Automatic Placement of Tasks to NUMA Nodes in Iterative Applications

Jiri Dokulil
Faculty of Computer Science
University of Vienna
Vienna, Austria
jiri.dokulil@univie.ac.at

Siegfried Benkner
Faculty of Computer Science
University of Vienna
Vienna, Austria
siegfried.benkner@univie.ac.at

***Abstract*—Manycore architectures with non-uniform memory access (NUMA) are commonly used for high-performance computing. On these systems, the placement of data and computation to the NUMA nodes has very significant impact on performance, especially with memory-bound applications. This placement can usually be defined by the programmer, but it is generally desirable to automate the placement to simplify the programmer's job, improve portability, and make the code more future-proof. Task-based runtime systems already assume a fair degree of responsibility for task placement, so it is only natural to involve them in mapping work and data to NUMA nodes. In this work, we propose a solution where the runtime system first performs a profiling run of the application and measures various performance characteristics. Then, the data collected by the profiling run is used by a stand-alone analyzer to create a plan for placing tasks to NUMA nodes so that the tasks are close to the data that they most rely on. This plan is then used by the runtime system to execute the application more efficiently. We focus on iterative applications, where the same patterns of tasks are being repeated. We identify these patterns and use them to create a plan that works for any number of iterations, not just the one that used when the application was observed. In our experiments, which were performed on modern manycore systems (Intel Skylake and AMD Zen) with 4 and 8 NUMA nodes, the proposed automated placement can either match or come close (<10%) to a hand-tuned placement.**

***Index Terms*—parallel tasks, NUMA, task scheduling**

I. Introduction

Non-uniform memory access (NUMA) architectures are a common way to build large computers. NUMA is not a design objective, but a necessity that arises from the design of multi-socket servers and/or certain CPU architectures. Ideally, every core should be able to access any part of the memory via a high-bandwidth, low latency link. However, this would require all cores to access the memory via the same memory controller, which would require placing the controller further away from the cores. Instead, each processor (or a part of a processor) has its own memory controller. Each controller is connected to some of the machine's memory modules, but these modules are not shared among controllers. Therefore, if a CPU core needs to access memory connected to a memory controller other than the local one, it needs to send a request to the remote memory controller. This extra "hop" increases the latency of the memory access and the bandwidth may be limited as well. On the other hand, it provides faster access to the local memory compared to a uniform memory architecture.

Clearly, the performance of an application on a NUMA system depends on data placement, especially for memory bound applications. If we can ensure that most memory accesses go through the local memory controller, the performance would almost certainly be better than if most accesses have to be server by a remote controller. In a multi-threaded application, the threads are likely to share some of their data with other threads. Therefore, it may not be possible to ensure all memory accesses are local. In a way, a NUMA system is similar to a distributed system as there is the notion of "local" and "remote" data. Unlike distributed memory systems, the remote data can be accessed transparently the same way as local data. Also, the penalty for accessing remote data in NUMA is much smaller than in a distributed system. Still, the placement of data and computation on a NUMA system can have significant effect on performance. In our experiments, we've routinely seen difference of over 2x between a good and bad placement.

In programming models based on threads, it may be possible to obtain good results without explicitly considering the NUMA architecture. If there is good data locality, the operating system may be able to ensure that the data is moved to the memory that belongs to the local memory controller. When fine-grain tasks and work-stealing schedulers are used, the tasks may move between threads and subsequently also NUMA nodes. We cannot completely restrict this movement. Assuming tasks are created by other tasks, such restriction would result in only one NUMA node being used.

