Adaptive scheduling of collocated applications using a task-based runtime system

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*—Task-based runtime systems are considered as one of the options for dealing with the challenges of upcoming parallel architectures. The greater flexibility of these runtime systems can also be used to dynamically adjust the resources allocated to the applications, adapting to the current load of the system and the progress of the applications. In our work, we have extended our implementation of the Open Community Runtime to support dynamic adjustment of execution threads. The runtimes communicate with an agent process, which collects performance data, computes thread allocation, and instructs the runtimes to make the required adjustments. We have tested our solution under different scenarios, focusing on producer-consumer applications, where the dynamic resource management was used to keep the applications in sync, improving the overall performance in some cases.**

***Keywords*-task-based parallelism, runtime systems, concurrent workloads, adaptive scheduling**

I. Introduction

Traditionally, the goal of job scheduling in HPC was to determine when a job should start and to select the right nodes to place its processes. In such solutions, the degree of parallelism is specified in the job description as the required number of processes. Either whole compute nodes are used as allocation unit, or it is assumed that a node has a certain number of slots (usually CPU cores) and that they can be used as independent allocation units. Later, co-scheduling approaches were developed, which consider multiple applications together, factoring in their resource usage and possible interactions. Such schedulers might decide to use a degree of parallelism that is below the maximum for a particular application, in order to (for example) reduce power usage or improve overall efficiency of executing a whole pack of applications. This is possible because performance scaling of many applications is below linear and their power usage above linear (w.r.t. the number of processes).

However, the process allocation in these cases tends to be static – the number of processes and their placement does not change. Big data applications often use more dynamic approaches. For example, Spark can dynamically request resources from the Mesos cluster manager, if it has a large number of tasks to execute. The resource allocation can change significantly during the lifetime of a job. Several programming models have been proposed as an alternative in HPC to provide such flexibility, which cannot be provided by MPI. Although there are also efforts to add such capability to MPI.

Task-based runtime systems are one of the interesting research directions. These systems decouple computation from threads/processes. The work is performed by tasks, which are scheduled by the runtime system. The runtime decides when and where a task should be executed. This may also include the ability of the runtime to move the task's code and data to a different compute node and execute it there. This gives the runtime the potential ability to automatically perform load-balancing or to allocate tasks to nodes where they can be executed most efficiently in a heterogeneous system. However, it also enables the runtime to adjust the amount of resources used on a node. By not scheduling tasks to a certain processor core, the runtime frees the core to be used by another application.

In this paper, we investigate how much flexibility this actually offers to the runtime in the terms of being able to dynamically move resources of a single node between applications. Using OCR-Vx [5], our implementation of the Open Community Runtime [13], we have performed several experiments executing multiple applications concurrently on a single node. The runtime was extended with the capability to quickly adjust the number of threads used for computation, when instructed to do so. An agent process is used to collect execution data from the applications and runtimes, make dynamic scheduling decisions, and instruct the runtimes how many threads they should use. The agent also tries to adapt the number of threads for applications that do not fully use CPU resources allocated to them.

The experiments show that these extensions introduce only a small overhead and that the number of threads can indeed be adjusted quickly, updating resource allocation with a frequency on the order of seconds. In a producer-consumer scenario, the dynamic thread management can be used to keep the producer and the consumer progressing at the same rate. It is possible even if the time per iteration is not constant, in which case the static methods would fail. Keeping the producer and the consumer in sync not only reduces the storage space needed for the intermediate data, but it may also improve the overall efficiency (and thus reducing the execution time) if the applications are not able to fully use all resources of the node

on their own.

Our main contributions are the following: (1) We have developed an architecture that allows threads to be allocated to multiple OCR applications dynamically at runtime. (2) We have designed and implemented several thread allocation strategies. The strategy for producer-consumer applications dynamically adapts to observed performance characteristics of the applications. (3) We have evaluated our design using a range of experiments using two applications – a seismic simulation code and an artificial producer-consumer workload.

The rest of the paper is organized as follows. Related work is discussed in Section II. Section III describes the architecture of our solution, followed by the description of the thread allocation strategies in Section IV. Experimental evaluation is covered by Section V. The final section concludes the paper and discusses future work.

II. Related work

Scheduling of MPI jobs is a widely studied field, both from theoretical and practical points of view [7]. There are both proprietary (e.g., IBM LSF or Altair PBS) and open-source (Slurm, OpenLava, various Sun Grid Engine forks, etc.) job schedulers available. Even though MPI does not support job preemption, the problem has been studied on the theoretical level [15]. However, the scheduled jobs have traditionally been viewed as independent. A job gets exclusive access to its allocated resources and does not interfere with other jobs.

Over time, new techniques were introduced to also deal with jobs where the degree of parallelism is not strictly prescribed by the job description, allowing tradeoffs to be made between performance of a single job and overall system efficiency [2], [3]. Power efficiency also became an important concern [9], leading to specific techniques [12], [14], [16]. These solutions are often referred to as co-scheduling – considering multiple jobs together while making scheduling decisions.

