NUMA-aware CPU core allocation in cooperating dynamic 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*—Building large scientific applications by composing multiple smaller applications is one of current research directions. If the individual component applications are executed together on one compute node, we need to allocate resources to the components. While the operating system is already capable of doing this, it might be possible to get higher efficiency with a more specialized solution. Our goal is to describe opportunities and challenges faced by anyone designing such a system. We look into the specific case where a dynamic runtime system (or multiple such runtime systems) are used by the applications, since we believe the fundamental design of these runtime systems makes them especially suitable for the role. The ideas described in this paper are based on our prior experience in building such a runtime system and our early experiments with cooperating applications. We will focus on CPU core allocation and point out the importance of making non-uniform memory access architectures a prime consideration in such work.**

I. Introduction

There are many ways in which we can deal with the increasing complexity of today's scientific applications. One interesting approach is to build a larger, more complex application out of multiple simpler applications. These applications might already exist, but combining them might be rather difficult, especially if we would like to build a single monolithic application, where everything is compiled and executed together.

The obvious alternative is to keep the applications separate, but allow them to share data, thus enabling them to work on the same problem. This way, we avoid the compatibility issues. However, there are still problems that need to be dealt with. First, we need to make sure that the data is exchanged correctly, at the right time, and that the progress of the individual applications is coordinated, so that we get the correct results. Second, we also need to make sure that the applications "play nice" with each other. While it would be possible to completely isolate them and run different applications on different compute nodes or virtual machines, it might be beneficial to run them on the same nodes. A tighter integration might allow more efficient data exchange (via shared memory) and synchronization, but also improve overall efficiency. If one application cannot use some resources at a point in time, we might be able to allocate them to another application, which can use them.

We will focus on CPU cores as the resources, but similar things could be done with accelerators if the applications use them. We will also consider memory as a factor in performance, but assume that there is enough memory available on the node to run all the components. To explore what might be possible, we will look at task-based dynamic runtime systems [1], [2] since applications built using such runtime systems are especially suitable for our scenario.

The main idea behind these systems is to use fine-grained tasks to express works, rather than threads. The tasks might potentially be executed in parallel and their synchronization is controlled by setting up dependencies between the tasks. By decoupling the work (tasks) from the processing units (CPU cores), these runtime systems get much more flexibility. Furthermore, most task-based runtime systems can dynamically adjust the number of worker threads used to run the tasks, or they could be extended with such capability with relative ease. Usually, there is a one-to-one mapping between worker threads and CPU cores, so changing the number of worker threads effectively changes the number of CPU cores that the application uses.

A side-effect of this is that these runtime systems can also easily move work between CPU cores, either by moving the worker threads or by stopping threads that use the cores that should become idle and starting new threads on the target cores. This is especially useful in the scenario of multiple cooperating applications, where the CPU cores of the node can be dynamically partitioned and allocated to the individual applications.

In the following text, we will further explore this topic, explain our previous work in the area and future plans. Based on our experience building and using the OCR-Vx [3] runtime system, we will make a case for NUMA layout of the node to be a prime consideration. Dealing with NUMA is already important for scientific applications and we will explain why we believe it is also very important when allocating CPU cores to multiple cooperating applications.

The rest of the paper is organized as follows. In the next section, we will further discuss how cooperating applications can benefit from task-based runtime systems, starting with brief overview of related work. Section III explains why we believe NUMA to be a key consideration in these scenarios.

Sections IV and V explain integration with codes that don't use tasks and put the work in context within distributed execution (MPI). The last section concludes the paper and discusses future work.

II. Task-based runtime systems and cooperating applications

There are already multiple established task-based runtime systems. The Intel Threading Building Blocks (TBB) library [2] was one of the first successful examples. Other systems include StarPU [4], OmpSs [5], Habanero [6] (with support for Java and C), HPX [7], or PaRSEC [8]. Even OpenMP has support for tasks (since OpenMP 3.0). In our work, we have mostly used the Open Community Runtime (OCR, [1]), which is an open specification of a task based runtime. We have created OCR-Vx, our own implementation of the specification. The OCR is well suited for research, since it has a relatively solid specification [9], small API, and it also moves the application data under the control of the runtime systems, in a fashion similar to how the execution of work is delegated to the runtime via tasks.

At the moment, most of the runtime systems assume that they have all node resources at their disposal. Some of the systems look at the core affinity set for the process and modify the number of threads accordingly. It is also possible to limit the resources used by the applications using Linux control groups (cgroups), since these are enforced by the operating system and the applications is therefore forced to follow them. However, only few of the parameters can be changed dynamically at runtime, not enough for our use. TBB is one of the few libraries built with a strong assumption that it is not running in isolation and the authors made a conscious effort to make it "play nice", automatically stopping unneeded threads. There is even a way to dynamically control the number of threads, although it is not exposed in an easy-to-use, well-documented way.

