An Adaptive Load-balancer for Task-scheduling in FastFlow

Md Moniruzzaman[*], Kamran Idrees[*], Michael Rossbory[†], José Gracia[*]
[*]High Performance Computing Center Stuttgart (HLRS), University of Stuttgart, Germany
[†]Software Competence Center Hagenberg GmbH, Austria
moniruzzaman@hlrs.de, idrees@hlrs.de, Michael.Rossbory@scch.at, gracia@hlrs.de

Abstract*—Balancing the computational load of multiple concurrent tasks on heterogeneous architectures is one of the critical requirements for efficient usage of such systems. Load-imbalance is inherently present if the computation load is distributed non-uniformly across various tasks or if execution time for the same kind of tasks varies from one class of processing element to the other. Load-imbalance may however also arise from causes that are beyond the control of the user, as for instance operating system jitter, over-subscription of the available workers, interference and resource contention by concurrent tasks, etc. Writing a balanced parallel application requires careful analysis of the problem and good understating of various hardware architectures of the computing nodes. FastFlow is a C++ library that offers high-level parallel pattern abstractions on the user side, and lowers those onto efficiently implemented architecture specific skeletons. The default FastFlow scheduler, however, assigns tasks to workers in a round-robin fashion and is thus not well suited to handle load-imbalance. In this paper, we present an adaptive load-balancing task scheduler for FastFlow, a model for the expected relative performance of our adaptive scheduler over the default round-robin scheduler, and finally evaluate the quality of the implementation with low-level as well as two specific application benchmarks. We find that the adaptive load-balancer does not introduce additional overheads if load-imbalances are not present, and that our scheme is particularly efficient in mitigating the effect of thread over-subscription. Finally, we show that the proposed scheduler can lead to substantial performance gain for real industrial applications.

Keywords–task scheduling; load balancing; heterogenous architecture; NUMA; FastFlow.

I. Introduction

The efficient utilisation of heterogeneous computing systems, in particular the balancing of application load on the available processing elements or workers, is a non-trivial tasks. One of the key factors for load-imbalance is the heterogeneity of processing elements architecture – a GPGPU will be faster than a CPU for certain workloads (and slower for others) [7]. Therefore, one has to consider the computing power at the time of assigning work to a particular worker. Ideally, the faster worker should not wait for the slower one in a heterogeneous hardware environment.

A further, often underestimated source for load-imbalance is variation of the execution time due to the very presence of other concurrent tasks as for instance operating system jitter, over-subscription of the available workers, interference and resource contention by concurrent tasks, etc. A particular example is inefficient access to memory and caches due to non-uniform memory architectures (NUMA), which are common already on multi-core systems.

The most obvious source for load-imbalance, however, is that the task itself exhibits non-uniform task execution time. The execution time of the task is not always the same, and the exact time requirement by the task, even in the case of homogeneous hardware environment, is not easy to determine. For instance, root-finding algorithms to minimize complex mathematical functions will converge much faster if the initial guess is close to the final minimum, but might take much longer if the initial guess is in an off region of the parameter space. In this sense, the execution time of the tasks is non-uniformly distributed.

If we consider the block of code in Figure 1, one easily notices that the execution time of the task `aTask` varies depending upon which section of if-then-else clause is being executed. Now, the challenge is that we cannot determine the execution time of a task prior to its execution. However, load-imbalance would be obvious, if we blindly distribute a number of tasks equally to different workers without having prior knowledge of the required time to finish the task. Clearly, equal task distribution to all workers is not an optimal solution in this scenario because the non-uniform task execution time is caused by the task itself and the underlying hardware architectures as discussed earlier. This non-uniform execution behaviour is one of the primary reasons of load-imbalance and non-optimal solution [8]. In order to mitigate this situation, we need an adaptive load-balancing task scheduler.

