Exploiting Hidden Non-uniformity of Uniform Memory Access on Manycore CPUs

Balazs Gerofi[1], Masamichi Takagi[2], and Yutaka Ishikawa[1]

[1] Graduate School of Information Science and Technology,
The University of Tokyo
bgerofi@il.is.s.u-tokyo.ac.jp, ishikawa@is.s.u-tokyo.ac.jp
[2] RIKEN Advanced Institute for Computational Science
masamichi.takagi@riken.jp

Abstract. As the rate of CPU clock improvement has stalled for the last decade, increased use of parallelism in the form of multi- and many-core processors has been chased to improve overall performance. Current high-end manycore CPUs already accommodate up to hundreds of processing cores. At the same time, these architectures come with complex on-chip networks for inter-core communication and multiple memory controllers for accessing off-chip RAM modules. Intel's latest Many Integrated Cores (MIC) chip, also called the Xeon Phi, boasts up to 60 CPU cores (each with 4-ways SMT) combined with eight memory controllers. Although the chip provides Uniform Memory Access (UMA), we find that there are substantial (as high as 60%) differences in access latencies for different memory blocks depending on which CPU core issues the request, resembling Non-Uniform Memory Access (NUMA) architectures.
Exploiting the aforementioned differences, in this paper, we propose a memory block latency-aware memory allocator, which assigns memory addresses to the requesting CPU cores in a fashion that it minimizes access latencies. We then show that applying our mechanism to the A-star graph search algorithm can yield performance improvements up to 28%, without any need for modifications to the algorithm itself.

1 Introduction

Although Moore's Law continues to drive the number of transistors per square mm, the recent stop of frequency and Dennard scaling caused an architectural shift in processor design towards multi- and many-core CPUs. Multicore processors usually implement a handful of complex cores that are optimized for fast single-thread performance, while manycore units come with a large number of simpler and slower but much more power-efficient cores that are optimized for throughput-oriented parallel workloads [1].

There have been manycore chips already built with 48 [2], 64 [3], 72 [4] cores and even an experimental processor with 1000 cores [5] has been announced. The Intel® Xeon Phi™ product family, also referred to as Many Integrated Cores (MIC), is Intel's latest manycore CPU providing sixty x86 cores [6].

Although manycore CPUs tend to come with complex networks-on-chip (NOC) and with multiple memory controllers [7], with respect to memory access there are mainly two approaches. Uniform memory access (UMA) architectures provide uniform access latencies for the entire physical memory regardless which CPU core is generating the request, while on the other hand, non-uniform memory access (NUMA) architectures allow explicit differences in terms of memory access latencies depending on the physical address and the CPU core that is accessing it [8]. Despite the fact that the large number of CPU cores and complex on-chip networks make it increasingly difficult to keep access latencies uniform, most of the existing manycore processor do follow the UMA approach for the sake of easy programmability.

The Xeon Phi also provides uniform memory access officially. However, we find that memory access latencies differ significantly depending on which CPU core accesses a given physical address. Access latencies to the same memory block can vary by up to 60% when issuing requests from different CPU cores, resembling NUMA architectures. Notice, that the above mentioned latency differences are at the memory level, unlike for caches in NUCA architectures [9].

Applications which access small data structures in a relatively random fashion, such as those operating on Recursive Data Structures (RDS) may experience significant performance degradation simply by accessing memory blocks that are located *far* from the CPU core that generates the request. RDSs include linked lists, trees, graphs, etc., where individual nodes are dynamically allocated from the heap, and nodes are linked together through pointers to form the overall structure [10]. For example, the A* (A-star) algorithm [11], a widely used graph search heuristic in artificial intelligence, exhibits exactly such characteristics.

Inspired by the above described observation, in this paper, we propose a memory allocator that is memory block latency-aware, i.e., it allocates memory to particular CPU cores in a fashion that it minimizes access latencies. In summary, we make the following contributions:

- We point out that *hidden non-uniformity* in otherwise uniform memory access architectures *can be significant on manycore CPUs.*
- We propose *a memory allocator*, which is optimized for allocating small data structures in a *memory block latency-aware* fashion and it lays out memory in a way that access latencies for the requesting CPU cores are minimized.
- We show that applying our allocator can yield up to 28% performance improvements for the A-star graph search algorithm solving a 16-tile puzzle, without any need for modifications to the application itself.