Still, at least some of the runtime systems can dynamically adjust the number of worker threads. We have extended OCR-Vx with this capability, specifically to support coordinated execution of multiple applications [10]. We used a simple producer-consumer scenario, where one application produces one data item per iteration and another application consumes one such item per iteration. Each iteration consists internally of multiple tasks that can be executed in parallel. We have used a dedicated agent process to coordinate their execution, dynamically adjusting the number of threads in both applications to keep them aligned, so that the producer is only ahead by a small number of iterations. The architecture is shown in Figure 1.

Each application starts with as many number of threads as there are CPU cores, but some of these threads may be suspended (blocked), to free up the corresponding CPU core. There are three different options that the runtime system can be configured to use for selecting the worker threads to suspend:

1) *Total number of threads* – the runtime is instructed by the agent to use a specific number of threads. As long as there are more threads than the target number, threads that are not currently executing a task get blocked. The blocked threads are not selected explicitly by the runtime, but based on their (in)activity. A thread executing a short task is more likely to be blocked than a thread executing a long task, since by the time the thread with the long task finishes, a sufficient number of threads may have been blocked to reach the target number. This also ensures that the target number is reached as soon as possible without preempting tasks, which is not supported by OCR-Vx. If the target number of threads is raised, the required number of extra threads are unblocked almost immediately. These threads are selected randomly. The threads may be bound (using affinity) to individual cores, to all cores in a NUMA node or unbound.
2) *Individual cores* – the runtime is instructed by the agent to block specific threads, which are bound to individual CPU cores. A thread blocks as soon as it finishes running a task or almost immediately if it is idle. Unblocking of threads is also nearly immediate.
3) *Number of threads per NUMA node* – in this setup, the threads are bound to NUMA nodes, not individual cores. If a NUMA node contains 6 CPU cores, there

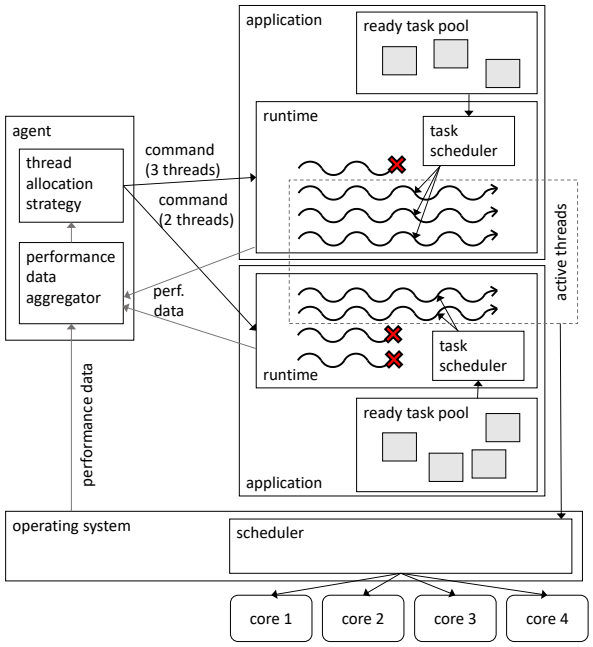

Fig. 1. The architecture of the system, with two running applications. The agent communicates with the runtime in both applications. It receives information about the execution from the runtimes (number of tasks executed, number of running threads, etc.) and it issues commands instructing the runtimes to use a specified number of threads. The threads over this limit are blocked. The task scheduler inside each of the runtimes schedules the ready tasks of the application to the available (not blocked) threads. All of the threads are scheduled to the CPU cores by the operating system. The agent also periodically queries the operating system to check the actual CPU load generated by the applications.

will be 6 threads bound to this NUMA node, but not to individual cores. The operating system is allowed to move the threads between the cores belonging to that NUMA node. Otherwise, this option behaves like the first option, but the target number of threads is specified for each NUMA node. For example, if there are two NUMA nodes with 6 cores each, it is possible for the agent to order the runtime to use 4 threads in the first NUMA node and 2 threads in the second NUMA node.

While we use the agent process to decide the number of threads to be used by the different runtime systems, it would also be possible to have the different runtime systems cooperatively come to an agreement.

The thread blocking functionality may not be directly offered by the other runtime systems. For example, the TBB API only allows the number of worker threads used by the scheduler to be set once when the scheduler is initialized. However, TBB has Resource Management Layer (RML), which can dynamically allocate threads to arenas – collections of worker threads used inside TBB. This could be used to adjust the total number of threads usable in an application, like option 1 offered by OCR-Vx. Furthermore, it is possible to bind some of the TBB threads to a NUMA node, so by binding all threads in an arena to a NUMA node and using RML to adjust the number of threads in the arenas, we should also be able to get something very similar to option 3 of OCR-Vx.

