Distributed Parallel Particle Advection using Work Requesting

Cornelius Müller, David Camp, Bernd Hentschel and Christoph Garth

Abstract— Particle advection is an important vector field visualization technique that is difficult to apply to very large data sets in a distributed setting due to scalability limitations in existing algorithms. In this paper, we report on several experiments using work requesting dynamic scheduling which achieves balanced work distribution on arbitrary problems with minimal communication overhead. We present a corresponding prototype implementation, provide and analyze benchmark results, and compare our results to an existing algorithm.

1 Introduction

Integral curves are one of the most intuitive means to depict vector fields and they are a cornerstone of visualization and analysis across a variety of application domains. Successful application of integration-based techniques to large data must crucially leverage parallel computational resources to achieve well-performing visualization. However, scalable integral curve computation has been difficult to attain. Due to the strong inherent data dependency of such curves, effective load balancing has proven difficult, and resulting algorithms have shown limits in scalability.

In this paper, we investigate the application of a general-purpose load balancing technique – work requesting – to integral curve computation. *Work requesting*, and similarly its cousin *work stealing* have been proven versatile schemes at balancing even very irregular and dynamic loads well, while also exhibiting excellent scalability if implemented well [8]. From a theoretical perspective, these techniques should handle integral curve computation well. Moreover, they do not require *a priori* knowledge about vector field data, facilitate generation of new integral curves during computation, and have modest complexity of implementation compared to other specialized schemes.

Based on a prototype implementation, we report the results of several experiments centered on a typical integral curve use case. We have investigated both scalability and efficiency, and compare our implementation against a baseline approach. Our results show that work requesting yields very good efficiency and scales well to a modest numbers of processors, and should therefore be considered as a viable strategy for integral curve load balancing.

2 Related Work

2.1 Parallel Particle Advection

Integration-based techniques are one of the cornerstones of flow visualization [11]. Integral curves are typically classified into *streamlines* and *pathlines*, depending on whether the underlying data is steady-state or time-varying, respectively. Here, we will focus related work regarding the parallel computation of large amounts of integral curves.

Given reasonably small data, GPGPU-based approaches which use a single GPU have been very successful. They exploit the straightforward possibility of concurrently computing individual traces. Differences mainly result from the supported input data type, i.e. support for steady-state vs. time-varying data on the one hand and structured vs. unstructured grids on the other [10, 17, 2, 3]. However, it seems that GPU-based approaches best serve highly interactive use cases where the handling of large data is not the top priority.

- *Cornelius Müller and Christoph Garth are with the University of Kaiserslautern. E-mail: {cmueller, garth}@cs.uni-kl.de.*
- *David Camp is with Lawrence Berkeley National Laboratory. E-mail: dcamp@lbl.gov.*
- *Bernd Hentschel is with RWTH Aachen University. E-mail: hentschel@vr.rwth-aachen.de.*

In contrast, several recent publications have targeted the handling of very large data sets. Pugmire et al. [16] considered root causes of load imbalance and identified two fundamental distribution schemes that minimize either communication load or I/O load.

The *parallelize-over-seeds* (POS) technique statically distributes integral curves among tasks. Each task works on its integral curves in isolation, which minimizes communication overhead. Data blocks are loaded as required by the computation. This results in two disadvantages. First, redundant data loading results in high I/O overhead. Second, static scheduling leads to load imbalances because of the fact that integral curve lengths usually vary strongly. Thus, some tasks finish their work long before others, and they stay idle because the static scheduling prevents any redistribution of work.

In contrast, *parallelize-over-blocks* (POB) minimizes the I/O overhead by statically and non-redundantly distributing data blocks among tasks. Each task computes the trace segments within its assigned blocks. If traces leave the local data domain, they are sent to the task which holds the required block. Thus, no I/O is needed after an initial loading phase. However, the frequent migration of traces leads to significant communication overhead. Additionally, the computation becomes highly imbalanced if only a few blocks actually contain particles, i.e. some tasks are overloaded while others idle.