In this paper, we propose a novel idea for an adaptive, load-balancing scheduler and realize it for FastFlow programming model [1]. The details of the approach are described in the Section II along with FastFlow's existing task scheduler. Our approach is evaluated by a series of benchmarks in Section III. The benchmarks include not only synthetic ones, but also two real world applications, namely a Molecular Dynamic code [5] and a Material Optimization Process [10]. Finally, we draw our conclusion in Section IV.

```
def aTask(condition) {
  if (condition)
    longCalculation()
  else
    shortCalculation()
}
```

Figure 1. An example of non-uniform task

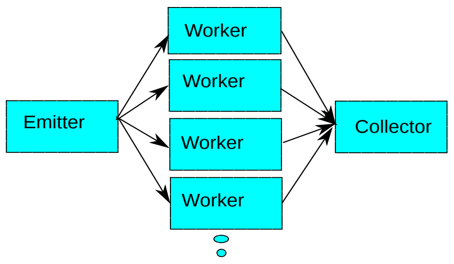

Figure 2. a FastFlow Farm pattern.

II. Related work and our approach

FastFlow is a C++ framework to write parallel programs [1]. The framework consists of parallel programming patterns such as farm, pipeline, map, etc, which are composed of various so-called nodes. The farm pattern for instance, is composed by an emitter node, several worker nodes, and a collector node as shown in the Figure 2. Note that in FastFlow, all nodes are implemented as individual OS threads. For instance in the farm pattern, each of the various worker nodes as well as the emitter and collector nodes, respectively, are separate threads. The nodes have input and output queues to connect to each other thus forming basic patterns. An emitter node, with help of an embedded load balancer, distributes tasks to the workers through their input queues. These workers, in principle, do the job concurrently, and the collector collects the results.

Task scheduling for parallel resources is an active area of research [13],[14] . However, the default scheduling policy of the emitter's embedded load-balancer is round-robin. Therefore, we refer to this default scheduler as round-robin scheduler (RRS). As argued in the introduction, we believe that the round-robin scheduling policy is not suitable for non-uniform tasks executing environment. Therefore, we propose an new load-balancing scheme, namely adaptive round-robin scheduler (ARRS). The pseudo-code of the ARRS is shown in Figure 5; the basic working is explained in the following.

The basic idea of the ARRS is to respect the computing capability of workers, i.e., faster workers will receive more tasks than slower workers. We do so by keeping the height of the input queues roughly at the same level. Whenever, a task is ready for being emitted onto the workers, it is issued to the worker with the smallest input queue length, i.e., with the lowest queue level.

Determining the length of input queues in FastFlow, however, is a comparatively expensive operation. In fact, FastFlow aims to take scheduling decisions within a few clock cycles. Recording the queue level of all workers, would thus incur a prohibitively expensive overhead. Instead, we do not record queue levels at each task scheduling event, but only at one in `CHUNK_SIZE` events, thus reducing the overhead by the same factor.

In many applications, tasks are created in bursts, with relatively few tasks emitted between bursts. Scheduling all tasks, which are of unknown duration, could lead to a situation, where the queue levels are balanced in terms of number of tasks, but very unbalanced in terms of total duration

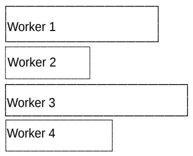

Figure 3. Queue length difference at a certain point.

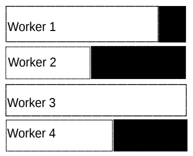

Figure 4. ARRS tries to balance the queues of the workers.

of the tasks in the respective queues. In order to mitigate this situation, we use a throttling technique. Tasks are not necessarily scheduled as soon as they become available, instead the scheduler defers scheduling decisions until at least one of the input queues of the workers is nearly empty. Only then we procede with scheduling a block of tasks before throttling again. For simplicity, the size of scheduling block is set to `CHUNK_SIZE`, i.e., the frequency of queue level recording.

As said earlier, continuously monitoring the length of the queues is prohibitively expensive. Therefore, we inject a so called *fake-task* into the queues of each worker towards the end of the block-wise scheduling episode. As soon as the first worker executes this fake-task, the throttling mechanism will start the next block-wise scheduling episode. Our scheduler is thus asynchronously driven by the length of the queues and does not require further explicit synchronisation. Note, that the queue of the fastest worker is not allowed to run fully empty. The fake-tasks are placed in the worker queues as soon as only a small number, `NEAR_END`, of tasks remain to be scheduled in the curent scheduling episode. This parameter is set to a small multiple of the number of workers.