While it is possible to leave the placement of tasks to NUMA nodes to the application, an automated solution is clearly desirable for various reasons, like easing application development effort or improving portability. In this paper, we propose such solution. In our design, the runtime system first observes the running application, collecting performance metrics relevant to the way tasks access the memory. Then, an offline analyzer uses the collected metrics to define placement for data and tasks. This *plan* can be used in subsequent executions of the application to achieve better work and data placement, assuming the structure of the application remains unchanged. We focus our work on iterative codes, where the same pattern is repeated many times. By identifying

these patterns, we obtain two benefits. First, by aggregating performance data for equivalent tasks across all iterations we get more reliable measurements. Second, the plan can then be applied to application executions with different number of iterations, so that we are not restricted only to the case that was used to gather the metrics. Our work targets applications written using OCR [1], which is an open specification of a task-based runtime system. We use OCR-Vx, an open source implementation of the OCR specification [2]. OCR-Vx already had a NUMA-aware scheduler that we needed for our work and we only had to make several localized changes to collect the data that we need and make the runtime system use our plan. We have evaluated our solution on two many-core systems: a four socket server with Intel Xeon Scalable (Skylake) CPUs with four NUMA nodes and a two socket AMD EPYC (Zen) server with eight NUMA nodes.

II. Architecture

On the highest level, our solution consists of three steps:

A) profiling run of the application (once);
B) construction of the plan for mapping data blocks and tasks to NUMA nodes from the profiling results (once);
C) tuned execution of application using the plan (repeated).

The first and last steps are performed by the OCR-Vx runtime system. For this purpose, it can be configured (at compile time) to work in different modes. A different configuration is used for profiling and "real" execution. The application itself is unchanged. The second step is performed by a stand-alone analyzer tool. The profiling run stores the result in a set of CSV files. These are read by the analyzer which then outputs the plan as a set of (different) CSV files. Finally, these files are read when the runtime system starts for the tuned execution.

For the profiling run, we use the fact that all data in an OCR application has to be stored in data blocks managed by the runtime system. Therefore, we can make a snapshot of the data and then undo all changes made by a task. This way, we can run a task multiple times in different settings to obtain a larger collection of performance data. To make the data as reliable as possible, the profiling run is done using only a single thread, although we do plan to move to a multi-threaded profiling run in the future.

When creating the plan from the data measured by the profiling run, it would not be practical to consider each task separately, as an OCR application is likely to generate a huge number of tasks (thousands, millions, or even more). We have decided to focus on iterative applications where the same pattern of tasks is repeated for each iteration. By identifying equivalent tasks across iterations, we can work with much smaller number of task equivalence classes (we will call them *supertasks*). Equivalent tasks are identified by looking for patterns in the way tasks are created. We assume that if two tasks are created the same way by equivalent tasks, they are also equivalent. A typical example would be a task T_1 that creates a clone of itself (T_2) that does the same work in the next iteration. Then T_1 and T_2 are equivalent. If both these tasks create two more tasks (T_1 creates A_1 and then B_1, while T_2 creates A_2 and then B_2), then A_1 and A_2 are equivalent as are B_1 and B_2.

In our implementation, we use *generalized task paths*, which are regular expressions that describe a path from the root task (each OCR application starts with just one) to any task, following the *created-by* relation. The star operator allows a single regular expression to describe multiple tasks, forming the generalized task path and defining a task equivalence class (all tasks that match the path are part the equivalence class). The following is an example of a task path:

```
(mainEdt, 0, creatorEdt):
(creatorEdt, 0, creatorEdt)*:
(creatorEdt, 1, workerEdt)
```

In this path the, mainEdt (the root task) creates a creatorEdt (the first line). The zero means that the creatorEdt is the first task created by mainEdt. The creatorEdt then creates another creatorEdt. This step may be repeated any number of times, corresponding to different iterations. Finally, any of the creatorEdt tasks may create workerEdt (the number 1 signifies it is the second task created by creatorEdt). So, the path actually identifies a worker task in any of the application iterations.

A major assumption in our work is that equivalent tasks access data (almost) the same way and that they perform (almost) the same computation. This way, we can synthesize data from profiling and generalize the plan to any number of iterations. In the profiling run, we run each task multiple times but vary the placement of data blocks read by the task. This is likely to change the execution time of the task. If the task relies heavily on fast access to some data block and this data block is placed far away (different NUMA node with slower access) the execution time increases.