Our work differs from these solutions in two ways. First, using a task-based runtime system, we dynamically adjust the number of resources (CPU cores) assigned to the application, allowing us to make changes even on a sub-second scale. Second, we support fine grained dependences between applications, allowing a pair of producer-consumer applications to make progress at the same pace.

In some aspects, our work is closer to the way a scheduler in an operating system works. The scheduler schedules threads to CPU cores in a dynamic way, reacting to the current state of the threads (whether they are blocked, how long they have been allowed to work recently, etc.) and the system (overall CPU load, availability and placement of memory pages, etc.). Our scheduler works on a slightly larger time granularity, but more importantly, it uses additional information about the application to make scheduling decisions. At the moment, this only includes the progress of producer/consumer applications, but it is easily possible to use additional information provided either by the application or the runtime system.

Other task-based runtime systems mostly do not provide such functionality. StarPU [1] or OpenMP tasks do not consider other applications. The Intel Threading Building Blocks (TBB, [10]) has support for a resource management layer (RML), which is responsible for assigning worker threads to applications. However, the default RML does not consider other applications. There is an ongoing effort to implement such functionality in Charm++ [8].

Our work on the producer/consumer applications is closely related to the work being done on in situ visualization and analysis [6], [11]. We approach the problem from the perspective of a task-based runtime system, which gives us different tools that can be used to achieve the same goal. On the other hand, we would like to apply our work not just to in situ analysis, but the more general situation where multiple programs share the same resources. In situ analysis is a special case, that may be worth giving it special treatment.

III. System architecture

All work in an OCR application is done by tasks. The tasks are defined by the application code along with their dependences. Once all dependences of a task have been fulfilled, it is ready to run. A task scheduler is responsible for deciding when and where the tasks should be executed. The scheduler is part of the OCR runtime.

The OCR application is linked with the OCR runtime implementation. Each executing application runs as a single (separate) process and the instance of the runtime is a part of this process. The runtime uses multiple threads for computation and control. By default, there are as many worker threads as there are cores in the machine. To be more exact, it is the number of "logical cores". If hyper-threading or similar technology is enabled, it is a multiple of the number of physical cores. In the following text, let us assume that there are 32 logical cores in our example machine.

If multiple OCR applications are executed concurrently, each of them still uses 32 worker threads. The operating system's scheduler is then responsible for allocating these threads to logical cores. Usually, such scheduler tries to be fair, not favoring one application over the others. As a result, if all of the OCR applications can utilize the full parallelism (number of cores) available, they are assigned an equal share of CPU time. Depending on the application and specific circumstances, this may or may not be the right solution.

In a task-based parallel system, the application should be written to be independent of the actual number of worker threads. Ideally, an application should complete successfully even with a single thread. In OCR, the tasks are required to be non-blocking, which means that once a task is started, it should run to completion no matter what the other tasks may be doing. As a result, a correct OCR application should work correctly even if there is only one worker thread. On the other hand, such application may be able to use all 32 threads, but not necessarily. To do that, the runtime would need to have 32 runnable tasks at any given point in time. This may not always be the case. For example, an iterative code may provide enough tasks during an iteration to fully utilize all 32 threads, but if there is a barrier at the end of an iteration, there will be

a period when less than 32 tasks remain to be executed. For a period of time, the number of busy threads may be dropping from 32 to 1, until the last task is finished and the execution can continue after the barrier.

A. Dynamic worker thread adjustment

There are different ways a runtime can deal with this situation. The simplest approach is to keep all threads looking for work. This means that to the operating system the threads appear to be busy all the time and it keeps scheduling them to CPU cores. A slight improvement is to have the threads perform a yield if they cannot find any task to execute. This tells the operating system that even though the thread is still active, it does not need to proceed immediately and it would be better to use the CPU core to run another thread. If a worker thread cannot find a task to execute for a certain amount of time, it is often blocked in a way that forces the operating system to not schedule the thread at all. For example, Intel TBB [10] uses a semaphore to block the thread and then resumes it once work becomes available. In our OCR implementation, we use a simpler approach, where such thread sleeps for a pre-defined time interval (50ms at the moment). Having the thread periodically wake up adds overhead, but our experiments have shown that it is only minor.

So far, we have assumed that a runtime should use the maximal number of available worker threads and that these threads are only temporarily suspended (by sleeping) if there is not enough work available. We have extended the task scheduler to allow it to adjust the number of worker threads upon request. Because repeatedly creating and destroying threads would be wasteful, we do not change the actual number of worker threads, but only block those that are not needed. This reduces overhead and speeds up the adjustment, but it also means that the runtime cannot use more than the initial (created when the runtime is first started) number of threads (32 in our examples). Since tasks cannot be preempted in our runtime, a thread can only block when it is not executing a task.