Within a scheduling episode, incoming tasks are scheduled onto the shorter queues in such a way that differences in queue lengths, which have been precomputed at the time of queue level recording, are leveled out, as illustrated in Figures 3 and 4. If the differences in queue length are smaller the `CHUNK_SIZE` to begin with, the queue lengths will balance well within the scheduling episode. The scheduler will then revert to a round-robin policy for the rest of the scheduling episode. Notably, ARRS will also revert to a round-robin policy at the very start of the application.

The proposed adaptive round-robin scheduler is a drop-in replacement for the default FastFlow round-robin scheduler and fully transparent to the user. ARRS is required to perform as well as RRS when there is no load-imbalance of tasks, i.e., the overhead on ARRS over RRS needs to be small. The efficiency of ARRS in the case of load-imbalance is determined mostly by the value of the parameter `CHUNK_SIZE`, i.e., the frequency of queue level recording and the duration over which it is assumed that relative queue levels do not change significantly. The other parameter `NEAR_END`, which gives the lead time of starting a scheduling episode over the emptying of the queues has been found to have a negligible effect as long as it is in the order of a few times the number of workers.

III. Evaluation

The proposed load-balancer ARRS has been integrated into FastFlow version 2.0.0 [6]. The same version of FastFlow with RRS has been compared with the performance of ARRS.

```
#define CHUNK_SIZE 300
#define NEAR_END 4*NUMBER_OF_WORKERS
int n_scheduled = 0

def arrsScheduler(the_task) {
    if (n_scheduled<CHUNK_SIZE) {
        // hand-out tasks

        if (queuesUnbalanced()) {
            scheduleLowestQueue(the_task)
        } else {
            scheduleRoundRobin(the_task)
        }

        if (CHUNK_SIZE-n_scheduled==NEAR_END) {
            sendFakeTaskToAllWorkers()
        }

        n_scheduled++
        return true
    } else {
        // wait for queues to run empty
        waitForFakeTask()

        recordQueueLevels()
        n_scheduled = 0
        return false
        }
    }
}
```

Figure 5. Pseudo code of the scheduling algorithm

During the experiments no other user process was running on the machine.

The evaluation steps are divided in to four categories. Firstly, we measured the overhead of the scheduler by creating a synthetic benchmark. Here, we want to see the possibility of using ARRS regardless of task type (uniform or non-uniform) as replacement for RRS. Secondly, we have defined a performance model for heterogenous hardware which has explicit non-uniform task execution behavior, and we compared the performance of the scheduler with the model by using a hardware simulation technique. Thirdly, we assessed the scheduler with implicit producer of load-imbalance such as NUMA and thread over-subscription. Finally, two real world applications have been exercised with the scheduler. During the experiment, the default `CHUNK_SIZE` was 300, if it is not mentioned explicitly.

Throughout the paper, we will discuss the overhead and benefit of ARRS over RRS in terms of the relative performance (improvement), S, for a given benchmark. The experimental relative performance is calculated as the ratio of average execution times using RRS, t_R, and ARRS, t_A, respectively, i.e.

$$S = \frac{\langle t_R \rangle}{\langle t_A \rangle} \tag{1}$$

In all benchmarks, the averages have been calculated over at least 10 runs. Error are estimated from the standard error of mean over the samples. Error bars are shown in plots only if they are relatively large.

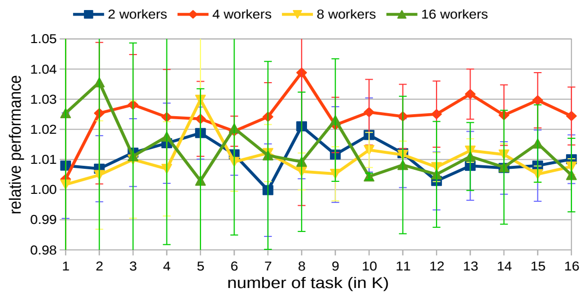

Figure 6. Relative performance of ARRS over RRS with uniform tasks.