There is no fundamental reason why something similar would not be possible with the other runtime systems. However, there may be some limitations. For example, OpenMP is allowed to suspend execution of a task at some points. If the task is *tied*, it is guaranteed to eventually resume execution on the same thread. Removing this thread from the worker pool would prevent the task from executing. Still, this could be solved by not suspending tied tasks.

If multiple instances of a runtime system or even multiple instances of different runtime systems support one of the thread blocking options and they have a common mechanism to achieve a consensus on how the CPU cores can be allocated, we get a powerful tool for managing CPU cores. The way of reaching the consensus could for example be the agent process that we used in our architecture shown in Figure 1. A simple core allocation strategy would be to give each application a fair share of the cores, so that the total number of worker threads across all applications is equal to the total number of available CPU cores. Normally, each application would create and use as many worker threads as there are cores, leading to significant over-subscription. This forces the operating system to constantly switch between threads of the different applications, leading to extra overhead and also decreasing cache efficiency. Without over-subscription, most threads would be allowed to run on the same core for extended periods of time, improving cache efficiency thanks to better locality of execution.

On the other hand, some over-subscription might be beneficial. If some tasks are unable to fully utilize the available cores, for example by being blocked in I/O operations, it might be beneficial if there are other threads available that could be scheduled to such cores. With our architecture, we can still achieve such over-subscription, but we have better control of it, being able to decide how many extra threads we want exactly and which applications should get these threads.

However, there is one important caveat with these over-subscription avoiding techniques. Our earlier experiments [10] have shown that in most cases, the Linux operating system can do a very good job when scheduling the threads of such applications, so the benefits of the thread allocation techniques may not be as good as one would imagine. We have observed a clear benefit on storage thanks to the reduced size of intermediate data (data generated by the produced but not yet processed by the consumer) but only marginal (a few percent) improvement in performance. In some cases, there was no measurable improvement. With a larger number of applications, the benefits would most likely increase, but our setup did not allow for such an experiment.

Therefore, most of the benefits will not come from simply reducing the OS scheduling overhead by only using as many threads as necessary. In fact, if the threads are unable to fully utilize the CPU cores at all times, the performance might even be reduced. One example where we might get benefits is the aforementioned reduction in intermediate data size. But we believe it is still possible to get actual speedup thanks to better CPU utilization. For example, if the scaling of the applications is less than linear, we might get better efficiency by reducing the number of threads. Note that we are not assuming that the performance of that application actually degrades with more threads. The application's performance might increase with any extra thread, but the scaling is not linear. In this case, it might be better to limit the number of threads allocated to this application and assign the CPU cores to another application, which can make better use of them.

Another scenario where dynamic core allocation might be beneficial are cooperating applications that are more tightly integrated, not just running on the same nodes, exchanging data at few, clearly defined points. For example, one application might use the other application like a library, delegating a specific job to it whenever needed. In this case, quickly shifting resources to the "library" application when it is called could improve efficiency. Similarly, when the "library" finishes, we can quickly free up the CPU cores that were used to run it and move them back to the "main" application. Furthermore, with even tighter integration, we might be able to not just move the threads, but also make sure that the core that wrote the data (that should be processed by the "library") also starts processing the data inside the other application, enabling cache reuse.

This is one of the directions that we consider to be potentially very interesting and we plan to pursue it in the future. We aim for a tight integration, where we compose low-level components (provided by different application codes), with frequent interaction between the components across application boundaries. In such case, one might view the

whole composed application as one enormous task graph that spans multiple processes (we assume the different applications cannot be integrated into one code and then run as one process). However, we don't expect it to be possible to have a single do-it-all scheduler, which would be able to schedule the whole task graph. Different kinds of workloads might benefit from using a scheduler tailored for the specific kind of problems. But even at a more fundamental level, different task-based runtime systems have a very different way of dealing with synchronization (like dependencies, but also controlling access to shared data) and it might be very difficult or even impossible to handle all of them together by a shared scheduler. Instead, each code would use its own runtime system, with a scheduler tailored for the specific purpose. The coordination of the individual runtime systems and schedulers would happen on the level of resource arbitration. The runtime systems would agree on core allocation and quickly transfer cores when necessary. So, while existing less tightly integrated applications might benefit from dynamic resource management (assuming they use dynamic runtime systems), we expect the benefits to be more in allowing efficient tight integration of small components.

III. NUMA EFFECTS

Since we are considering running multiple applications on a single node, the node is likely to be *fat*, with a large number of cores and large memory. Such nodes are usually built as non-uniform memory access (NUMA) architectures, where the memory access time depends on the actual location of the data. As scientific applications tend to be memory bound, they need to take NUMA into account. We have seen that with OCR-Vx, it is possible to get very significant speed improvement with NUMA-aware codes over NUMA-oblivious alternatives [11]. We have performed our experiments on the Intel Knights Landing (KNL) processor, where the NUMA is optional and can be switched off. It was possible to get good performance from the NUMA-oblivious codes by switching the process to non-NUMA mode. But on most multi-socket servers, the NUMA is inherent to the hardware design and it is impossible to opt out[1]. We have performed the same experiments on a multi-socket server based on Intel Xeon CPUs[2] and the speed improvement over the NUMA-oblivious code is significant, even larger than on the KNL with enabled NUMA.