Due to the strong data dependency of integral curve computation, choosing the best parallelization strategy cannot be done *a priori*. To address this, Pugmire et al. developed a hybrid scheduling scheme that combines ideas of POB and POS [16]. A central master process dynamically assigns trace computations to slave processes. The scheme successfully mitigates load imbalances, yet its centralized nature may become a bottleneck for increasing processor counts.

Other solutions to the load-balancing issues are presented in [7, 14, 18]. Rather than adapting the scheduling of particle traces, they statically partition the domain into blocks of approximately equal workload by using different schemes. While these solutions yields good results, they comes at the cost of an additional pre-processing step.

Peterka et al. present a variant of POB that is able to handle time-varying data [15]. Based on this work, a pipeline-parallel computation of the Finite-Time Lyapunov Exponent (FTLE) has been proposed [13]. It exploits temporal coherence in order to advect particles from different points in time in parallel.

Most approaches are implemented using the Message Passing Interface (MPI) [12] only. Camp et al. proposed to use *hybrid parallelism* in order to better utilize the increasing number of processing units in today's HPC machines. They observed good scaling for a combination of process-level MPI parallelization and thread-level parallelization in comparison to an MPI-only implementation [5]. More recently, they demonstrated a system in which GPUs carry most of the workload [6]. With this approach, they specifically target a class of modern HPC cluster designs which combine a fast, many-core accelerator with classical CPUs.

2.2 Dynamic Scheduling

The key challenge for parallel integral curve computation is its *dynamic data dependency*: the exact data required to compute the curve cannot be determined *a priori*. This information, however, would be

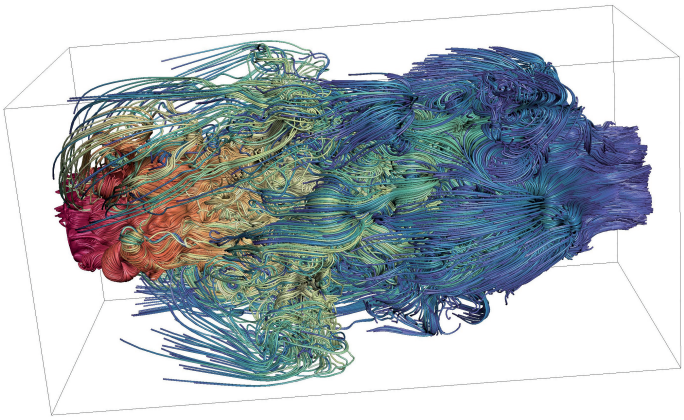

Fig. 1. An overview of the *Jet4* dataset showing 25,000 streamlines. A jet of fast air is entering a medium at rest (on the left), with streamlines seeded around the inlet. Strongly non-uniform particle behavior is apparent, with turbulence, recirculation, and laminar flow visible in the image. This induces a strong load imbalance when using the parallelize-over-seeds algorithm.

crucial for an optimal, parallel schedule. Hence, there is no way to create an efficient scheduling *a priori*. One either has to bear with an inefficient solution or perform *dynamic scheduling*.

In general, dynamic scheduling strategies are used for parallel computations, in which the distribution of the work is only discovered at runtime. Two proven schemes are *work stealing* and *work requesting*. For both, the problem is partitioned into several *work items* which are small enough to be solved by a single processing unit. A processing unit will be called a *task* in the rest of this paper to match the term used in MPI. In this paper, a work item correspond to an integral curve or the integration work on an integral curve. Work items are initially distributed among all tasks. Each task holds its items in a *work queue*, which is processed in FIFO order. Once a task runs out of work, it first dynamically selects a random task and then tries to retrieve work from that task's work queue. The requesting task is called a *thief*, the other task is the *victim*. If the victim itself has no work items, another task is randomly selected. It has been shown that choosing a victim *at random* yields optimal results in the general case [1].