A. *Evaluation environment*

The experiments have been conducted on an Intel Nehalem machine. This machine has two NUMA domain each with 4 cores. However, since hyper-threading was enabled, operating systems see (8+8) 16 logical cores. Each core is equipped with 32KB L1 data cache and 256KB L2 cache. A 8MB L3 cache is shared by all cores on a socket. Each socket has 6GB memory. The two sockets are connected by a Quick Path Interconnect (QPI) bus controller.

B. *Overhead*

The purpose of this section is to measure the scheduling overhead by ARRS. To measure the overhead, a FastFlow application with uniform task is devised. The uniform task can be described as a task that requires, in principle, the same amount of time every time it is executed by a worker.

The FastFlow application consist of a farm, i.e., emitter, collector and workers. The experiment was exercised with 2, 4, 8 and 16 workers separately. The emitter of the farm distributes tasks to the workers through a scheduler (RRS or ARRS). The uniform task consists in calculating all prime numbers up to 75,00, which takes $\sim 0.28\,\mathrm{ms}$ time to execute on this machine.

Figure 6 shows the relative performance of ARRS over RRS as a function of number of scheduler tasks for this FastFlow application running with 2, 4, 8 and 16 workers, respectively. If the value of the relative performance, S, is 1, both schedulers perform equally; if the value is higher than 1, ARRS performs better than RRS. The error estimate for the relative performance is shown at each point of the experiment with error bars. The relative performance is on average always larger than 1 for this benchmark. Therefore, we can conclude there is no significant overhead of ARRS compared to RRS for uniform task. In fact, the trend seems to indicate that ARRS performs slightly better than RRS. This is most likely due to the presence of OS jitter which causes slight imbalance of tasks [3], which do affect the execution time of with RRS, but are leveled out with ARRS.

C. *Explicit load-imbalance of task execution*

This section presents an idealized model for the relative performance of ARRS over RRS in case of heterogeneous architectures. The model is used to evaluate the efficiency of the implementation. Let us consider that number of tasks to schedule is n. All tasks are uniform, i.e., they take the same

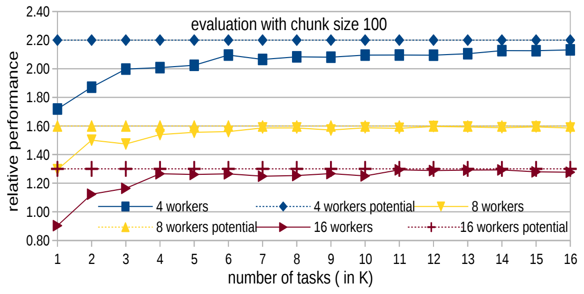

Figure 7. Performance comparison in hardware simulation with 1 faster worker.

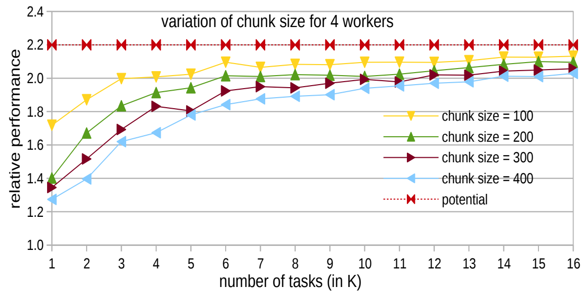

Figure 8. Performance comparison in hardware simulation with 1 faster worker.

time to execute on given architecture. However, the execution time on different architectures may differ. Let p be the number of workers. All workers participate in the execution of tasks and do nothing else. We assume there is no overhead by the scheduler. Each worker i, with $1 \leq i \leq p$, shows relative speed s_i for the task to execute. The higher the value of s_i, the faster a given worker executes. Without loss of generality, assume worker $i = 1$ is the slowest worker.

The time to execute all n tasks by an ideal round-robin scheduler is $t_R \propto \max(\frac{n}{p\,s_i}) = \frac{n}{p\,s_1}$, where the maximum is taken over the group of workers, and the last equality is due to the assumption that worker $i = 1$ is the slowest one.

In the case of an ideal adaptive round-robin scheduler, each worker executes $n_i = ns_i / \sum_p s_i$ out of n total tasks. Here, $\sum_p s_i$ can be considered as aggregated computing power of the workers. If the work load is perfectly balanced, then the time to execute all n tasks is given by $t_A \propto n_i/s_i = n/\sum_p s_i$.