When we have the data from the profiling run, we combine measurements for equivalent tasks and compute affinities between task equivalence classes and data blocks. A high affinity tells us that a certain class of tasks would benefit from being placed close to a certain data block. The data blocks are first placed to NUMA nodes according to a pre-defined data distribution pattern and then task placement is determined for task equivalence classes from the data block placement and the affinities. Note that we also work with data block classes, not individual data blocks, using generalized paths as well.

The plan is then used by the tuned execution to place the tasks and data blocks to the NUMA nodes specified in the plan. As a NUMA node generally contains multiple CPU cores, there are multiple execution threads per one NUMA node. The standard task stealing algorithm is used to schedule the tasks inside a NUMA node, but no tasks are stolen across NUMA node boundaries.

III. Experimental evaluation

Our experiments were performed on two different servers running Linux. One server contains four Intel Xeon Scalable Gold 6138 processors (Skylake architecture, 20 cores, AVX-512 support, 32 KB L1 data cache per core, 1 MB L2 cache

per core and 27.5 MB last level cache in total). There are 24 memory modules installed in the machine to fully utilize the 6 memory channels per CPU. Each module has 8 GB capacity. The total number of physical cores is 80. There is one NUMA node for each processor, meaning the server has the total of four NUMA nodes, each with 20 cores.

The other server contains two AMD EPYC 7501 processors (Zen architecture, 32 cores, 32 KB L1 data cache per core, 512 KB L2 cache per core and 64 MB last level cache in total). The machines has 16 of the 8 GB memory modules, to match the 8 memory channels provided by each of the two processors. The total number of physical cores is 64. Since each processor is internally divided into four NUMA nodes (16 cores each), there are eight NUMA nodes in total.

A. Applications

We have evaluated two different applications. These are the two iterative, stencil OCR applications that we are aware of. The *stencil2d* application was obtain from the now defunct application repository that accompanied the OCR implementation created by Intel and Rice University [3]. The *seismic* application is distributed with the OCR-Vx implementation.

Stencil2d: The stencil2d application is a proxy application that applies a stencil operation to a grid. It is possible to specify the size of the grid, the number of iterations, and the number of rectangular sub-grids that the whole grid is split into. The sub-grids are then processed in parallel, so there is one strand of task for each sub-grid. The application prints out the number of MFLOPS it achieved.

Seismic: The seismic application performs a simple iterative simulation of seismic wave propagation. It is also a 2D stencil, but it uses different data partitioning and a different way of expressing the work as tasks. The data is split into N horizontal blocks and each block is internally sub-divided into M horizontal blocks, each processed by a different strands. The application prints the execution time of the computation.

B. Tested configurations

The OCR-Vx runtime system can be configured in several ways. Apart from the single-threaded scheduler used in the profiling runs, there are two different schedulers. It is possible to use the task scheduler provided by the Intel Threading Building Blocks [4]. This is a highly efficient task-stealing scheduler, but it is NUMA-oblivious. When choosing a victim thread for task stealing, NUMA is not considered. The other option is a custom NUMA-aware scheduler. Within a single NUMA node, it performs task stealing like the TBB scheduler, but it does not steal tasks across NUMA boundaries. Even though there is also a scheduler that can steal tasks across NUMA boundaries, we have not included it in the results presented here, as it was not able to provide competitive performance. It does not fully use the benefits of NUMA locality and it is not as efficient and scalable as the TBB scheduler.

The TBB scheduler cannot process the plan generated by the analyzer, leaving us with three options: TBB scheduler, NUMA scheduler using application-specified hints (hand-tuned variant), NUMA scheduler using the plan. We will denote these as TBB, HINT, and PLAN in the following text.