After a worker thread finishes executing a task, it checks whether it should block. As this test happens after every task, it needs to be fast. Therefore, we use just two atomic counters: the number of running worker threads and the desired number of worker threads. The thread compares the two counters. If the desired number of threads is less than the actual number of threads, the thread decrements the number of running threads and blocks. In the actual implementation, care needs to be taken to avoid race conditions, as multiple threads can reach the decision point at the same time and the desired number of threads can also be changed at any time. If the desired number of threads is less than the number of running threads and the difference is N, the first N threads that reach the decision point (finish executing a task) block. So if all tasks take 100ms, we have to wait at most 100ms for the thread count to be adjusted.

A condition variable is used to block the threads. This way, the operating system can see that the threads are blocked and it does not schedule the threads. When the desired number of threads is increased, the condition variable is signaled and the threads wake up. Again, the threads need to check carefully the actual number of worker threads to decide whether they should actually start working or whether they need to block again. A disadvantage of this solution is the fact that all suspended threads are woken up once the desired number of threads is increased. This adds some overhead, but on the other hand makes the whole process simpler and the newly released threads can start working quickly.

In our example, the desired number of threads can be set to anything from 0 to 32. If 0 is specified, all worker threads block and the application makes no progress. Clearly, it should be possible to again increase the number of workers. The blocked worker threads cannot do that. To perform management tasks (not application tasks), our runtime uses management threads. These threads are suspended most of the time, allowing all CPU cores to be used for computation, but they may wake up upon an event or using a timer.

B. Reporting and management thread

One of the management threads is used as an interface between the OCR runtime and other processes. It periodically publishes some basic performance data, like the number of running worker threads or the number of executed tasks. The ZeroMQ library (ØMQ) is used for communication. The management thread publishes the performance data, but it also listens for commands. At the moment, there is just one command available – adjusting the number of worker threads. So, when the number of worker threads is set to 0, all of the worker threads are blocked, but if the management thread receives the command to increase the number of workers, it sets the atomic counter which stores the desired number of threads and signals the condition variable.

C. OCR agent

Each of the running OCR applications contains an instance of the OCR runtime. It would be possible to allow the runtimes to communicate in a peer-to-peer manner, but we have decided to use a centralized solution. A dedicated OCR agent process is started on the machine and all OCR runtimes report to the agent, sending it the performance data and listening for its commands. As the number of applications that run concurrently on a single machine would most likely be limited, we believe that the added benefit of having all relevant information in one place greatly outweigh any potential scalability issues.

The agent uses the information from the applications as an input to its thread allocation strategy and then it tells each application how many worker threads it has been assigned by the strategy. The agent may use different strategies to achieve different goals. These will be discussed in the next section.

The agent not only receives performance data from the OCR applications, it also uses services of the operating system to monitor the actual CPU usage of the applications. It is not uncommon for an application that is given – for example – 16 threads to only actually produce CPU load equivalent to 12

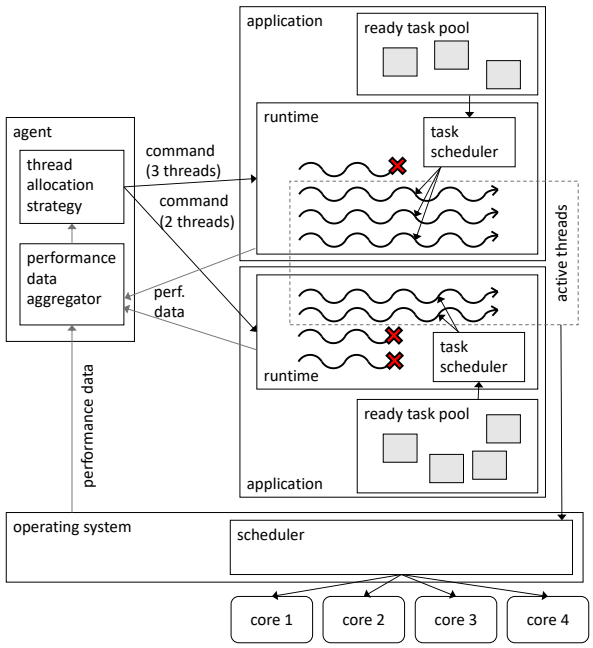


Fig. 1. The architecture of the system, with two running applications. The agent communicates with the runtime in both applications. It receives information about the execution from the runtimes (number of tasks executed, number of running threads, etc.) and it issues commands instructing the runtimes 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.

threads. This can be due to insufficient parallelism or because the code inside the tasks does not fully use the CPU core that it is running on. Some agent strategies use this information to perform a "boost". If we have two applications which only use 12 cores when they are given 16 threads and we want to run them at the same time to fully utilize our machine, giving 16 threads to each of them may not be sufficient. The total load would be equivalent to 24 fully loaded cores, leaving the equivalent of 8 cores idle. With oversubscription (boost), assigning 20 threads to each application would most likely increase their individual load, moving the total load up from 24 closer to the desired 32.

The overall architecture is shown in Figure 1.