Finally, the expected potential relative performance, S, of the ideal adaptive round-robin scheduler over the non-loadbalancing round-robin scheduler is

$$S = t_R/t_A = \frac{\sum_p s_i}{p\,s_1} \qquad (2)$$

The main difference between the assumed ideal adaptive round-robin scheduler and our ARRS is that the former takes scheduling decisions for each incoming tasks, while ARRS does so block-wise for `CHUNK_SIZE` tasks.

In the following, we simulate the effect of a heterogenous system consisting of two types of workers and contrast the experimental relative performance with the ideal model in (2). We simulate heterogeneity by consistently decreasing the execution time of tasks on exactly one specific worker. Specifically, the tasks on slower workers calculate all prime numbers up to 7500, the fast worker only up to 2000. In our system, the execution times are $\sim 0.28\,\text{ms}$ and $\sim 0.048\,\text{ms}$, i.e., $s_1 = 1, s_f \sim 5.8$.

The result of the simulation in terms of measured relative performance (see (1)) as a function of total number of tasks is shown in Figure 7. The experiment was conducted with 4, 8 and 16 workers, respectively, out of which one is considered to be faster than the others. The plot also indicates the expected relative performance calculated from the model (2). The data demonstrates, that in general ARRS always schedules tasks

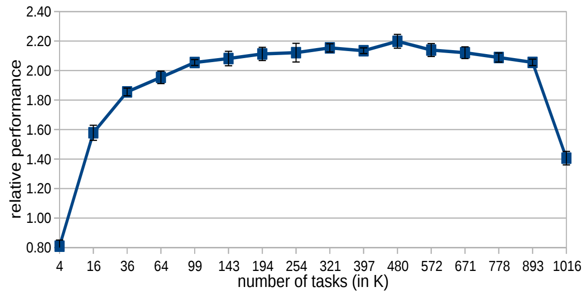

Figure 9. Relative performance improvement by ARRS over RRS by mitigating NUMA effect.

more efficiently than RRS, as the relative performance is always larger than unity.

Notably, at low number of tasks the relative performance is rather low, but increases with increasing number of tasks as it approaches the expected theoretical value (2) at larger number of tasks. As anticipated the parameter `CHUNK_SIZE` has an impact of the efficiency of our (non-ideal) ARRS. Figure 8 shows the same benchmark as above with varying `CHUNK_SIZE`. The plot shows clearly that the theoretical curve is approached faster for smaller values of the parameter. Our implementation of ARRS, unlike the ideal theoretical one, schedules tasks in blocks. The first block, however, is scheduled in a round-robin fashion resulting in load-imbalanced queues on heterogenous systems. The imbalance needs to be corrected by successive scheduling event; however, the magnitude of the initial load-imbalance increases with on the block size and thus takes longer to correct for.

D. Implicit load-imbalance of task execution

There are many possible effects giving rise to an implicit imbalanced situational to a parallel application. In this section, we discuss 1) non-uniform memory architectures, 2) thread or worker over-subscription, as implicit cause of load-imbalance.

1) NUMA effect: In a NUMA architecture, threads running on cores which belong to one NUMA domain, suffer performance penalties if they access data which resides in another NUMA domain. This phenomenon may result in non-uniform task execution times [11], particularly if data access patterns do not allow to cache data efficiently.

We have implemented a benchmark consisting of only 2

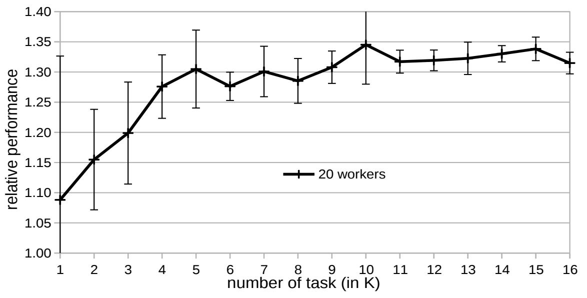

Figure 10. Relative performance of ARRS over RRS with uniform tasks & thread over-subscription.