As a result, we should expect that most of the applications will be NUMA-aware, placing the data and tasks in a way that optimized the data access. Allocating cores to such applications by specifying the total number of worker threads could be very inefficient, unless the runtime systems used by these applications can make good decisions about which threads to block to reach the target number. The current implementation of OCR-Vx is not capable of that, but it would be possible to extend it to spread the blocked threads evenly across the NUMA nodes. But all runtime systems would have to make such decisions and also make sure that they are compatible. For example, we would not want all runtime systems to decide that when they get exactly the right number of threads to fully occupy one NUMA node that they will all use node 0.

We believe that in this case, it would be better to use the option 3 for selecting threads to block (the three options were explained in the previous section) and instruct the runtime systems how many threads to use on the different NUMA nodes. Consider a scenario with 4 applications, machine with 4 NUMA nodes and 8 CPU cores in each NUMA node. We could give each application two cores in each NUMA node, but also give all cores in one NUMA node to each application. There are many other ways to partition the machine, like giving one application 5 threads in all nodes and one thread per node to all the other applications.

In this example and further text, we work with two important assumption. First, each thread is bound to a NUMA node. Most operating systems do allow the code to specify thread affinity, listing which CPU cores the thread is allowed to use. We assume that the affinity for each thread is set to cores in one NUMA node, therefore the thread can run on any core of that NUMA node, but nowhere else. The second assumption is that there are at most as many threads bound to a NUMA node as there are CPU cores in that NUMA node. In other words, there is no over-subscription.

With these assumptions, we don't need to distinguish between threads and cores. Without over-subscription, the operating system can allow the threads to mostly run uninterrupted on the core they have first been assigned, without moving the threads around. In our experience, the Linux scheduler does a very good job in this regard. So, for performance reasons, we can assume that all threads are running and that each threads is bound to a core. The exceptions are rare enough to make this a sufficiently good approximation. In the following text, we use the terms threads and cores interchangeably, since they play the same role and it is not always easily possible to decide what properties would be better attributed to a core and which properties belong to the thread running on that core.

A. Model

To get some insight into different scenarios, we have created a simplified model of applications running on a NUMA machine and implemented a tool that runs simulations with this model. The applications can be configured to use different core allocations using option 3 (number of threads set for individual NUMA nodes). Since we need to model the core performance and memory access, we use the roofline model [12], which can give us performance estimates based on just these numbers: arithmetic intensity (AI) of an application (number of floating point operations per one byte transferred to from/memory), peak memory throughput (GB/s), and peak compute performance (GFLOPS). We work with several assumptions:

1) a single CPU core has the same peak GFLOPS for each application;

[1]It may be possible to enable node interleaving, which does hide the NUMA architecture, but this degrades performance of most applications and is not recommended if a NUMA-aware operating system is used.

[2]These results were obtained as part of the work on OCR on KNL [11], but not presented in the paper or any other publication.

2) for the purposes of computation, the CPU cores are completely independent (e.g., there is no DVFS);
3) each threads tries to access memory at the peak bandwidth based on the arithmetic intensity and peak GFLOPS of the core (e.g., a core with 10 GFLOPS running code with AI=2 would try to read $10/2 = 5$ GB/s);
4) memory bandwidth is shared by all cores in the same NUMA node;
5) the actual memory bandwidth is split to the cores so that each core can get at least its equal share of the total node bandwidth (with 40 GB/s per node and 8 cores this is $40/8 = 5$ GB/s) and the remainder is split proportionately to the attempted memory access (e.g., a code that would want to make twice as many memory operations above the baseline will end up getting twice as much of the remaining bandwidth).

Let's revisit our previous examples with 4 applications, this time assuming three of them are memory bound (AI=0.5) and one is compute bound (AI=10). The machine has 4 NUMA nodes, with 8 cores each, and each core capable of peak 10 GFLOPS. The memory bandwidth is 32 GB/s per NUMA node. The memory-bound applications would want to use 20 GB/s (10 GFLOPS of the core, divided by AI of 0.5). The compute bound application would only use 1 GB/s. We will model three different ways of allocating threads to these applications. A graphical overview of the possibilities is shown in Figure 2.