IV. Agent thread allocation strategies

Given that there is no limitation on when and how can the number of threads be adjusted (besides the inability to go beyond a fixed maximum), there are virtually limitless possibilities for the design of the thread allocation strategy. The ones we have designed are just examples of what can be done and there is a lot of room for further improvement.

The first design decision was to have the strategies re-evaluate thread allocation at fixed time intervals. In our experiments (in Section V), we use 5 seconds, but we have successfully tested configurations where the interval is just 0.5 seconds. That means that the number of threads assigned to each runtime can change completely every 5 (or 0.5) seconds.

A. Even thread split strategy

The most obvious strategy is to split all threads evenly to all running applications. For well scaling OCR applications (the Seismic code used in our experiments is one example), this works well. Every application is started as soon as the job is submitted. The agent takes as many threads as there are compute cores in the machine and splits them evenly among all of the applications.

B. Simple unfair strategies

We have also tried two other configurations. In the first case, one long running application was given priority and received half of the cores, while the remaining (shorter) applications split the other half among themselves. The second strategy gave all cores to the application that started the last and none to the rest. The last submitted application was therefore allowed to finish first, then the second last, etc. This formed a stack of applications. The results for all these strategies were comparable. Keep in mind that this assumes that the applications scale almost perfectly and that they can quickly adapt to any number of threads (and use them efficiently). The different strategies demonstrate how processes can be dynamically prioritized according to different criteria.

C. Producer-consumer thread allocation strategy

Our main result is a strategy for producer-consumer applications, where one application generates a data item in every iteration and another application consumes one item (preserving the order in which they were generated) in every iteration. Our goal was to keep the two applications synchronized. Therefore, the strategy uses the number of items that have been produced but not yet consumed (the *queue length*) as the main input for its decisions. The target queue length is configured when the agent starts, although it would be possible to extend the interface to allow applications to control this parameter. The applications use an extended OCR interface to report the iteration they are in to their local runtime. The runtime then sends the information to the agent.

The producer-consumer thread allocation strategy compares the desired queue length with the actual queue length, which is the difference between the iterations reported by the producer and the consumer. Let us denote the target queue length as N. If the producer is ahead by more than $N/2$ items, the number of threads of the producer is set to 0 and the number of threads of the consumer is set to the maximum. If the consumer is ahead by more than $N/2$ items, the opposite happens (consumer gets 0, producer gets the maximum). If the difference is between $-N/2$ and $N/2$, each application gets a proportionate share of the threads, so that the sum is the maximal thread count of an application. As an example, if the machine has 32 cores and the difference is 0 (the queue length is exactly the desired target N), both the producer and the consumer get 16 threads.

D. Oversubscription (thread boost)

This works well if the producer and the consumer scale well. As this was not the case in our experiments, we have added a "boost" feature, which further increases the number of threads. Remember that it is possible to get any application up to the maximal number of threads (e.g., 32) irrespective of what was assigned to the other applications. As there will be more threads than cores, the operating system would have to step in and schedule the threads to the available cores, so it won't be possible for all threads to make constant progress. But if the threads are not kept suspended for too long (which the operating systems generally avoid), it is not a problem for task-based models, since the active threads pick up the work that was originally assigned to the now suspended threads.

We have experimented with a constant boost, which adds a fixed number of threads to all applications with non-zero allocation, and linear boost, which multiplies all thread allocations by a constant $c > 1$. Some of them worked well in certain cases, but they had to be tuned (selecting the right constants) for different workloads. We have therefore moved to an adaptive boost, which monitors the CPU usage of the process and adjusts the thread count based on these observations (a reactive model). When an application is given a certain number of threads and we measure the actual CPU usage of the process. For each possible thread count, we store the average CPU usage achieved. The original (unboosted) thread count is then treated as a target load and we select the number of threads that was historically able to achieve that CPU load. So, if both producer and consumer get 16 threads, but the producer is 100% efficient and the consumer only uses 75% of the allocated CPU power, they will actually get 16 and 22 threads respectively, because the historical data would show that with 21 threads the consumer only loads 15.75 cores.

This method is only a rough approximation. It assumes that the measured CPU is an accurate representation, but more importantly that the performance can be estimated purely by looking at the performance of the application in isolation, only considering the allocated number of threads. The scaling characteristics of the application may be much more complicated, the application can work in several distinct phases with different characteristics, it may be memory bound etc. While there is a lot of room for improvement, the strategy turned out to provide good performance in our experiments.

V. Experimental evaluation

The experiments were performed on a Linux server with two Intel Xeon X5680 CPUs (6 cores and 12 MB cache per CPU) and with 24 GB RAM. With hyper-threading enabled, the machine supports up to 24 hardware threads. For our experiments, we use two workloads. A seismic simulation and a pair of producer-consumer applications.

A. Seismic

Seismic is a 2D grid simulation of seismic wave propagation [4]. It is a simple iterative code, written from scratch as a native OCR application. It has been highly tuned and optimized, so it can run the computation in any degree of parallelism efficiently when properly configured. It can also be tweaked to increase computational complexity of the problem. The original computation is rather simple, making it mostly memory bound.