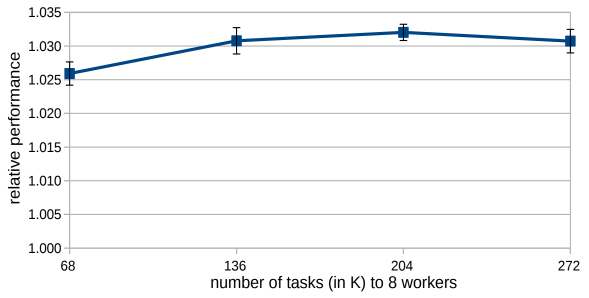

Figure 11. CMD Speedup (with ARRS over RRS) versus number of task/molecules.

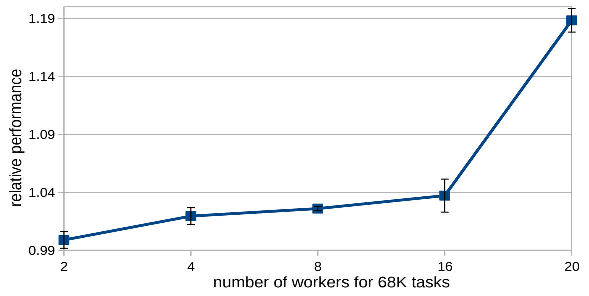

Figure 12. CMD Speedup (with ARRS over RRS) versus number of workers.

workers, for simplicity. Both workers are pinned to different NUMA domains using a tool called *likwid* [4]. Each time the task is invoked, it accesses and updates a different small region of memory in a cache in-efficient manner. All the data regions used in the benchmark have been initialized by only one of the workers and thus reside in its entirety in this worker's NUMA domain.

Figure 9 shows relative performance of ARRS over RRS for this NUMA scenario. The figure shows very clearly that NUMA effects have a significant impact on the performance, which is mitigated by the ARRS scheduler. In fact, one can estimate the ratio, $s_2/s_1 = 2S-1 \approx 3.4$, of the execution time of a task on both workers, respectively, from the maximum of the relative performance, $S \approx 2.2$, taken from the figure. Again we observe a slow rise-up of the efficiency. Notably, for very large number of tasks, the relative performance goes down again. Here, the memory used by the benchmark is so large, that it no longer fits into a single NUMA domain and is thus distributed among both NUMA domains. This results in overall less load-imbalance as both workers – not only one as before – have to access data from remote domains.

2) Thread over-subscription: Typical FastFlow applications consist of dozens of FastFlow nodes organized in (possibly nested) patterns as farm, pipeline, map, etc. Each of these nodes is mapped to its own separate thread. Thus in general, the number of threads of a FastFlow application is larger than the number of processing elements, say cores, on the system it executes on. Moreover, all of these threads compete for their share of on-core-time at the same time. We refer to this scenario as thread or worker over-subscription. The situation is worsened by the fact that FastFlow workers are usually pinned to specific processing elements.

We have exercised the same uniform-tasks benchmark code as in Section III-B for 20 workers. In our system, 8 of these workers share a core, while the remaining 12 are assigned to a core exclusively. Figure 10 shows the experimental result of this benchmark. Again, ARRS mitigates the effect of worker over-subscription although the variation of the benchmark's execution time is relatively large (as evident from the error bars), possibly due to the larger impact of OS jitter and other factors when the system is under such increased stress.

E. Real world applications

1) Molecular Dynamics: We have tested ARRS with a Molecular Dynamics simulation code called *CMD* [5]. Molecular Dynamics (MD) is a simulation methodology used for modeling of molecules in a large range of scientific fields as material science, chemistry, theoretical physics, and biology. The code spends most of its time in the calculation of the force acting between any pairs of molecules or atoms [5]. This part was parallelised with a FastFlow farm pattern [12] in such a way, that a task consists of the force calculation for single pair of molecules. The tasks takes the same (very short, ≈ 8 ms for 68000 pair of molecules using 2 workers) amount of time for each pair [9] and is thus considered uniform in the sense used in this paper. We therefore do not expect any performance improvement, but would rather aim to assert that the overhead of ARRS is negligible also for this real application exhibiting very short task duration.