If we allocate 1 thread per NUMA node for each of the three memory-bound applications and 5 threads to the compute-bound application, the total desired bandwidth is 65 GB/s ($1 * 20$ for each memory bound and $5 * 1$ for the compute-bound code, therefore $3 * 1 * 20 + 1 * 5 * 1$). The available bandwidth is only 32 GB/s, and the baseline bandwidth per core is 4 GB/s (the total of 32GB/s divided to 8 cores). As the compute-bound application is asking for 1 GB/s per thread, it will get the total 5 GB/s required together by the 5 threads. The memory-bound applications are asking for 20GB/s per thread, so they get the baseline of 4 GB/s per thread. At this point, 17 GB/s has been allocated to the four applications ($3*4+1*5$), with 15 GB/s still remaining. We split this evenly among the three memory-bound applications, giving each of them 5 GB/s more to the total of 9 GB/s (4 GB/s baseline, 5 GB/s extra). In performance, this translates to 50 GFLOPS for the compute-bound code (5 threads, 10 GFLOPS each) and 4.5 GFLOPS for each of the memory-bound codes (9 GB/s and AI of 0.5). In total, we get 63.5 GFLOPS per NUMA node, for the total of 254 GFLOPS. This computation is also shown in a more structured form in Table I.

If we allocated two threads to each application on every NUMA node, we would get much less performance from the compute-bound code (20 GFLOPS per NUMA node) and only slightly more from the memory-bound codes (5 GFLOPS per NUMA node). Overall, this works out at 140 GFLOPS. The exact way of coming to this result is shown in Table II. If we gave one NUMA node to each application, we would get 128 GFLOPS in total (80 for the compute-bound code and 16 for each memory-bound code).

In the example above, the best and second best results are obtained by allocating the same number of threads to each application on every NUMA node. Assigning each application to its own NUMA node provided the worst performance. However, the applications that we used so far were perfectly adapted to NUMA, so they only read local (belonging to the same NUMA node) memory. To explore different applications, we need to further extend the model to handle reading data from other NUMA nodes. We support two kinds of applications: perfectly adapted to NUMA, like the two applications in the above example, and the worst case application, which stores all its data in a single NUMA node. The machine model now defines peak bandwidth between all pairs of NUMA nodes. The simulation has been modified to handle these in a way that is simple but captures to some degree experimental results that we have obtained using the STREAM benchmark [13] on a four socket server with Xeon Skylake processors: a memory first tries to serve requests from other NUMA nodes (up to the maximum bandwidth provided by the link to that remote node) and splits the remaining bandwidth as before.

With this model, we used two different types of memory-bound applications: three instances of a NUMA-perfect applications with AI=0.5, one instance of "NUMA-bad" application that stores all its data in one NUMA node and has AI=1. We have compared two different thread allocations. If we give 2 cores to each application in each NUMA node, the total performance is 138 GFLOPS. This situation is shown in Figure 3. By giving each application all threads in a NUMA node (and ensuring the NUMA-bad code is on the right node), we get 150 GFLOPS. This is the same as the case c) in Figure 2. So, the result is the opposite from the previous example, where dedicating a whole NUMA node to an application was the slowest option.

This shows us that when allocation cores to applications, we need to be aware of the NUMA architecture and also of the way memory is used by the application. Preferably, there should be a way to not only figure out the access patterns, but also to influence where the application stores its data. In the ideal case, the application should be able to move the data to a different NUMA node. This would easily be possible in OCR, where the runtime system is also in charge of managing the data, but it might be very difficult in applications based on TBB, where the runtime is oblivious to the application data.

B. Experimental evaluation

The model presented above is a relatively rough approximation of the complex behavior of a NUMA system. To see if it somewhat corresponds to the real world, we have implemented a simple synthetic benchmark that can behave like the applications used to evaluate the model. We have tested it on a NUMA server with 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