Another configuration option is the memory allocator. Again, there are two options applicable to our case. It is possible to use the scalable allocator provided by TBB. The TBB allocator maintains a per-thread memory pool to avoid the bottleneck imposed by a centralized allocator. The NUMA node is selected by the first-touch strategy employed by the operating system. The second alternative is a custom OCR-Vx NUMA-aware allocator. It uses the `hwloc` library to allocate memory on the selected NUMA node. Due to the limitations of OCR-Vx, it is not possible to combine the NUMA allocator and TBB scheduler. The other three combinations are all allowed. In the following, we will use TBB and NUMA to refer to the two allocators.

C. Experiment results

All results presented in this paper are averages of 10 executions. As no meaningful comparison can be made between the two applications, we present the results using the metric preferred by the individual applications: MFLOPS for stencil2d and seconds for seismic. The results for both applications are shown in Table I.

1) Stencil2d: The stencil application is always configured to use as many sub-grids as there are cores (80 on Intel, 64 on AMD). The data size is 1024×1024 during profiling and 10240×10240 during experiments. For profiling, we only executed 4 iterations, as even such small number turned out to be sufficient. The tuned execution performs 1000 iterations. On the Intel server, the best performance is obtained by using the NUMA scheduler with our plan and the NUMA allocator. The speedup over application-provided hints on Intel is only 1.9% but it is statistically significant ($p < 0.05$). On the other hand, on the AMD server, the hand-tuned version is almost 7% faster (also statistically significant). This might be due to the more complex structure of the NUMA nodes on the AMD server, where (unlike the Intel server) the communication time between NUMA nodes is different for different pairs of NUMA nodes. In all cases, the hand-tuned and our solution are much faster (more than 2x) than the TBB scheduler.

2) Seismic: The seismic was configured to split the data into as many blocks as there are NUMA nodes and use twice as many strands to process each block as there are cores in a NUMA node (this is a recommended usage of seismic). On Intel, this means 4*40 on Intel and 8*16 on AMD. The data size used for profiling was 1600×1600 and 16000×16000 for measurements. The number of iterations was 10 for the profiling run and 100 during experiments. The results are more varied than in the case of stencil2d. The application depends heavily on memory bandwidth, so the performance penalty for bad NUMA placement is extremely high, therefore the very poor performance of TBB scheduler, which is likely to move tasks between NUMA nodes. A hand-written TBB application would probably be able to perform better, the way the OCR runtime systems submits tasks to the TBB scheduler might

TABLE I
PERFORMANCE OF THE STENCIL2D AND SEISMIC APPLICATIONS

machine	configuration scheduler/allocator	stencil2d MFLOPS	stencil2d speedup over TBB/TBB	seismic time (s)	seismic speedup over TBB/TBB
Intel	TBB / TBB	47403	1.000	30.896	1.000
	HINT / TBB	127266	2.685	8.560	3.609
	HINT / NUMA	134148	2.830	8.693	3.554
	PLAN / TBB	128163	2.704	8.512	3.629
	PLAN / NUMA	136639	2.883	8.618	3.585
AMD	TBB/TBB	50758	1.000	26.294	1.000
	HINT/TBB	116424	2.294	10.114	2.600
	HINT/NUMA	118204	2.329	10.107	2.602
	PLAN/TBB	110719	2.181	10.124	2.597
	PLAN/NUMA	110808	2.183	10.086	2.607

increase their migration between threads and therefore also between NUMA nodes. On Intel, the performance advantage of our solution is small but statistically significant. On AMD, the improvement with NUMA allocator is even smaller (0.2%) and the statistical significance is borderline ($p = 0.06$). When the TBB allocator is used, our solution is slower than the hand-tuned one by 0.1%, but this result is not significant ($p = 0.39$).

Overall, our solution provides very good results for the two stencil codes, either outperforming the hand-tuned versions or matching their performance. In both cases, there are much faster than the NUMA-oblivious TBB scheduler.