We have run CMD on 8 workers with realistic parameters similar to those that would be used in smallish MD production runs. Figure 11 indeed shows that ARRS does not introduce any measurable overhead over RRS. In fact, it performs consistently better than RRS by a few percent.

While CMD, due to its simple structure, would in general not be used under conditions of thread over-subscription, we have nonetheless benchmarked it. Figure 12 illustrates the relative performance for number of workers from 2 to 20. The relative performance of ARRS over RRS increases slightly with the number of workers up to 16. This increase likely stems from load-imbalance originating in the higher likely-hood with increasing thread count of adverse thread-synchronization. At 20 workers, the application is load-imbalanced as some of the CPU cores need to serve two workers, while other only one. This scenario is much more efficiently handled by ARRS than by RRS.

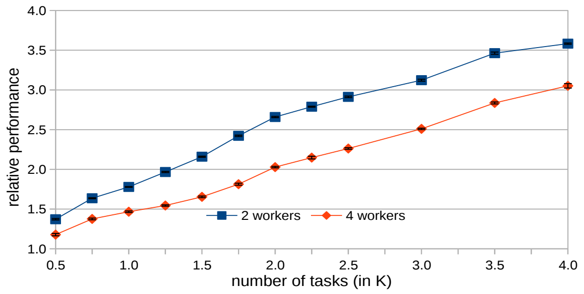

Figure 13. Material Optimization Process speed-up by ARRS over RRS.

2) Material Optimization Process: The purpose of this application is to minimize the consumption of raw material in an industrial production process, and thus, reduce the production cost. The application is a complex mathematical optimization process subject to multiple constraints. In essence, it maintains a pool of given size of acceptable mathematical solutions. Solutions are taken from the pool and passed on to one of various different pipelines for further optimization. Each of these optimization pipelines follows different strategies, all of which take different time to finish. Solutions which potentially improve on the optimization goal are returned to the pool for further processing.

This application has been parallelized using a FastFlow farm. Each of the farm's workers follows a specific optimization strategy resulting in varying execution time across workers. The farm's emitter issues solutions from the task pool to the workers, while the collector replenishes the pool with promising solutions. The application continues until a desired optimal solution is found, the pool is empty, or another termination criterion is triggered [10]. In this case the application, running with 2 workers to compute 1000 task, needs $\approx$ 70917 ms with RRS whereas $\approx$ 39865 ms with ARRS. Figure 13 shows the relative performance of ARRS over RRS as a function of the size of the solution pool, which is a parameter that needs to be adapted by the user of the application to find the best trade-off between long application runtime and quality of the material optimization. While the pool size is not directly proportional to the total number of tasks executed by the application, in general the latter tends to increase with the pool size. Clearly, ARRS outperforms RRS for all values of the pool size. For realistic pool sizes ARRS can be more than three times faster than RRS if two workers are used.

IV. Conclusion

In this paper, we have presented an adaptive load-balancing task-scheduler, ARRS, and applied it to the FastFlow pattern-based parallel programming model. We have also presented an idealized theoretical model for its performance. We have benchmarked ARRS with the aim to assert 1) the overheads under conditions where it will not shown performance improvements by design, and 2) the efficiency of our implementation in terms of the idealized model.

Our benchmarks support the conclusion that ARRS overheads are negligibly small and that our implementation can reach the theoretically expected efficiency. However, the efficiency of our implementation depends on the number of tasks that are scheduled; it approaches the ideal value asymptotically as the number of tasks increases. Careful analysis of our implementation reveals that the cause for this behaviour is an initial phase – whose length is given by the block size parameter – of round-robin scheduling of tasks. This phase leads to initial load-imbalances that need to be balanced out over successive scheduling periods.

Nonetheless, our real world application benchmarks show that the load-balancing scheduler leads to substantial performance increases in all cases considered. For an industrial relevant Material Optimization Problem, ARRS is up to three times faster than the default scheduler. The proposed load-balancer is well capable of dealing with non-uniformity of tasks and heterogeneity of hardware. We also show that ARRS may mitigate the performance impact of NUMA effects, load-imbalances due to OS jitter, etc., and thread over-subscription. We claim that ARRS is drop-in replacement for FastFlow's default scheduler that increases the speed of most applications, particularly on heterogenous systems, at no additional development cost for the FastFlow user.