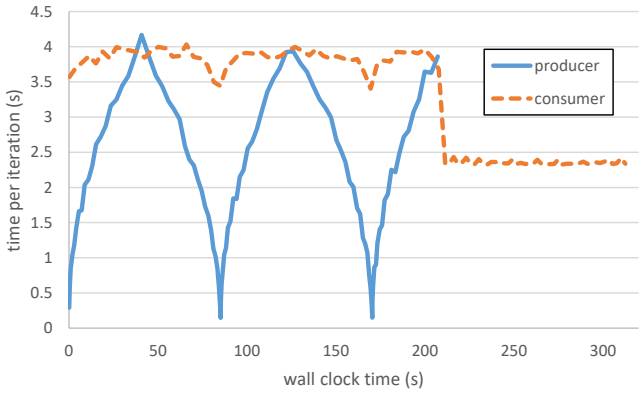
The Seismic workload is an example of a workload that scales very well and which can be easily adjusted dynamically. There are two fundamental reasons for this. First, it uses a large number of relatively small tasks (64k tasks in total, run time of a single task between 1ms and 16ms, depending on configuration and system load), so if the runtime is asked to decrease the number of threads, it can adapt quickly. Second, the tasks use fine-grained synchronization. There is no barrier at the end of an iteration. As a result, the runtime always has tasks available to use all available threads. Also, if the number of threads does not evenly divide the degree of parallelism inside an iteration, no time is wasted waiting at the barrier for the threads with more work to finish – the idle threads start working on the next iteration right away.

TABLE I
Performance of seismic (1 slow, 7 fast instances)

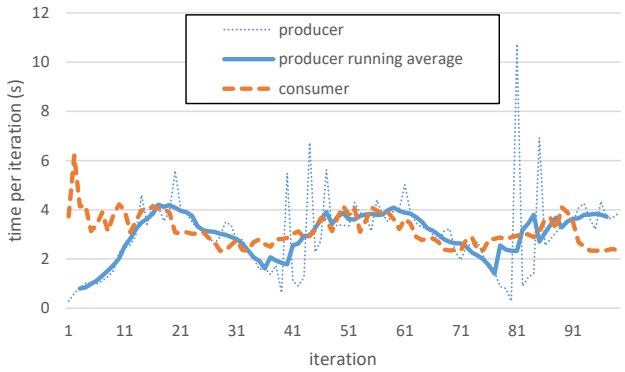
configuration	time (s)	speedup
sequential	89.194	1.000
concurrent, no agent	85.390	1.045
concurrent, with agent	86.092	1.036

In our experiments, we have used one "slow" instance of Seismic (around 9.3ms per task under ideal conditions) and 7 "fast" instances (1.1ms per task). The results are shown in Table I. The "sequential" case means that the 8 instances are executed one after the other. In the "concurrent" cases, the slow instance starts first and after 10 seconds, the 7 fast instances are started. Without the agent, the scheduling of the instances is left completely to the operating system. With the agent, it controls the thread allocation, trying to assign equal number of threads to all instances. Our machine has 24 cores and there are up to 8 instances, but as we have decided to only use thread counts that are multiples of 2, when all 8 instances are running, each instance gets either 2 or 4 threads, rather than 3. As a result, some of the fast instances finish more quickly. To be more specific, the fast instances that get 4 threads need 31.43 seconds on average, while the average execution time without the agent is 41.13 seconds, providing a 30.9% speedup for the fast instances that get 4 threads. On the other hand, the fast instances that only get two threads are somewhat slower than if no agent is used. This is to be expected, as the faster performance of some instances is not due to higher efficiency, but due to the fact that they are given more resources at the expense of others.

Overall, concurrent execution is more efficient than sequential, being 4.5% faster without the agent and 3.6% faster with the agent. This means that the agent does add a little overhead. On the other hand, it allows four of the fast instances to finish much faster (30.9% speedup) than if there was no agent. The main message of this experiment is that the number of



(a) Without the agent. The performance of the producer keeps slowing down and then speeding up, as it was designed to do. It runs in parallel with the consumer the whole time, so its performance only changes with the up-down cycle. The performance of the consumer improves significantly after the producer finishes. It does not double, but only increases by 1.63x, because the consumer is unable to use the whole machine. The x-axis is wall clock time, not iteration number, to better demonstrate the absence of synchronization between the producer and the consumer, even though it makes the up-down cycle appear to be non-linear (the blue "teeth" are curved, not straight).



(b) With the agent. The agent moves the resources (threads) between the producer and the consumer to balance their speed. As a result, their performance should be comparable. This is better seen when a running average of 7 iterations is shown for the producer, rather the per-iteration time. Initially, the producer is faster, because the agent is giving it extra resources to quickly get to the desired queue length. At the end, the consumer speeds up, because the producer is (on average) 8 iterations ahead and therefore finishes sooner, leaving the whole machine to the consumer.