The rest of this paper is organized as follows. We begin with providing a detailed overview of the Xeon Phi along with measurements on memory block latencies as seen from different CPU cores in Section 2. Section 3 discusses our target application, the A-star algorithm and Section 4 describes the proposed memory allocator. Experimental evaluation is given in Section 5. Section 6 provides further discussion, Section 7 surveys related work, and finally, Section 8 presents future plans and concludes the paper.

2 Background and Motivation

In this Section we provide an overview of the Xeon Phi processor focusing in particular on components that contribute to memory access latency. The architectural overview of the Intel Xeon Phi processor is shown in Figure 1. The chip we used in this paper comes on a PCI Express card, with 60 CPU cores, where each core supports four hyperthreads (i.e., 4-way symmetric multithreading).

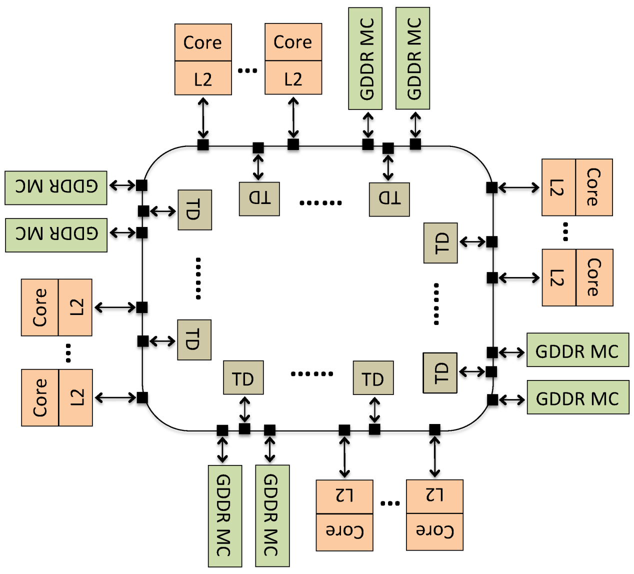


Fig. 1. Architectural overview of the Intel Xeon Phi manycore CPU. *The chip consists of 60 CPU cores, each core with 4-way symmetric multithreading and a 512kB private slice of the unified L2 cache. There are 8 GDDR memory controllers and 64 cache tag directories, which are all connected with a bidirectional ring.*

Each processor core runs on up to 1.2GHz and besides the relatively low clock frequency (compared to standard multi-core Xeon chips), cores on the Xeon Phi have no support for out-of-order execution [6]. All CPU cores have their own 32kB L1 caches (both data and instruction) and a 512kB private slice of the unified L2 cache. Both the L1 and L2 caches use the standard MESI protocol [12] for maintaining the shared state among cores. To address potential performance limitations resulting from the lack of an O (Owner) state found in the MOESI protocol [13], the Intel Xeon Phi processor coherence system uses a distributed tag directory (DTD) of ownership similar to that implemented in many multiprocessor systems [14].

The card is equipped with 8 Gigabytes of GDDR5 memory for which there are eight GDDR5 memory controllers encompassed in the chip and all components are connected via a bi-directional ring. Intel does not provide detailed information on how memory blocks are assigned to DTDs and memory controllers, but assumably a hash function based on the address of the line is used [15]. There is also no support for modifying the mapping.

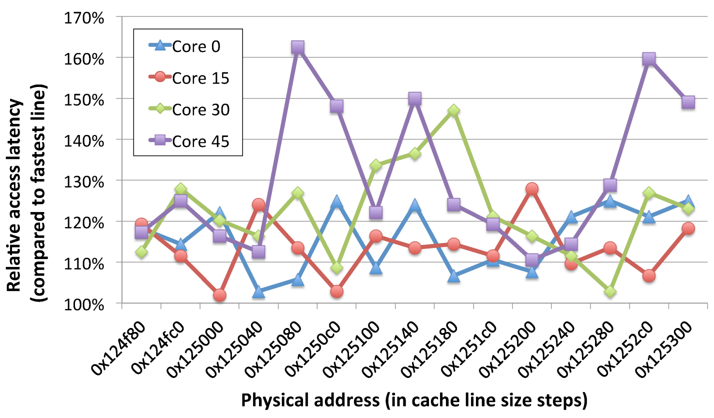


Fig. 2. Differences in memory access latency on subsequent memory blocks seen from four CPU cores. *Data is ensured to be in RAM by invalidating both L1 and L2 caches before each access and prefetching is disabled.*

When a core encounters a cache miss, it requests the line from the corresponding DTD and eventually, from the corresponding memory controller. Considering the distances between CPU cores, DTDs, and memory controllers, one can expect that access latencies to the same memory block likely vary across different CPU cores.

