, like our approach, leverage information about the structure of a computation that is available prior to execution.
Mechanisms for automatic thread and page migration have been developed for the Linux kernel [3, 15–17]. These mechanisms monitor memory accesses and, therefore, cannot act until sufficient data is collected. This monitoring based approach can result in pathological behavior such as page [18, 19] and task bouncing [3, 20]. Our scheme can find an optimal task and page placements before a computation starts.
Stencil aware memory management for NUMA systems has been mentioned in side notes [21–24]. To the best of our knowledge we are the first to report an in-depth analysis of stencil aware memory management for NUMA systems with a broad range of stencil types and problem sizes.
Pilla et al. present a NUMA-Aware scheduler [25]. The scheduler considers the communication between concurrently executing threads and collocates them on NUMA nodes to minimize communication across CPU boundaries. This approach suffers from similar problems as other monitoring based approaches (see above). Chen and Olivier et al. present work stealing for NUMA systems [26, 27]. Their approaches reschedules work at run time in case work was not distributed equally. Our approach distributes work

equally before a computation starts by taking the memory system of the target system and stencil specific memory access patterns into account.

8 Conclusion and Future Work

We argue that NUMA optimizations, should be embedded in skeleton implementations by utilizing implicit knowledge encoded in them. We present a case study with stencil computations. We evaluate a stencil aware page placement (PP) scheme that exploits the regular and predictable stencil memory access patterns. We then investigate two further optimizations that build on Stencil Aware PP. Firstly, we show that automatic NUMA Balancing, an advanced optimization technique in the Linux kernel, degrades the performance of stencil computations when Stencil Aware PP is used. Secondly, we investigate a novel stencil and NUMA aware work distribution scheme. Stencil Aware PP improves the performance of applications by up to 12x and never degrades performance. NUMA Balancing degrades the performance of stencil applications by up to 0.27x if stencil aware PP is used and should be deactivated. Finally, we show that the performance of some stencil computations can be further improved by up to 1.1x with a stencil aware work distribution. Future work includes a heuristic that predicts whether the new stencil aware work distribution scheme will be beneficial. Furthermore, we will consider the fact that, in multiprogrammed scenarios, the proposed deactivation of LinuxNUMA has an impact on other applications that run on the system. Finally, we will investigate NUMA optimizations for further skeletons.

9 Acknowledgments

Thanks to Tom Deakin and Simon McIntosh-Smith from the University of Bristol for giving us access to one of their machines. Grant EP/L01503X/1 for the University of Edinburgh School of Informatics Centre for Doctoral Training in Pervasive Parallelism from EPSRC supported this work.

References

1. Talbot, S.A.M., Kelly, P.H.J. In: Stable Performance for CC-NUMA Using First-Touch Page Placement and Reactive Proxies. Springer US, Boston, MA (1998)
2. McCurdy, C., Vetter, J.: Memphis: Finding and fixing NUMA-related performance problems on multi-core platforms. In: Proceedings of ISPASS. (2010)
3. van Riel, R., Chegu, V.: Automatic NUMA balancing. In: Red Hat Summit. (2014)
4. Gaud, F., Lepers, B., Funston, J., Dashti, M., Fedorova, A., Quéma, V., Lachaize, R., Roth, M.: Challenges of memory management on modern NUMA system. Queue **13**(8) (2015)
5. Cole, M.: Bringing skeletons out of the closet: a pragmatic manifesto for skeletal parallel programming. Parallel computing **30**(3) (2004)
6. González-Vélez, H., Leyton, M.: A survey of algorithmic skeleton frameworks: high-level structured parallel programming enablers. Software: Practice and Experience **40**(12) (2010)
7. Enmyren, J., Kessler, C.W.: SkePU: a multi-backend skeleton programming library for multi-GPU systems. In: Proceedings of HLPP. (2010)