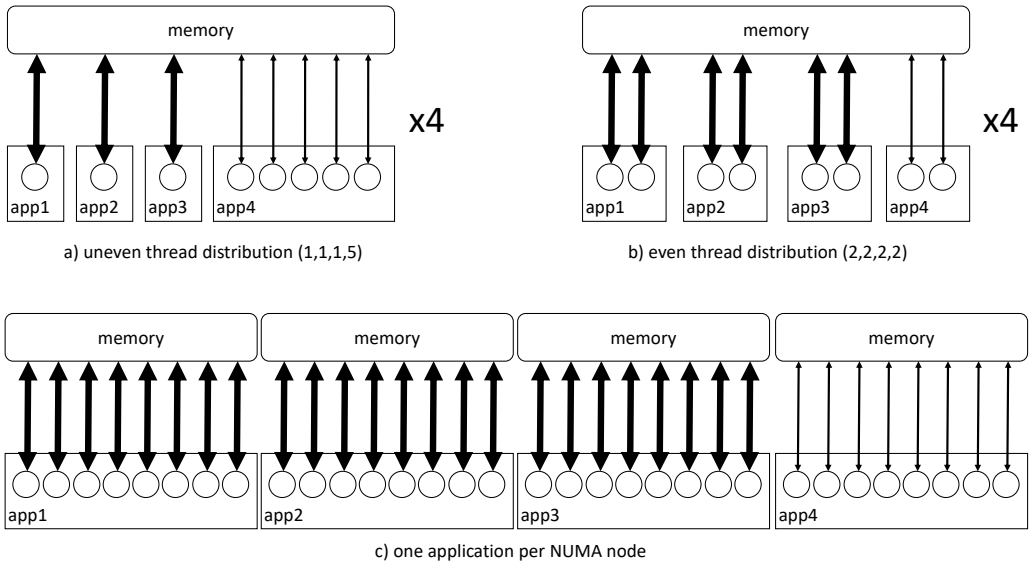

Fig. 2. The three different scenarios used to run the applications. The app1, app2, and app3 boxes represent the three memory-bound applications, while app4 is the compute-bound application. Circles represent threads. The two figures at the top show situation for a single NUMA node, so each is actually repeated 4 times in the actual mode. The figure at the bottom shows all four NUMA nodes.

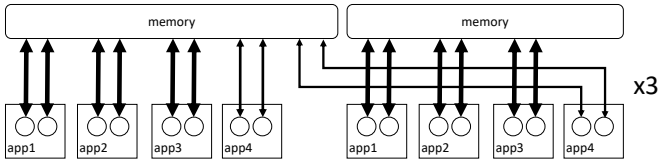

Fig. 3. The cross-node configuration of the case with the NUMA-bad application (app4). The part on the right is repeated three times in the model, with app4 always accessing memory of the node on the left.

memory modules installed in the machine to fully utilize the 6 memory channels per CPU. Each module has 8 GB capacity.

While it is easy to select the performance parameters (the peak performance, memory bandwidth and AI) to use in the model, it is difficult to replicate the same performance characteristics on real hardware and software. While the theoretical peaks of the machine can be derived from available information, the actual behavior of an application is likely to be far from those, without extensive tuning effort. Instead, we have only been able to make our best effort (with the available resources) to make the application work as well as possible and then estimate the parameters of the machine from the measured performance of the application.

We have configured the benchmark to match the even thread allocation scenario shown in Figure 2 b), tuned the AI of the applications to obtain meaningful results, and estimated the hardware's performance parameters from this case. The performance is consistent with 100GB/s memory bandwidth and 0.29 peak GFLOPS per thread. Note that there are 20 cores per NUMA node on our server, not 8 like in the model example. We have set up the model with these parameters and arithmetic intensities (1/32 for memory-bound, 1 for compute-bound, and 1/16 for the NUMA-bad application).

Then we have evaluated all five scenarios in the model and executed the corresponding benchmarks on the real hardware. The results are shown in Table III. For the first three cases, where we use three compute-bound applications and one memory-bound application with different thread assignment, the results are a good match. Of course, keep in mind that the second scenario is how the model was configured, so it needs to match. But also the first case is very close. As for the fourth and fifth case, where we evaluate combination of NUMA-perfect and NUMA-bad applications, the results are somewhat further away, but they are still reasonable approximations. Our model overestimates the performance by around 5% in both cases, so it gives good relative performance comparison, even though the absolute values are not fully accurate.

IV. Non-worker threads

Ideally, we would want the computation to be only done by the worker threads, which we can control. This would usually mean putting all the work into tasks. This is the way OCR is designed, but most runtime systems do not have this requirement. For example, TBB preserves the usual structure of the application with a main thread. This main thread may invoke parallel algorithms that are then processed by the worker threads. While the algorithm is running and the main thread is waiting for the result, the main thread might also be used by TBB to run tasks. Also, because doing I/O usually requires blocking calls to be made, extra threads may be created by the application to do the I/O, since blocking inside a TBB task is not advised.

As a result, we might get threads that are doing work, but are not controlled by the task-based runtime system. If such a thread is only submitting work to the runtime system or it

	memory-bound	compute-bound
arithmetic intensity (AI)	0.5	10
number of instances	3	1
threads per NUMA node	1	5
peak memory bandwidth per thread (peak GFLOPS / AI)	$10/0.5 = 20$	$10/10 = 1$
peak memory bandwidth per instance (per-thread * #threads)	$20 * 1 = 20$	$1 * 5 = 5$
total memory bandwidth of all instances (per-instance * #instances)	$20 * 3 = 60$	$5 * 1 = 5$
total required bandwidth	$60 + 5 = 65$	
baseline GB/s per thread (total GB/s #threads)	$32/8 = 4$	
allocated baseline per thread (min(peak,baseline))	$min(20, 4) = 4$	$min(1, 4) = 1$
allocated node GB/s ($\sum$ #apps * #threads * GB/s per thread)	$3 * 1 * 4 + 1 * 5 * 1 = 17$	
remaining node GB/s	$32 - 17 = 15$	
still required GB/s per thread (peak - allocated)	$20 - 4 = 16$	$1 - 1 = 0$
still required GB/s	$3 * 1 * 16 + 1 * 5 * 0 = 48$	
remainder given to a thread (remaining node GB/s / unsatisfied threads)	$15/(3 * 1) = 5$	
total allocated to each thread (baseline + split remainder)	$4 + 5 = 9$	$1 + 0 = 1$
GFLOPS per thread (allocated GB/s * AI)	$9 * 0.5 = 4.5$	$1 * 10 = 10$
GFLOPS per application (#threads * per-thread)	$1 * 4.5 = 4.5$	$5 * 10 = 50$
total GFLOPS per node	$3 * 4.5 + 1 * 50 = 63.5$	
total GFLOPS	$4 * 63.5 = 254$	