We have developed a simple benchmark tool that measures differences in memory latencies depending on which CPU core accesses a particular memory block. Note that data is always ensured to be in RAM by invalidating both L1 and L2 caches before each access as well as disabling the compiler generated prefetch instructions. Figure 2 reveals our findings for a couple of subsequent memory blocks as seen from four different CPU cores. The X axis shows the physical address of the given memory block, while Y axis represents the relative access latency compared to the fastest access (lower is better). As shown, there are significant differences among the values. For example, accessing the physical address $0x125080$ from CPU core 45 is approximately 60% slower than from core 0. Such differences can easily show up in application performance, especially when taking into account that the Xeon Phi cores can do only in-order execution.

3 The A* Algorithm

This Section gives an overview of the A* (A-star) algorithm [11] emphasizing attributes that can be exploited by a memory block latency-aware memory allocator for improving overall performance.

Listing 1.1. Pseudo code of the A* algorithm

```
function A*(start,goal)
  closedset := the empty set    // The set of nodes already evaluated.
  openset := {start}    // The set of tentative nodes to be evaluated.
  start.came_from := NULL

  start.g_score := 0    // Cost from start along best known path.

  // Estimated total cost from start to goal.
  start.f_score := start.g_score + heuristic_cost_estimate(start, goal)

  while openset is not empty
    current := the node in openset having the lowest f_score value
    if current = goal
      return reconstruct_path(goal)

    remove current from openset
    add current to closedset
    for each neighbor in neighbor_nodes(current)
      // Find neighbor in closedset
      if neighbor in closedset
        continue

      tentative_g_score := current.g_score +
                               dist_between(current, neighbor)

      if neighbor not in openset or
        tentative_g_score < neighbor.g_score
        neighbor.came_from := current
        neighbor.g_score := tentative_g_score

        neighbor.f_score := neighbor.g_score +
          heuristic_cost_estimate(neighbor, goal)

        if neighbor not in openset
          add neighbor to openset

    return failure

function reconstruct_path(current_node)
  if current_node.came_from
    p := reconstruct_path(current_node.came_from)
    return (p + current_node)
  else
    return current_node
```

A* is an informed best-first graph search algorithm which aims at finding the lowest cost path from a given start to a goal node. During the search, both the cost from the start node to current node and the estimated cost from the current node to a goal state are minimized [16]. The pseudo code of the A* algorithm is shown in Listing 1.1. The A* algorithm uses two sets of nodes for housekeeping, the so-called *open-set* holds nodes to which the search can continue in subsequent steps, while the *closed-set* stores nodes that have been already evaluated.

Depending on the problem being solved, these sets can grow considerably large while at the same time lookup operations from these sets are required in each iteration of the algorithm (see line 20 and 26 of the pseudo code). In order to attain good lookup performance hash tables are normally utilized, however, as a result memory accesses come with very low data locality, i.e., following a nearly random access pattern. Moreover, problem state (i.e., a node of the graph) can be often represented with relatively small data structures, fitting easily into the size of one cache line. As we will show later through quantitative evaluation, the above mentioned characteristics of the A* algorithm suit well the assumptions we described earlier in Section 1.

With respect to utilizing multiple CPU cores, since we are focusing on the effect of hidden non-uniformity of memory accesses, we simply use different goal states on different CPU cores. This keeps the parallel code simple, because open and closed sets are separated per core, and it also eliminates the possibility of false sharing. Note that there are several studies on how to parallelize efficiently the A* algorithm when searching a shared global state [17], [18], [19] and further investigating this issue is outside the scope of this paper.