The difference between work stealing and work requesting is how tasks obtain work items from one another. In a work requesting scheduler, the thief asks the victim for work and the victim actively provides it. In contrast, in a work stealing scheduler, the victim is oblivious of its role: the thief directly accesses to the victim's queue without the victim being actively involved. Due to the symmetric communication, work requesting schedulers incur more communication overhead, yet they are far easier to implement, particularly in a distributed memory setting. Both strategies have shown to be efficient in theory and practice [8], because communication is only required if the load actually is unbalanced.

3 Implementation

As mentioned above, an optimal schedule for integral curve computation cannot be determined *a priori*. Therefore, we suggest the use of work requesting as a dynamic scheduling strategy. The relative simplicity of work requesting is attractive, because it has been shown to work well for a large class of unbalanced computations, and includes relatively few parameters that need manual tweaking by a user. We decided against work stealing because it needs that each task has access to the work queues of all other tasks, requiring a form of remote memory access (RMA). However, it proved difficult to deploy a platform-independent RMA implementation as corresponding frameworks or libraries are typically less mature than implementations of the widely-used Message Passing Interface (MPI) [12].

Algorithm 1 Pseudo code of the worker thread.

```
loop
  if work queue empty then
    Wait for new work item or termination.
  else
    Pop work item from the queue's front.
    Load data block if needed.
    Integrate as long as all required blocks are in cache.
    if integral curve finished then
      Delete work item.
    else
      Create new work item starting at last particle position.
      Push new work item to the back of the work queue.
    end if
  end if
end loop
```

Algorithm 2 Pseudo code of the supervisor thread.

```
loop
  if received work request then
    Hand over half of the local work queue, starting at the back.
  end if
  if work queue empty then
    Perform termination detection.
    Pick random victim and request work from it.
  end if
end loop
```

In our application, each work item equals one integral curve. By default, all curves are equally distributed among all tasks; however, initial distribution is arbitrary and not a fundamental part of the adaptive distribution scheme. Therefore, we test the worst case where all work items are initially at just one task to see how fast the system can balance it self. But for normal calculation of integral curves, all tasks have the same number of work items at the beginning, which is also true for the other experiments.

Every task has a local block cache, where the data blocks which were previously loaded are kept to be used in further calculations. The cache may be smaller than the data set. If a newly loaded block does not fit into the cache, room is made by deleting the oldest block in a FIFO style.

Each task executes a basic parallelize-over-seeds algorithm. The processing of one work item consists of two parts. Firstly, the data block where the integral curve starts is loaded from disk if it is not already in the local block cache. This ensures that at least a part of the curve can be calculated. Secondly, the actual integration is done. In this second part, no additional data is loaded. This leads to two possible endings for the processing of the work item. Either the integral curve is completed. Then the work item will be deleted. Or the curve reaches a point where an unloaded block is needed. In this case, the unfinished curve forms a new work item. The point where the intega-tion stopped is the new starting point. The new work item will be put back in the work queue.

If it would be put at the front of the queue, it would be the next processed work item and certainly needs to load a new data block from disk. Therefore, it is put at the back of the queue, leaving the possibility that the next item at the front does not need to load a new block. This is likely when the work items are sorted at the start of the program, so that the starting points of the work items next to each other in the queue are also next to each other in the data domain.

The actual work requesting operates like described in the last section for the general work requesting algorithm: If one task has finished calculating all its work items, it sends a work request to another task, called victim. This victim is randomly chosen which was proven to be optimal [1]. Dividing the work so that both the victim and the thief have the same amount of work after the operation has shown the