Future work will address the short-comings of our approach – which are relevant only at (possibly intermittent) low number of tasks – by shortening the initial round-robin phase and by adapting the scheduler's block size to the number of task that are ready for scheduling.

Acknowledgement

This work has been supported by the European Commission Seventh Framework Programme through the projects *ParaPhrase: Parallel Patterns for Adaptive Heterogeneous Multicore Systems* (contract no.: 288570) and *Polca: Programming Large Scale Heterogeneous Infrastructures* (contract no.: 619686), and by the German Research Foundation (DFG) through Priority Programme 1648 *Software for Exascale Computing (SPPEXA).*

References

[1] Aldinucci, Marco, Massimo Torquati, and Massimiliano Meneghin. "FastFlow: Efficient parallel streaming applications on multi-core." arXiv preprint arXiv:0909.1187 (2009).

[2] Torquati, Massimo. "Single-producer/single-consumer queues on shared cache multi-core systems." arXiv preprint arXiv:1012.1824 (2010).

[3] Vicente, Elder, and R. Matias. "Exploratory study on the linux os jitter." In Computing System Engineering (SBESC), 2012 Brazilian Symposium on, pp. 19-24. IEEE, 2012.

[4] Treibig, Jan, Georg Hager, and Gerhard Wellein. "LIKWID: Lightweight Performance Tools." In Competence in High Performance Computing 2010, pp. 165-175. Springer Berlin Heidelberg, 2012.

[5] Idrees, Kamran, Mathias Nachtmann, and Colin W. Glass. "Evaluation of FastFlow Technology for Real-World Application." In Sustained Simulation Performance 2013, pp. 77-88. Springer International Publishing, 2013.

[6] FastFlow Source: http://sourceforge.net/projects/mc-fastflow/files/fastflow-2.0.0.tar.bz2/download 2015.04.20

[7] Lee, Victor W., Changkyu Kim, Jatin Chhugani, Michael Deisher, Daehyun Kim, Anthony D. Nguyen, Nadathur Satish et al. "Debunking the 100X GPU vs. CPU myth: an evaluation of throughput computing on CPU and GPU." In ACM SIGARCH Computer Architecture News, vol. 38, no. 3, pp. 451-460. ACM, 2010.

[8] Goli, Mehdi, John McCall, Christopher Brown, Vladimir Janjic, and Kevin Hammond. ”Mapping parallel programs to heterogeneous CPU/GPU architectures using a monte carlo tree search.” In Evolutionary Computation (CEC), 2013 IEEE Congress on, pp. 2932-2939. IEEE, 2013.

[9] Niethammer, Christoph, Colin W. Glass, and José Gracia. ”Avoiding serialization effects in data/dependency aware task parallel algorithms for spatial decomposition.” In Parallel and Distributed Processing with Applications (ISPA), 2012 IEEE 10th International Symposium on, pp. 743-748. IEEE, 2012.

[10] Rossbory, Michael, and Werner Reisner. ”Parallelization of Algorithms for Linear Discrete Optimization Using ParaPhrase.” In DEXA Workshops, pp. 241-245. 2013.

[11] Thomadakis, Michael E. ”The architecture of the Nehalem processor and Nehalem-EP SMP platforms.” Resource 3 (2011): 2.

[12] Brown, Christopher, Vladimir Janjic, Kevin Hammond, Holger Schoner, Kamran Idrees, and Colin Glass. ”Agricultural reform: more efficient farming using advanced parallel refactoring tools.” In Parallel, Distributed and Network-Based Processing (PDP), 2014 22nd Euromicro International Conference on, pp. 36-43. IEEE, 2014.

[13] Thoman, Peter, Herbert Jordan, Simone Pellegrini, and Thomas Fahringer. ”Automatic OpenMP loop scheduling: a combined compiler and runtime approach.” In OpenMP in a Heterogeneous World, pp. 88-101. Springer Berlin Heidelberg, 2012.

[14] Wang, Zheng, and Michael FP O’Boyle. ”Mapping parallelism to multi-cores: a machine learning based approach.” In ACM Sigplan Notices, vol. 44, no. 4, pp. 75-84. ACM, 2009.