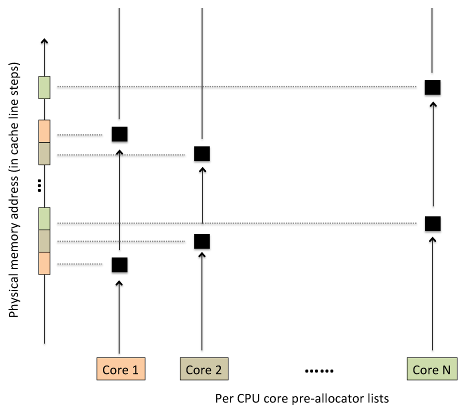
4 Memory Block Latency-aware Memory Allocator

We now describe the design and implementation of the memory block latency-aware memory allocator.

As mentioned earlier we developed a simple tool that measures access latencies to a particular memory block from different CPU cores. We used this tool to build a latency data base, in which for each memory block (i.e., the physical memory address of the block) access latencies for all CPU cores are stored. The basic idea is that when memory is requested by the application the runtime system pre-allocates a large chunk of memory and queries the physical addresses of the corresponding pages from the acquired mapping. For the purpose of obtaining the physical translation of an arbitrary virtual address, we introduced a new system call (see below for details on the kernel we used). Once the physical addresses are known, the latency data base is consulted to determine which memory blocks have low latency access from CPU cores used by the application and the runtime places the addresses onto the corresponding per-core allocator lists. Although we are explicitly targeting small memory requests in this paper, it is worth mentioning that larger allocations can be still satisfied simply by falling back to the regular glibc memory allocator. For further discussion on larger allocation sizes as well as on the memory efficiency of the proposed system refer to Section 6.

The architecture of the memory block latency-aware allocator is shown in Figure 3. The colored addresses on the left of the Figure represent memory blocks which can be accessed with low latency by the CPU core designated by the same color. The per-core lists hold these addresses (denoted by the black squares) for each application core and memory allocation requests are directly satisfied from these lists.

With regards to implementation details, the RIKEN Advanced Institute of Computational Science and the Information Technology Center at the Univer-



Physical memory address (in cache line steps)
Core 1
Core 2
Core N
Per CPU core pre-allocator lists

Fig. 3. Memory block latency-aware per-core allocator lists. *The colored rectangles on the left indicate low latency memory blocks when accessed from a CPU core denoted by the same color.*

sity of Tokyo have been designing and developing a new scalable system software stack for a new heterogeneous supercomputer. Part of this project is an operating system kernel targeting manycore processors [20]. Our OS kernel is binary compatible with Linux and supports all system calls so that applications using pthreads can be executed without modifications.

We have implemented the proposed allocator on top of our custom OS kernel in the form of a library interposed between the application and glibc. We note that as a proof of concept our prototype implementation distributes memory blocks during initialization (i.e., memory pre-allocation) phase of the application, but utilizing dedicated allocator threads the technique can be easily adapted to an actual runtime solution. As for the memory block latency data base, it is simply a collection of files which we store on local SSDs for fast access. It is also worth mentioning that our custom kernel does not migrate application threads among CPU cores, i.e., threads are pinned to the same core throughout the whole execution of an application. This is with particular importance, since memory addresses returned by the allocator yield low latency access only for the core which performs the allocation and moving a thread to another core would defeat the very purpose of the policy. Besides the custom system call for obtaining physical address for a user mapping, we also provide a special call that returns the APIC CPU core ID so that threads can easily determine where they execute.

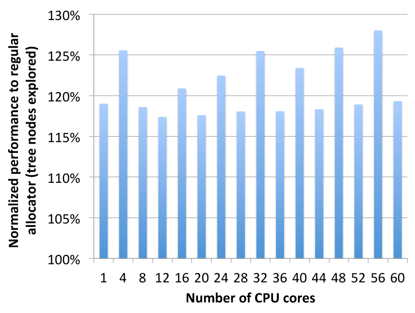
5 Evaluation

5.1 Experimental Setup

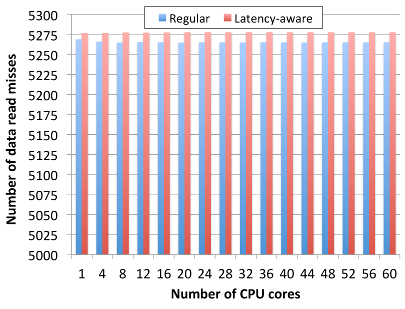
Throughout our experiments the host machine was an Intel® Xeon® CPU E5-2670. For the manycore processor we used the *Knights Corner* Xeon Phi 5110P card, which is connected to the host machine via the PCI Express bus. As mentioned earlier, it provides 8GB of RAM and a single chip with 60 x86 cores running on up to 1.2GHz, each processor core supporting a multithreading depth of four. The chip includes coherent L1 and L2 caches and the inter-processor network is a bidirectional ring [6].