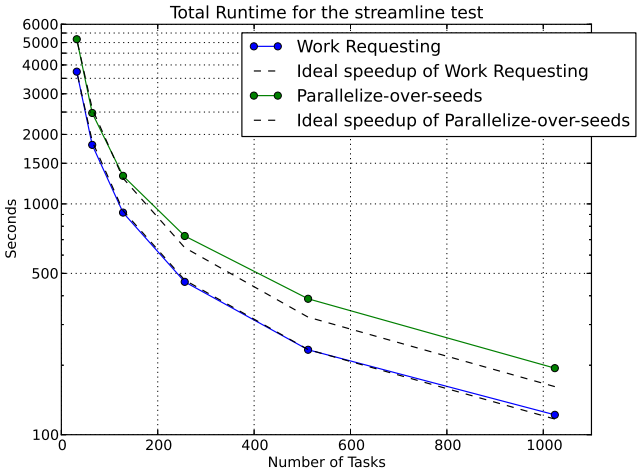

Fig. 2. Total runtime of the scalability test comparing work requesting with parallelize-over-seeds. The number of tasks goes from 32 to 1024. A clear gap between both algorithms can be seen which gets even bigger on higher numbers of tasks. Additionally, work requesting scales nearly perfect, while parallelize-over-seeds gets scalability problems from 256 tasks on.

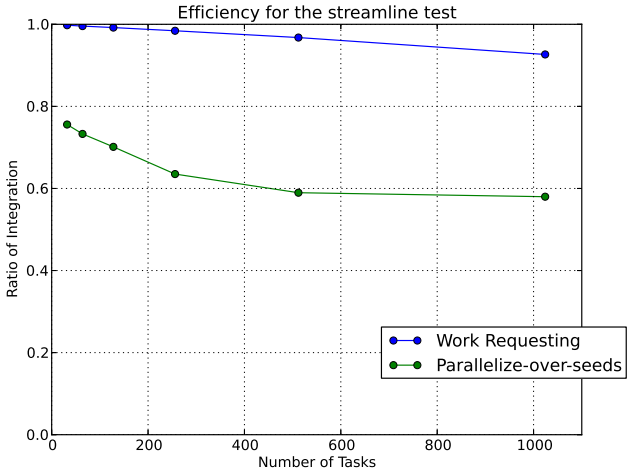

Fig. 3. Efficiency of the scalability test comparing work requesting with parallelize-over-seeds. Ratio of the total runtime which was spent doing the actual integration is shown. Work requesting has an efficiency of more than 90% even on 1024 tasks while parallelize-over-seeds achieves a maximum of only 76% and drops significantly on higher task counts.

best performance [8]. Since each integral curve is expected to take the same calculation time without further knowledge, the victim sends half of the work items in its queue back to the thief. As in [1], the local task takes its work items from one side of the queue (front) but the items which are given to another task is taken from the other side (back). The work items which would need to load a block are put at the back, so they are the first to be sent to thieves. This way, unnecessary I/O could be avoided because the victim does not have the needed block whereas the thief could have it in its block cache.

Our first implementation was single-threaded and the communication requests were only handled in between the processing of work items. This led to several problems. The most important one was that a task could not answer work requests from other tasks while it was calculating a work item, so the requesting task had to wait for some time before getting a response. This resulted in long idle times until a task with an empty work queue had found new work items. This had a significant negative impact on computation time and scalability.

Consequently, we implemented a multi-threaded version with two threads: a worker and a supervisor. The worker thread processes the work items, see Algorithm 1, which means it is responsible for I/O and integration. The supervisor thread handles the communication, see Algorithm 2. It replies to work requests from other tasks, sends out work requests if its local work queue is empty, and does the termination detection. This approach prevents the problems of the single-threaded version. The response times of the work requests are much shorter, which improves the work request performance.

Note that beyond using a block cache on each task, the implementation we describe here does not make use of some possible optimizations such as improved caching and other optimizations [8]. The selection of the victim is completely random to ensure work requesting fairness, and the work items are processed in sequence, without taking in account the data blocks in the cache. As it is our goal to evaluate work requesting as a distribution algorithm for integral curve parallelization, we aim to keep the basic algorithm as simple as possible to obtain an unobfuscated impression of work requesting scalability. Furthermore, diverging from the usual work requesting scheme, which has been shown to perform excellent in general problems [1, 8] can easily yield worse performance than without optimization. For example, custom selection of work items while stealing inevitably results in longer steal times and therefore more contention while looking for work [8].