TABLE I
TWO APPLICATIONS MODELED ON A MACHINE WITH 4 NUMA NODES, 8 CORES PER NUMA NODE, PEAK 10 GFLOPS PER CORE, AND 40 GB/S BANDWIDTH PER NUMA NODE. UNEVEN THREAD ALLOCATION (1,1,1,5).

	memory-bound	compute-bound
arithmetic intensity (AI)	0.5	10
number of instances	3	1
threads per NUMA node	2	2
peak memory bandwidth per thread (peak GFLOPS / AI)	$10/0.5 = 20$	$10/10 = 1$
peak memory bandwidth per instance (per-thread * #threads)	$20 * 2 = 40$	$1 * 2 = 2$
total memory bandwidth of all instances (per-instance * #instances)	$40 * 3 = 120$	$2 * 1 = 2$
total required bandwidth	$120 + 2 = 122$	
baseline GB/s per thread (total GB/s #threads)	$32/8 = 4$	
allocated baseline per thread (min(peak,baseline))	$min(20, 4) = 4$	$min(1, 4) = 1$
allocated node GB/s ($\sum$ #apps * #threads * GB/s per thread)	$3 * 2 * 4 + 1 * 2 * 1 = 26$	
remaining node GB/s	$32 - 26 = 6$	
still required GB/s per thread (peak - allocated)	$20 - 4 = 16$	$1 - 1 = 0$
still required GB/s	$3 * 2 * 16 + 1 * 2 * 0 = 96$	
remainder given to a thread (remaining node GB/s / unsatisfied threads)	$6/(3 * 2) = 1$	
total allocated to each thread (baseline + split remainder)	$4 + 1 = 5$	$1 + 0 = 1$
GFLOPS per thread (allocated GB/s * AI)	$5 * 0.5 = 2.5$	$1 * 10 = 10$
GFLOPS per application (#threads * per-thread)	$2 * 2.5 = 5$	$2 * 10 = 20$
total GFLOPS per node	$3 * 5 + 1 * 20 = 35$	
total GFLOPS	$4 * 35 = 140$	

TABLE II
TWO APPLICATIONS MODELED ON A MACHINE WITH 4 NUMA NODES, 8 CORES PER NUMA NODE, PEAK 10 GFLOPS PER CORE, AND 40 GB/S BANDWIDTH PER NUMA NODE. EVEN THREAD ALLOCATION (2,2,2,2).

scenario	model GFLOPS	real GFLOPS
uneven thr. (1,1,1,17)	23.20	22.82
even thr. (5,5,5,5)	18.12	18.14
one app per node	15.18	15.28
NUMA-bad cross-node	13.98	13.25
NUMA-bad on-node	15.18	14.52

TABLE III
COMPARISON OF PERFORMANCE ESTIMATED BY OUR MODEL AND RESULTS OBTAINED BY A SYNTHETIC BENCHMARK ON REAL HARDWARE. THE "CROSS-NODE" PERFORMANCE REFERS TO THE CASE WHERE THE NUMA-OBLIVIOUS APPLICATION STORES ALL OF ITS DATA ON A DIFFERENT NUMA NODE, WHILE "ON-NODE" MEANS THAT THE DATA IS STORED ON THE CORRECT (LOCAL) NUMA-NODE.

is mostly blocked in I/O function calls, it is not a big issue from the load balancing point of view. But it might still be important when dealing with NUMA, as the I/O threads will most likely be reading and writing data that is also used for computation.

Then we have the threads that perform computation, but that are not worker threads of the runtime system. This might be the main thread used by TBB to run tasks, but also threads explicitly launched by the application to do some computation not using tasks. There could even be some applications that form a part of our one big composed application, but that are not based on a task-based runtime system. We might still be able to use thread affinities provided by the operating system to move such threads. Using priorities may also help in controlling how much compute times these threads actually get, but without significant changes to the code, we would probably not be able to fully stop such threads. Also, the applications might be written with the assumption that all their threads progress at a similar rate, leading to significant inefficiency if we break this assumption. One example of such code is the OpenMP parallel for loop with static scheduling.