5.2 Results

We used the A* algorithm solving the 16 tile puzzle problem to evaluate our proposal, but it is worth noting that our approach could be generally applied to a wide range of Recursive Data Structures (RDSs). RDS includes familiar objects such as linked lists, trees, graphs, etc., where individual nodes are dynamically allocated from the heap, and nodes are linked together through pointers to form the overall structure [10].



(a) Relative performance to regular allocator.



(b) Average number of memory read accesses that miss the internal cache per A* iteration. *(On CPU core 0.)*

Fig. 4. Performance of memory block latency-aware allocator compared to regular pre-allocation on the A* algorithm solving a 16 tile puzzle.

Specifically, we used a publicly available A* code [21] as reference implementation. The state space of the 16 tile puzzle is large enough so that the search graph does not fit into the L2 cache of the Xeon Phi, and thus making the lookup operations generate actual memory accesses.

We used two configurations of the application. First, we ran with regular memory pre-allocator, i.e., memory is simply divided among the threads. We then modified the allocation routine to call into our library and use the memory block latency-aware allocation. We measured the number of graph nodes the algorithm explores in unit time and report the normalized improvement of the memory block aware allocator compared to the regular solution. Results are indicated by Figure 4a, where each measurement was repeated five times and the chart shows the average values.

As seen, performance improvement scales from 17% to 28% and varies depending on the number of CPU cores utilized in the application. Initially we expected there would be a gradual increase in performance improvement with the growing number of cores, assuming that the on-chip traffic is better balanced among the CPU cores and the memory controllers, an observation which had been also pointed out for NUMA architectures previously [22]. Surprisingly, however, there seem to be no direct relation between the performance improvement and the number of cores involved in the execution. As the Figure shows, utilizing 12 and 20 cores yields the lowest improvement. Besides the random nature of the regular allocator in terms of memory block latencies, we believe the distributed tag directory based cache coherence may also contribute to this effect. Intel doesn't provide any details about the characteristics of the on-chip network and thus it is hard to assess whether traffic congestion occurs due to communication between cores and the tag directories or the memory controllers. Nevertheless, we do observe the highest improvement for 56 CPU cores.

We also measured the number of read accesses that missed the internal data cache on CPU core 0, where the same goal state was used across all runs to ensure fair comparison. Results are shown in Figure 4b. As the number of cache misses is approximately the same regardless the underlying memory allocator, we believe that the observed performance improvement results indeed from the lower latency memory accesses.

6 Discussion

This Section provides a short discussion on some of the limitations of our proposed approach. First, since we exploit memory access latency differences at the memory block level, allocations larger than a memory block size (i.e., 64bytes on x86 64bit) cannot be laid out in a continuous fashion on to low latency memory blocks. At present, we simply return a regular allocation, however, splitting structures in a clever way could also help to overcome this limitation [23]. Second, the smaller the number of cores utilized by the application, the lower the ratio of low latency memory blocks becomes corresponding to the participating cores. Consequently, our allocator provides the best memory usage efficiency when the entire chip is utilized.

Third, we also need to note that our technique favors applications, where the per-core data sets are distinct. Communication between the cores of course is inevitable, and if necessary data from one core's low latency line could be copied over to another one's, such as it would be required for the EM3D application [10]. Forth, one might argue that spreading memory allocations over low latency memory blocks will increase the price of TLB misses. In our experiments we used large pages for memory mappings and both in case of regular and low latency allocators, the per-core memory used could be covered by the L2 TLB entries.

Despite the above mentioned restrictions, we emphasize that our intention is to demonstrate that it is possible to take advantage of hidden memory latency differences in current many-core CPUs.

7 Related Work

As we mentioned earlier, the hidden non-uniformity of the UMA property officially provided by the Xeon Phi closely resembles non-uniform memory access (NUMA) architectures.