4 Experiments

In this section we describe the experimental setup that we used to compare our new scheduling strategy to the established parallelize-over-seeds approach. We selected POS as base-line because it is similar to our work requesting implementation apart from the work balancing part. So we can eliminate other factors on the performance like initial I/O or calculation restrictions. The key question behind these experiments is whether or not the potentially better work distribution outweighs the additional communication involved in work requesting.

Dataset — All tests have been conducted with the *Jet4* simulation dataset, which describes a high-speed jet entering a medium at rest. Each of the 735 time steps consists of a regular grid with 32 M points, which amounts to 384 MB per time step and 275 GB overall. The time steps are partitioned into 1,024 blocks, each.

This dataset offers a large variety of different integral curve behaviors, including recirculation and quickly terminating curves. Figure 1 gives an impression of integral curves in this dataset. For streamline computations, we used a single time step, whereas pathline integration was executed on the entire data set.

Runtime Environment — All the test results in this paper were obtained on the Hopper system at Lawrence Berkeley National Laboratories. Hopper is a Cray XE6 system which features 153,216 compute cores organized in 6,384 compute nodes which are connected by a proprietary Cray Gemini network. Each node is equipped with two twelve-core AMD MagnyCours processors and 32 GB of memory. I/O is handled through a parallel Lustre file system that provides access to approximately 2 PB of usable disk space.

Test Cases — We performed a strong scaling study and an artificial load balancing test. In order to assess scaling behavior, we compare our work requesting scheme to POS for a varying number of tasks given a constant amount of work. For each run, 1 million integral curves were integrated. This number was chosen to ensure a sufficient degree of available parallelism. Our tests comprise settings for 32, 64, 128, 256, 512, and 1,024 tasks. Particles were seeded on a regular grid uniformly covering the entire domain.

In order to test the load balancing abilities of our work requesting scheme in an extreme case, we set up the following artificial distribution test. All of the starting seed points were initially assigned to task 0 and we then observed how work propagated throughout the system by the means of work requesting. We choose the same number and distribution of seed points as the scaling tests, so we could compare the

results but we limited the test to one run with 256 tasks. This test anticipates the case of strong load imbalance which is typically encountered in methods which seed new integral curves during runtime. Examples for such adaptive algorithms with on-the-fly refinement strategies are *stream surfaces* [6] and *streaklines*.

All measurements were collected with *VampirTrace* [9]. We perform manual instrumentation using VampirTrace's regions feature to reduce the overhead of profiling and obtain a clear picture of runtime spent in different phases of the algorithm. The sections we measured were initialization (consisting of setting up MPI, reading the meta data of the database and loading the initial seed set), communication (for work requesting and termination detection), idle time, data I/O, the integration calculation, and the termination detection (voting).

5 Results

We analyzed the timing logs to determine the scalability of both algorithms, and then compute the efficiency of each algorithm to integrate the integral curves. To test how well the work requesting algorithm can perform under extreme load imbalance we designed an artificial test to see the distribution of integration work across the tasks. The results are presented in this section.

5.1 Scalability Test

We look at the total time and efficiency to process 1 million integral curves at different concurrency levels. Additionally, Gantt charts are presented in Section 5.1.3 to show the load balancing effect.

5.1.1 Streamline Runtime

The total runtime of the test runs can be seen in Figure 2. The x-axis shows the number of tasks, from 32 to 1024. On the y-axis the total runtime in seconds can be seen using logarithmic scaling. The blue line represents the test using the work requesting algorithm and the green line represents the tests using the parallelize-over-seeds algorithm. For both, the ideal speedup is plotted as a dashed line. The ideal speedup shows the theoretical runtime if the performance of the program would scale linearly with the number of tasks. It was calculated based on the runtime of the test with 32 tasks, as in this test it can be assumed that the amount of overhead due to parallelization is the lowest.