Fig. 2. The time per iteration of the producer and the consumer, when the up-down mode is enabled in the producer.

threads used by the runtime can be adjusted dynamically by the agent, with only very minor drop in efficiency, but it gives the agent a way to significantly influence the performance of the individual instances of the runtime (and application).

B. Producer-consumer

The second workload is an artificial producer-consumer (P-C) workload. It consists of two OCR applications. Both producer and consumer work in iterations, producing or consuming a single item at a time. A configurable number of tasks is used to produce or consume a data item in parallel. The producer saves the data to a file. The consumer reads the file and processes it. If the file is not yet ready, the consumer blocks and waits. The computational complexity is defined by a parameter which defines how much computation a single task of producer/consumer does. Furthermore, the producer can be configured as up-down in order to simulate dynamic performance variability of applications. In this case, the complexity varies between iterations. For the first 20 iterations, it increases linearly from 0.1 to 4.1 of the baseline complexity. It does the reverse for the next 20 iterations (going from 4.1 to 0.1). This 40 iteration long cycle keeps repeating until the end of the producer's work. Figure 2(a) shown this graphically based on measured execution times.

Even if the up-down mode is not used, the producer may be faster or slower than the consumer. If it is slower, the consumer starves – there is not always enough work available to keep the consumer working. If the producer is faster than the consumer, the consumer is always busy, but the producer may move ahead of the consumer significantly. This means that the data in the queue between the producer and the consumer needs to be stored somewhere. If the data items are large and the queue is long, it may require a considerable amount of space. With the up-down producer, it is possible to encounter both situations – starving consumer and a long queue.

Even though the data items are stored as files, making the runtime and agent unaware of their existence, the applications use our extension of the OCR API to notify the runtime about their progress. The producer reports the number of iterations that it just finished and the consumer reports the number of iteration it is just about to start. This data is used by the thread allocation strategy to make its decisions.

We have used three different configurations for the relative speed of the producer and the consumer. The producer may be faster or slower than the consumer, but the producer and consumer can also be configured to require the same amount of work for each iteration. We have tested two variants of the fast producer experiment, with the up-down feature enabled and disabled. The results can be seen in Table II.

1) Fast producer with up-down: In the first experiment, we have used 240 tasks per iteration (both producer and consumer), 100 iterations, the up-down producer, and the agent was configured with the target queue size of 8. The producer was set up to be faster than the consumer, except for the slowest consumer iterations, where their performance (time per iteration) was nearly identical. This can be seen in Figure 2(a), where the solid blue line (producer) barely grows above the dashed orange line (consumer) in the first 55 iterations, where the producer and consumer run in parallel. The data was gathered without the agent, so the consumer speeds up once the (fast) producer finishes. With the agent (see Figure 2(b)), the agent controls the performance so that on average the time per iteration of the producer and the consumer is close. But the cost (in CPU cycles) of the producer's iterations is the same as without the agent, so the two and a half "mountains" still show up as "hills" when a similar is drawn from data

TABLE II
PERFORMANCE OF VARIOUS CONFIGURATIONS OF PRODUCER-CONSUMER

	fast producer, up-down		fast prod., no up-down		balanced		slow producer	
configuration	time (s)	speedup	time (s)	speedup	time (s)	speedup	time (s)	speedup
sequential	359.61	1.00	304.13	1.00	247.82	1.00	289.68	1.00
concurrent, no agent	317.41	1.13	274.90	1.11	199.38	1.24	272.84	1.06
concurrent, with agent	316.20	1.14	269.46	1.13	200.91	1.23	268.91	1.08

gathered when the same workload is executed with the agent.

Figure 3 shows a different view of the same data. Rather than showing time per iteration, we show *progress* – how many iterations has the application finished at a given point in time. Without the agent, the producer moves forward much faster than the consumer. This not only means that the queue gets longer (requiring more storage), but it also means that the consumer spends longer time running alone, which is the most likely reason for the lower performance compared to the setup with the agent. The consumer cannot fully utilize the machine, so the whole time it spends running alone, part of the CPU resources is not used. Figure 4 shows the number of threads assigned to the producer and the consumer, as well as the size of the queue. One can see that the queue is kept around the target of 8 threads. Also note that up to 40 threads are used in total, due to the thread boost feature of the agent.

As one can see from the results in Table II, the sequential execution is significantly slower than concurrent. This is due to the fact that neither the producer nor the consumer is optimized enough to fully use the whole machine. The best performance is achieved with the agent, although the difference is small (but statistically significant, with $p < 0.01$).

2) Fast producer without up-down: If we disable the up-down feature of the producer, the time per iteration is constant in both the producer and the consumer, with the producer being around 1.9x faster than the consumer. The overall results of the experiment are similar to the case when up-down is enabled, as can be seen in Table II. Even in this setup, the use of the agent is beneficial, improving the overall performance (the difference is statistically significant, with $p < 0.01$).