A large body of management policies for memory and thread placement in NUMA architectures have been previously proposed. Bolosky et al investigated page replacement policies so that data are placed close to the process that is using them [24]. LaRowe et al. built an analytic model of the memory system performance of a local/remote NUMA architecture and investigated heuristics when pages should be moved or remotely referenced [25]. Verghese et al. studied the performance improvements provided by OS supported dynamic page migration and replication in NUMA environments where remote access latencies were significantly higher than those to local memory [26]. Avdic et al. demonstrated the correlation of memory access latency with difference between cores and memory controllers through parallel sorting on the Intel SCC [27]. Although the goal of the above mentioned studies is similar in nature to ours, i.e., to optimize for access locality, they explicitly deal with NUMA system where the granularity of access inequality is at least page size. On the contrary, we exploit hidden non-uniformities at the memory block level.

Some recent studies approach memory management issues from the aspect of resource contention. Knauerhase et al. argued that the OS can use data obtained from dynamic runtime observation of task behavior to ameliorate performance variability and more effectively exploit multicore processor resources, such as the memory hierarchy [28]. Another recent work points out that performance degradation in current NUMA systems doesn't mainly derive from the cost of remote accesses. Instead, congestion on memory controllers and interconnects caused by memory traffic from data-intensive applications hurts performance much more [22]. As the Xeon Phi's on-chip network connects a large number of various components, network congestion during communication among CPU cores, cache tag directories and memory controllers likely constitute to performance degradation of memory intensive applications. We believe that part of the merit of assigning low latency memory blocks to CPU cores is the alleviation of on-chip traffic congestion.

8 Conclusion and Future Work

Many-core CPUs come with an increasing number of components, such as CPU cores, memory controllers, cache tag directories, etc., and the on-chip networks connecting these components are becoming more and more complex. Nevertheless, uniform memory access is still the most frequently provided memory model due to its ease of programmability.

In this paper, we have pointed out that many-core CPUs, such as Intel's Xeon Phi, can exhibit substantial hidden non-uniformity in memory access latencies among CPU cores accessing the same memory block. To the best of our knowledge, this is the first time such differences have been shown for a UMA architecture. We have proposed a latency-aware memory allocator and demonstrated its superior performance on the A* search algorithm. Most importantly, we encourage chip manufacturers not to hide such differences or at least to provide the system with the ability to reconfigure mappings so that NUMA properties could be explicitly leveraged at the software level.

In the future, we intend to look at further possible usage scenarios accelerating applications relying on recursive data structures, such as the EM3D or Barnes-Hut's N-body problem [10] and Monte-Carlo based tree search algorithms.

Acknowledgment

This work has been partially supported by the CREST project of the Japan Science and Technology Agency (JST) and by the National Project of MEXT called Feasibility Study on Advanced and Efficient Latency Core Architecture.

References

1. Saha, B., Zhou, X., Chen, H., Gao, Y., Yan, S., Rajagopalan, M., Fang, J., Zhang, P., Ronen, R., Mendelson, A.: Programming model for a heterogeneous x86 platform. In: Proceedings of the 2009 ACM SIGPLAN conference on Programming language design and implementation. PLDI '09, New York, NY, USA, ACM (2009) 431–440
2. Intel Corporation: Single-Chip Cloud Computer. `https://www-ssl.intel.com/content/www/us/en/research/intel-labs-single-chip-cloud-computer.html` (2010)
3. Adapteva: Epiphany-IV 64-core 28nm Microprocessor (E64G401). `http://www.adapteva.com/epiphanyiv` (2014)
4. Tilera: TILE-Gx8072 Processor Product Brief. `http://www.tilera.com/sites/default/files/images/products/TILE-Gx8072_PB041-03_WEB.pdf` (2014)
5. The University of Glasgow: Scientists squeeze more than 1,000 cores on to computer chip. `http://www.gla.ac.uk/news/archiveofnews/2010/december/headline_183814_en.html` (2010)
6. Intel Corporation: Intel Xeon Phi Coprocessor System Software Developers Guide. `https://www-ssl.intel.com/content/www/us/en/processors/xeon/xeon-phi-coprocessor-system-software-developers-guide.html` (2013)
7. Nychis, G.P., Fallin, C., Moscibroda, T., Mutlu, O., Seshan, S.: On-chip Networks from a Networking Perspective: Congestion and Scalability in Many-core Interconnects. SIGCOMM '12, New York, NY, USA, ACM (2012) 407–418
8. Lameter, C.: NUMA (Non-Uniform Memory Access): An Overview. ACM Queue **11**(7) (2013) 40
9. Kim, C., Burger, D., Keckler, S.: Nonuniform cache architectures for wire-delay dominated on-chip caches. Micro, IEEE **23**(6) (Nov 2003) 99–107
10. Luk, C.K., Mowry, T.C.: Compiler-based prefetching for recursive data structures. ASPLOS VII, New York, NY, USA, ACM (1996) 222–233
11. Hart, P., Nilsson, N., Raphael, B.: A Formal Basis for the Heuristic Determination of Minimum Cost Paths. Systems Science and Cybernetics, IEEE Transactions on **4**(2) (July 1968) 100–107