It can be seen that there is a significant gap between the graphs corresponding to both algorithms. This gap increases further for higher number of tasks: using 32 tasks, the parallelize-over-seeds algorithm needs 38% more time to run than the work requesting algorithm, but using 1024 tasks, it needs 59% more.

The comparison with the ideal speedup is also interesting. For the work requesting algorithm, both lines are nearly identical, which means that it scales nearly perfectly. But using the parallelize-over-seeds algorithm, a clear gap can be seen for 256 and more tasks.

5.1.2 Streamline Efficiency

Efficiency is defined as the ratio of the total runtime which was spent doing the actual integration work. It is shown in Figure 3 on the y-axis. The x-axis again gives the number of tasks. On these graph, both axes use linear scaling. Again, the blue line represents the work requesting algorithm and the green line the parallelize-over-seeds algorithm.

It can be seen that the work requesting algorithm reaches nearly 100% efficiency on lower task counts and while it drops at higher task counts, it is still 93% using 1024 tasks.

On the other hand, the parallelize-over-seeds algorithm reaches a maximum of 76% efficiency using 32 tasks and drops significantly on higher task counts, down to 58%. But the decrease seems to slow down and will maybe settle above 55%.

5.1.3 Streamline Gantt Charts

A gantt chart shows a summary of one parallel program run. The x-axis represents the runtime of the program. Each thread is drawn as a horizontal bar. The color of the bar corresponds to the activity the thread did at the certain time: Blue stands for the integration work, green means loading a block, the different shades of red symbolize different kinds of communication, dark gray is the waiting of one task for the others after it finished and light gray is the sleeping the supervisor thread does when it is waiting for communication requests.

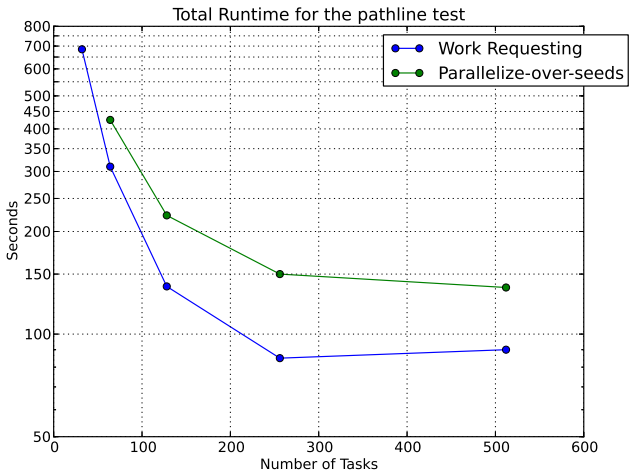

Fig. 6. Strong scaling results for pathline computation using work requesting and parallelize-over-seeds, respectively.

Figure 4 is the work requesting algorithm gantt chart, it can be seen that nearly 100% of the time is used for integration work, like one would assume from the efficiency plot. All of the tasks finish at nearly the same time.

Figure 5 is the parallelize-over-seeds algorithm gantt chart, the load imbalance can be seen clearly. The task which finishes its work last, takes approximately double time than the first task to complete its work. But all tasks have to wait for the other tasks to complete, leading to the inefficiency of the parallelize-over-seeds algorithm. With larger number of tasks, the difference between the completion of the first and last task gets even bigger, this explains the worse efficiency when using more tasks.

5.1.4 Pathline Runtimes

Figure 6 shows the strong scaling tests for both the work requesting and parallelize-over-seeds algorithms calculating pathlines on the Jet dataset. Both algorithms start scaling well, but then level out at 256 tasks. This leveling is because of the rising cost of I/O. Both algorithms have to load a lot more data to integrate the pathlines to completion. The cache size is reduced in half because of the requirement of needing two data blocks to integrate the curves between time segments. This reduces the chance of finding the data block needed to continue integrating a pathline. Also when the task steals work it is almost a given that every data block in the cache is no good, because data in the cache will be from the end of the data time and the new work items will require data blocks from a different time segment. This will likely cause the task to reload all new data.