We have also tried disabling the thread boosting feature of the agent. In that case, performance is significantly reduced – the execution takes 295.69 seconds, compared to 269.46 with boost, only slightly faster than sequential execution (304.13).

3) Same speed of producer and consumer: For the next experiment, we have not used the up-down feature of the producer and adjusted the difficulty of the producer and consumer to be the same. As a result, when executed sequentially, the producer and consumer take the same amount of time. Again, the results of the experiments are shown in Table II. In this case, the best performance is achieved when the agent is disabled. Due to the way the experiment is set up, the producer and consumer stay in sync on their own, since the operating system does a very good job of assigning both of them the same amount of CPU time.

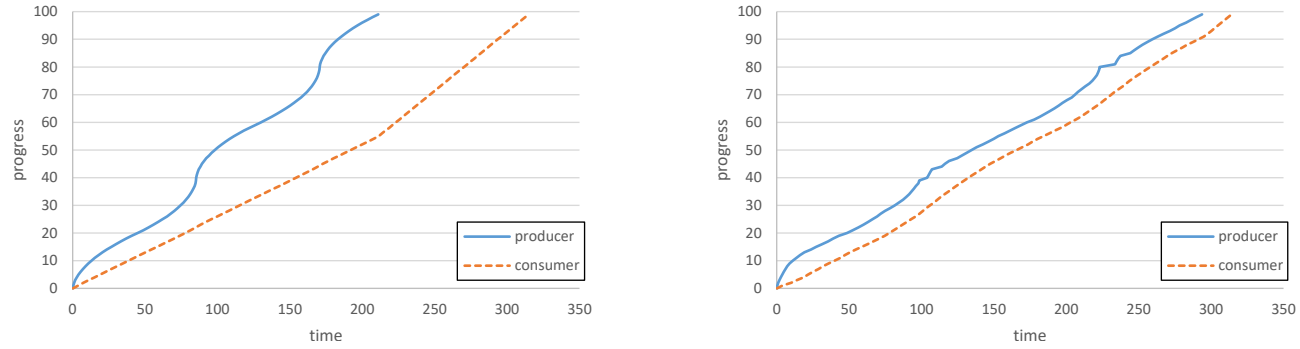
With the agent, some time is lost at the beginning, where the agent slows the consumer down, until the desired queue length (8 by default) is achieved. The similar problem is encountered at the end, where the producer finishes but the consumer still needs to process the 8 items (on average) in the queue. By reducing the size of the queue from 8 to 4, the performance is improved, getting close to the performance without the agent. This is the value shown in the table. Still, the configuration with the agent lags behind the configuration without the agent (again, the difference is small, but statistically significant, with $p < 0.01$). With target queue length of 8, the execution time increases by 5.59 seconds to 206.50 seconds.

4) Slow producer: In our last experiment, the producer is slower compared to the consumer (the consumer is around 3.2x faster). The up-down is not used in the producer, as it is not particularly interesting (the producer would be slower nearly all the time anyway). As one can see in Table II, unlike in the previous naturally balanced case, the agent is once again beneficial. The performance with the agent is slightly better (as before, the result is statistically significant, with p even below 0.001). The likely reason for that is that without the agent the consumer finishes too early in each iteration and is then idle for the remainder of the iteration. As the producer is unable to fully utilize the machine, some efficiency is lost. This outweighs even the overhead added by the agent, making the use of the agent the better option.

VI. CONCLUSION AND FUTURE WORK

We have explored the possibility of achieving a fine-grained control of application execution by adjusting the number of worker threads used by the task-based runtime system. It turns out to be a viable option, introducing only a small overhead and in some cases even improving the overall performance by improving resource utilization. The producer-consumer scenario has shown improved performance in all cases, except for the naturally balanced scenario where the producer and the consumer make progress at the same pace on their own. Furthermore, the agent prevents the producer and consumer from drifting apart, therefore significantly reducing the storage requirements.

However, the biggest challenge would be applying these node-level improvements on a cluster level. The number of threads can be adjusted quickly because all data is available to all threads, so tasks can be moved from the suspended threads to the still active threads with minimal cost. In a distributed memory setting, the cost is much higher. Our approach could easily be used to improve performance within the individual nodes, if the nodes are not used exclusively by just a single job. With OCR, tasks can also be moved between the nodes, so the techniques could be applied on the cluster level, but the issues of data movement cost and scalability have to be



(a) without the agent, the producer moves ahead too quickly

(b) with the agent, the producer and consumer are in sync

Fig. 3. Progress of the producer and consumer. The progress is shown as a number of completed iterations at a given time. By definition, the producer is always ahead of the consumer, so the producer line is above the consumer line. The distance between the line at any given time is the length of the queue.