12. Ivanov, L., Nunna, R.: Modeling and Verification of Cache Coherence Protocols. In: Circuits and Systems, 2001. ISCAS 2001. The 2001 IEEE International Symposium on. Volume 5. (2001) 129–132 vol. 5
13. Hackenberg, D., Molka, D., Nagel, W.E.: Comparing Cache Architectures and Coherency Protocols on x86-64 Multicore SMP Systems. In: Proceedings of the 42Nd Annual IEEE/ACM International Symposium on Microarchitecture. MICRO 42, New York, NY, USA, ACM (2009) 413–422
14. Hennessy, J.L., Patterson, D.A.: Computer Architecture - A Quantitative Approach (5. Edition). Morgan Kaufmann (2012)
15. Ramos, S., Hoefler, T.: Modeling Communication in Cache-coherent SMP Systems: A Case-study with Xeon Phi. HPDC '13, New York, NY, USA, ACM (2013) 97–108
16. Russell, S., Norvig, P.: Artificial Intelligence: A Modern Approach. 3rd edn. Prentice Hall Press, Upper Saddle River, NJ, USA (2009)
17. Dutt, S., Mahapatra, N.: Parallel A* algorithms and their performance on hypercube multiprocessors. In: Parallel Processing Symposium, 1993., Proceedings of Seventh International. (Apr 1993) 797–803
18. Burns, E., Lemons, S., Zhou, R., Ruml, W.: Best-first Heuristic Search for Multicore Machines. In: Proceedings of the 21st International Jont Conference on Artifical Intelligence. IJCAI'09, San Francisco, CA, USA, Morgan Kaufmann Publishers Inc. (2009) 449–455
19. Rios, L.H.O., Chaimowicz, L.: A Survey and Classification of A* Based Best-First Heuristic Search Algorithms. In: Advances in Artificial Intelligence SBIA 2010. Volume 6404 of Lecture Notes in Computer Science. Springer Berlin Heidelberg (2010) 253–262
20. Gerofi, B., Shimada, A., Hori, A., Ishikawa, Y.: Partially Separated Page Tables for Efficient Operating System Assisted Hierarchical Memory Management on Heterogeneous Architectures. CCGrid '13 (may 2013)
21. Heyes-Jones, J.: A* Algorithm Tutorial. `http://heyes-jones.com/astar.php` (2013)
22. 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. ASPLOS '13, New York, NY, USA, ACM (2013) 381–394
23. Chilimbi, T.M., Davidson, B., Larus, J.R.: Cache-conscious Structure Definition. PLDI '99, New York, NY, USA, ACM (1999) 13–24
24. Bolosky, W., Fitzgerald, R., Scott, M.: Simple but Effective Techniques for NUMA Memory Management. SOSP '89, New York, NY, USA, ACM (1989) 19–31
25. LaRowe, Jr., R.P., Ellis, C.S., Holliday, M.A.: Evaluation of NUMA Memory Management Through Modeling and Measurements. IEEE Trans. Parallel Distrib. Syst. **3**(6) (November 1992) 686–701
26. Verghese, B., Devine, S., Gupta, A., Rosenblum, M.: Operating system support for improving data locality on cc-numa compute servers. ASPLOS VII, New York, NY, USA, ACM (1996) 279–289
27. Avdic, K., Melot, N., Keller, J., Kessler, C.: Parallel sorting on Intel Single-Chip Cloud Computer. In: In Proceedings of the 2nd Workshop on Applications for Multi and Many Core Processors. (2011)
28. Knauerhase, R., Brett, P., Hohlt, B., Li, T., Hahn, S.: Using os observations to improve performance in multicore systems. IEEE Micro **28**(3) (May 2008) 54–66