IV. RELATED WORK

While the idea of using tasks to manage the computation is already quite common, using relocatable data blocks to let the runtime system automatically manage data is not so widespread, especially in share-memory systems. For example, Intel Threading Building Blocks (TBB) uses tasks but the data is managed like in a normal shared-memory application using pointers that are not exposed to the runtime system. A task can access any valid address at any time. It would still be possible to make experiments, by moving tasks between NUMA nodes and observing their performance.

Other runtime systems often have some NUMA support, allowing workers to be placed to NUMA nodes. StarPU [5], OmpSs [6], and ParalleX [7] are examples of such systems, where task placement can be made NUMA-aware, but only to limit moving tasks between NUMA nodes, not to actually place tasks based on where their data is placed.

V. CONCLUSION AND FUTURE WORK

We have designed and implemented a system which automatically generates a plan from placing tasks and data blocks to NUMA nodes in applications where the same patterns are repeated across iterations. When using the generated plan, the application performance is comparable to a hand-tuned placement. Rather than placing individual tasks, we work with groups of tasks that serve the same purpose in different iterations, allowing us to use a profiling run with a small number of iterations to create a plan for execution with any number of iterations. The tasks and iterations are identified automatically using generalized task creation paths, so there is no need for the application developer to expose this information explicitly.

Our architecture can be seen as a framework, where the different components can be replaced with more sophisticated or more specialized alternatives. For example, the supertasks identification is a form of clustering, so rather than using task creation paths, we could use statistical analysis or even machine learning. As we have already mentioned, it would be very interesting to merge task and data placement steps, so that we could use affinities between tasks and data blocks to place both tasks and data blocks using optimization techniques from graph theory on the affinity graph. It would even be possible to use radically different approaches, like genetic programming. Another interesting problem are applications that work in multiple phases, where each phase requires different data placement. This would have to be automatically identified, adding a new step to the beginning of our workflow. It might also be possible to use hints provided by the application to speed up and improve profiling and analysis.

Acknowledgment The work was supported in part by the Austrian Science Fund (FWF) project P 29783 Dynamic Runtime System for Future Parallel Architectures.

REFERENCES

[1] T. Mattson and R. Cledat, Eds., *The Open Community Runtime Interface*, April 2016, https://www.univie.ac.at/ocr-vx/doc/ocr-v1.1.0.pdf.

[2] J. Dokulil, M. Sandrieser, and S. Benkner, "OCR-Vx - an alternative implementation of the Open Community Runtime," in *International Workshop on Runtime Systems for Extreme Scale Programming Models and Architectures, in conjunction with SC15. Austin, Texas*, 2015.

[3] T. G. Mattson *et al.*, "The Open Community Runtime: A runtime system for extreme scale computing," in *2016 IEEE High Performance Extreme Computing Conference (HPEC)*, 2016, pp. 1–7.

[4] A. Kukanov and M. J. Voss, "The foundations for scalable multi-core software in Intel Threading Building Blocks," *Intel Technology Journal*, vol. 11, no. 04, pp. 309–322, 2007.

[5] C. Augonnet, S. Thibault, R. Namyst, and P.-A. Wacrenier, "StarPU: A Unified Platform for Task Scheduling on Heterogeneous Multicore Architectures," *Concurrency and Computation: Practice and Experience; Euro-Par 2009*, vol. 23, pp. 187–198, 2011.

[6] J. Bueno, J. Planas, A. Duran, R. Badia, X. Martorell, E. Ayguade, and J. Labarta, "Productive programming of GPU clusters with OmpSs," in *IPDPS 2012 Parallel Distributed Processing Symposium*, 2012.

[7] K. Hartmut, M. Brodowicz, and T. Sterling, "Parallex an advanced parallel execution model for scaling-impaired applications," in *Proceedings of the 2009 International Conference on Parallel Processing Workshops (ICPPW '09)*, 2009, pp. 94–401.