If an agent process is used for resource arbitration, we also need to be careful about the way it affects the machine. If it is only required to occasionally perform quick (in the sense of CPU time) decisions, the operating system scheduler

would most likely be able to prevent any adverse effects these short interruptions (which make one CPU core temporarily unavailable for the computing applications). But if some sophisticated, CPU-intensive scheduling algorithm is used, we need to account for it. It could be run on a dedicated thread bound to a core not used for computation or even on a separate, dedicated machine.

V. Distributed environment

So far, we have always assumed that computation is performed on a single node. This is not the case for most large scientific applications. Usually, MPI is used to run them in a distributed fashion on a large number of compute nodes. If such application is composed of multiple cooperating but independent components, the nodes may be statically partitioned either by allocating nodes to the different components exclusively (e.g., component A gets 8 nodes, component B gets 8 different nodes) or splitting each node into several parts and giving each part to a component.

Our suggestion is to do a dynamic variant of the second option, by allowing the components to run on the same nodes, but dynamically shifting resources between them. This adds another layer of complexity to the problem, as the overall design of the MPI code usually assumes that the nodes provide comparable performance and allocate work either statically or dynamically, but with the assumption that the performance of the nodes is stable.

If the work is allocated statically to the nodes, we should attempt to provide some speedup on all nodes, favoring stability over maximal performance, so that it translates to an overall speedup. If the workload is being redistributed dynamically, we might be able to use more aggressive strategies. Still, the dynamic approach is more suitable for codes that are not tightly synchronized. If the code requires a barrier (or similar) after every iteration, the benefit of speeding up the iteration body on some of the nodes is rather limited. If the synchronization is loose, like an application that needs to perform a lot of independent tasks (many big data applications behave this way), most of the local speedup should translate to overall speedup.

VI. Conclusion and future work

In this paper, we have outlined why we consider the topic of dynamic CPU core allocation important and why task-based runtime systems might be a good fit for the job. The ability to suspend or move worker threads proved the kind of flexibility that would be difficult to achieve in a traditional thread-based design. On the other hand, properly dealing with NUMA systems is more challenging with dynamic systems, with the dynamic CPU allocation making it even more difficult. We do believe that it is a problem that can be solved and that it would provide higher efficiency within a compute node. Another step is translating the on-node speedup to improved overall performance even for large distributed applications. We also believe this to be possible, especially if the distributed application has some degree of flexibility on its own.

In the future, we plan to further develop these ideas and turn them into a workable solution, which would allow multiple runtime systems to effectively share on-node resources. In the first iteration, we plan to continue with our work on OCR-Vx, but also incorporate TBB, allowing TBB and OCR-Vx applications to cooperatively manage CPU cores. As we have already explained, we believe that the greatest potential might be in enabling low-level (fine granularity) composition of applications from codes that use different runtime systems.

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] 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.

[3] 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.

[4] 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.

[5] 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.

[6] V. Cavé, J. Zhao, J. Shirako, and V. Sarkar, "Habanero-java: The new adventures of old x10," in *Proceedings of the 9th International Conference on Principles and Practice of Programming in Java*, ser. PPPJ '11. New York, NY, USA: ACM, 2011, pp. 51–61.

[7] H. Kaiser, T. Heller, B. Adelstein-Lelbach, A. Serio, and D. Fey, "HPX - a task based programming model in a global address space," in *The 8th International Conference on Partitioned Global Address Space Programming Models (PGAS)*, 2014.

[8] G. Bosilca, A. Bouteiller, A. Danalis, M. Faverge, T. Herault, P. Lemariner, and J. Dongarra, "PaRSEC: Exploiting heterogeneity to enhance scalability," *IEEE Computing in Science and Engineering*, vol. 15, no. 6, pp. 36–45, 2013.

[9] J. Dokulil, "Consistency model for runtime objects in the Open Community Runtime," *The Journal of Supercomputing*, vol. 75, no. 5, pp. 2725–2760, May 2019.

[10] J. Dokulil and S. Benkner, "Adaptive scheduling of collocated applications using a task-based runtime system," in *30th International Symposium on Computer Architecture and High Performance Computing, SBAC-PAD 2018, Lyon, France, September 24-27, 2018*. IEEE, 2018, pp. 41–48.

[11] J. Dokulil, S. Benkner, and J. Yaghob, "The Open Community Runtime on the Intel Knights Landing architecture," in *Algorithms and Architectures for Parallel Processing - 17th International Conference, ICA3PP 2017, Helsinki, Finland, August 21-23, 2017, Proceedings*, S. Ibrahim, K. R. Choo, Z. Yan, and W. Pedrycz, Eds. Springer, 2017, pp. 801–813.

[12] G. Ofenbeck, R. Steinmann, V. Caparros, D. G. Spampinato, and M. Püschel, "Applying the roofline model," in *2014 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, March 2014, pp. 76–85.

[13] J. D. McCalpin, "Memory bandwidth and machine balance in current high performance computers," *IEEE Computer Society Technical Committee on Computer Architecture (TCCA) Newsletter*, pp. 19–25, Dec. 1995.