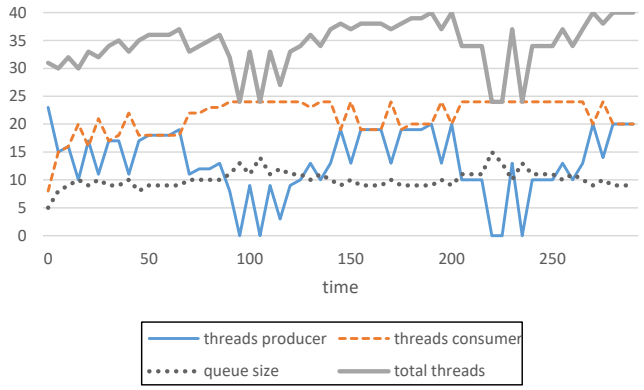


Fig. 4. Thread counts and queue size. The figure shows the number of threads assigned to the producer and consumer, as well as the total number of threads (sum of the two). Even though the machine can only support 24 threads, up to 40 threads are actually used due to the "boost" feature of the agent. The figure also shows that the queue size is kept around the target of 8 items. Due to the way the data for this graph is acquired by the agent, we do not have data points for the short period after the producer finishes (295 seconds) until the consumer finishes (316 seconds). The agent becomes inactive at that point, as no further coordination is needed. The consumer is assigned the maximal number of threads to process the remaining items.

addressed. It is likely that the reactive model that we used for the producer-consumer strategy would have to be replaced by a more sophisticated predictive model, which would allow planning the data movement in advance.

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] 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*, 23:187–198, 2011.

[2] G. Aupy, M. Shantharam, A. Benoit, Y. Robert, and P. Raghavan. Co-scheduling algorithms for high-throughput workload execution. *Journal of Scheduling*, 19(6):627–640, Dec 2016.

[3] M. Bhadauria and S. A. McKee. An approach to resource-aware co-scheduling for cmps. In *Proceedings of the 24th ACM International Conference on Supercomputing*, ICS '10, pages 189–199. ACM, 2010.

[4] J. Dokulil and S. Benkner. The Open Community Runtime on the Intel Knights Landing architecture. In *17th International Conference on Algorithms and Architectures for Parallel Processing (ICA3PP)*, 2017.

[5] J. Dokulil, M. Sandrieser, and S. Benkner. Implementing the open community runtime for shared-memory and distributed-memory systems. In *2016 24th Euromicro International Conference on Parallel, Distributed, and Network-Based Processing (PDP)*, pages 364–368, Feb 2016.

[6] N. Fabian, K. Moreland, D. Thompson, A. C. Bauer, P. Marion, B. Gevecik, M. Rasquin, and K. E. Jansen. The paraview coprocessing library: A scalable, general purpose in situ visualization library. In *Large Data Analysis and Visualization (LDAV), 2011 IEEE Symposium on*, pages 89–96. IEEE, 2011.

[7] D. G. Feitelson, L. Rudolph, U. Schwiegelshohn, K. C. Sevcik, and P. Wong. Theory and practice in parallel job scheduling. In *Workshop on Job Scheduling Strategies for Parallel Processing*, pages 1–34. Springer, 1997.

[8] L. V. Kale and S. Krishnan. Charm++: A portable concurrent object oriented system based on C++. In *Proc. of the Eighth Annual Conference on Object-oriented Programming Systems, Languages, and Applications*, OOPSLA '93, pages 91–108, New York, USA, 1993. ACM.

[9] S. Kamil, J. Shalf, and E. Strohmaier. Power efficiency in high performance computing. In *2008 IEEE International Symposium on Parallel and Distributed Processing*, pages 1–8, April 2008.

[10] A. Kukanov and M. J. Voss. The foundations for scalable multi-core software in Intel Threading Building Blocks. *Intel Technology Journal*, 11(04):309–322, 2007.

[11] K.-L. Ma. In situ visualization at extreme scale: Challenges and opportunities. *IEEE Computer Graphics and Applications*, 29(6):14–19, 2009.

[12] O. Mämmelä, M. Majanen, R. Basmadjian, H. De Meer, A. Giesler, and W. Homberg. Energy-aware job scheduler for high-performance computing. *Computer Science - Research and Development*, 27(4):265–275, Nov 2012.

[13] T. Mattson and R. Cledat, editors. *The Open Community Runtime Interface*. April 2016. https://xstack.exascale-tech.com/git/public?p=ocr.git;a=blob;f=ocr/spec/ocr-1.1.0.pdf.

[14] D. K. Newsom, O. Serres, S. F. Azari, A. H. A. Badawy, and T. El-Ghazawi. Energy efficient job co-scheduling for high-performance parallel computing clusters. In *2015 IEEE International Conference on Smart City/SocialCom/SustainCom (SmartCity)*, pages 550–556, 2015.

[15] S. Uwe and Y. Ramin. Fairness in parallel job scheduling. *Journal of Scheduling*, 3(5):297–320, 2000.

[16] J. Weidendorfer and J. Breitbart. Detailed characterization of hpc applications for co-scheduling. In *Proceedings of the 1st COSH Workshop on Co-Scheduling of HPC Applications*, page 19, 2016.