8. Ribeiro, C.P., Mehaut, J.F., Carissimi, A., Castro, M., Fernandes, L.G.: Memory affinity for hierarchical shared memory multiprocessors. In: Proceedings of ICS. (2009)
9. Yang, R., Antony, J., Rendell, A., Robson, D., Strazdins, P.: Profiling directed NUMA optimization on Linux systems: A case study of the gaussian computational chemistry code. In: Proceedings of IPDPS. (2011)
10. Bircsak, J., Craig, P., Crowell, R., Cvetanovic, Z., Harris, J., Nelson, C.A., Offner, C.D.: Extending OpenMP for NUMA machines. In: Proceedings of ICS. (2000)
11. Broquedis, F., Furmento, N., Goglin, B., Wacrenier, P.A., Namyst, R.: ForestGOMP: an efficient OpenMP environment for NUMA architectures. International Journal of Parallel Programming **38**(5) (2010)
12. Che, S., Boyer, M., Meng, J., Tarjan, D., Sheaffer, J.W., Lee, S.H., Skadron, K.: Rodinia: A benchmark suite for heterogeneous computing. In: Proceedings of IISWC. (2009)
13. Baily, D., Barszcz, E., Barton, J., Browing, D., Carter, R., Dagum, L., Fatoohi, R., Fineberg, S., Frederickson, P., Lasinski, T., Schreiber, R., Simon, H., Venkatakrishnan, V., Weeratunga, S.: The nas parallel benchmarks. Technical Report RNR-94-007, NASA Ames Research Center (1994)
14. Dashti, M., Fedorova, A., Funston, J., Gaud, F., Lachaize, R., Lepers, B., Quema, V., Roth, M.: Traffic management: a holistic approach to memory placement on numa systems. In: ACM SIGPLAN Notices. Volume 48., ACM (2013)
15. Corbet, J.: Autonuma: the other approach to NUMA scheduling. https://lwn.net/Articles/488709/ (March 2012)
16. Corbet, J.: Toward better NUMA scheduling. https://lwn.net/Articles/486858/ (March 2012)
17. Gorman, M.: Foundation for automatic NUMA balancing. https://lwn.net/Articles/523065/ (November 2012)
18. Bolosky, W., Fitzgerald, R., Scott, M.: Simple but effective techniques for NUMA memory management. ACM SIGOPS Operating Systems Review **23**(5) (1989)
19. Gaud, F., Lepers, B., Decouchant, J., Fuston, J., Fedorova, A., Quéma, V.: Large pages may be harmful on NUMA systems. In: Proceedings of USENIX ATC. (2014)
20. Gorman, M.: Automatic NUMA balancing V4. https://lwn.net/Articles/526097/ (November 2012)
21. Christen, M., Schenk, O., Burkhart, H.: Patus: A code generation and autotuning framework for parallel iterative stencil computations on modern microarchitectures. In: Proceedings of IPDPS. (2011)
22. Kamil, S., Chan, C., Oliker, L., Shalf, J., Williams, S.: An auto-tuning framework for parallel multicore stencil computations. In: Proceedings of IPDPS. (2010)
23. Shaheen, M., Strzodka, R.: NUMA aware iterative stencil computations on many-core systems. In: Proceedings of IPDPS. (2012)
24. Lin, P.H., Yi, Q., Quinlan, D., Liao, C., Yan, Y.: Automatically optimizing stencil computations on many-core NUMA architectures. In: Proceedings of LCPC, Springer (2016)
25. Pilla, L.L., Ribeiro, C.P., Cordeiro, D., Bhatele, A., Navaux, P.O., Méhaut, J.F., Kalé, L.V.: Improving parallel system performance with a NUMA-aware load balancer. Technical Report TR-JLPC-11-02, INRIA-Illinois Joint Laboratory on Petascale Computing (2011)
26. Chen, Q., Guo, M., Guan, H.: LAWS: locality-aware work-stealing for multi-socket multi-core architectures. In: Proceedings of the Intl. Conference on Supercomputing. (2014)
27. Olivier, S.L., Porterfield, A.K., Wheeler, K.B., Spiegel, M., Prins, J.F.: OpenMP task scheduling strategies for multicore NUMA systems. IJHPCA **26**(2) (2012)