5.2 Distribution Test

The distribution test analyzes the behavior of the algorithm under extreme load imbalance by initially assigning all work items to just one task. This anticipates the imbalance which adaptive methods like streaklines typically have. Figure 7 is a gantt chart of the this test. It shows only the first 50 seconds from the total runtime of 479 seconds and only about one thirds of the tasks. This is done to see the costs of distributing the work across the tasks. The task at the bottom is the one which initially has all the work items.

After the initialization time of about 4 seconds, all of the empty tasks send out work requests which can be seen in red. The bottom task sends out work items (purple), loads the necessary data blocks (blue and green), and starts integrating its work items. It takes roughly 15 seconds for most of the other tasks to have work items and start

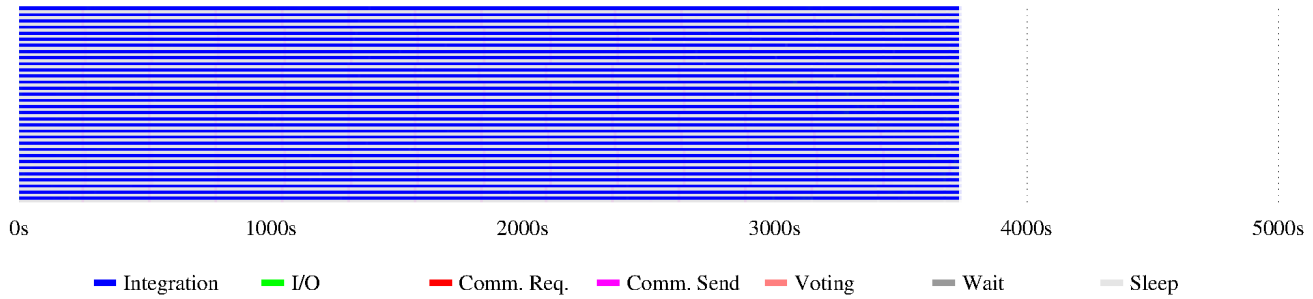

Fig. 4. Gantt chart of the work requesting algorithm using 32 tasks. It can be seen that nearly all the time is spent with integration and all tasks finish at nearly the same time, leading to a visible shorter runtime than parallelize-over-seeds as seen in Figure 5.

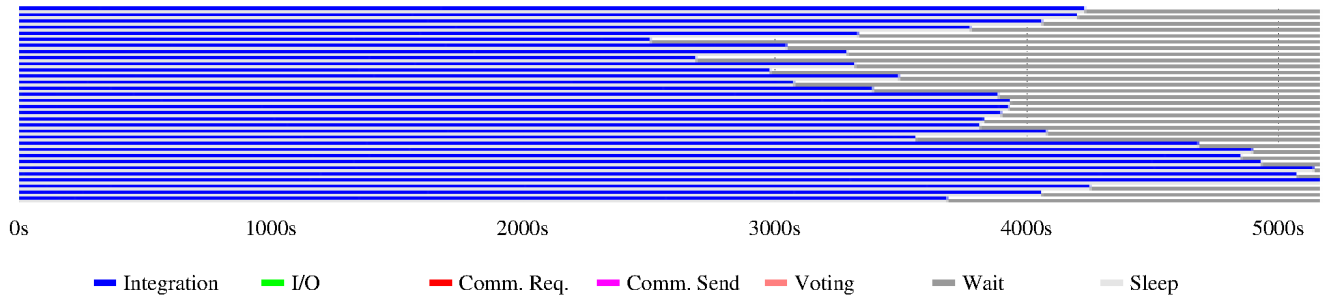

Fig. 5. Gantt chart of the parallelize-over-seeds algorithm using 32 tasks. The load imbalance between the tasks is clearly visible which leads to an inefficient use of system resources, which results in a longer runtime.

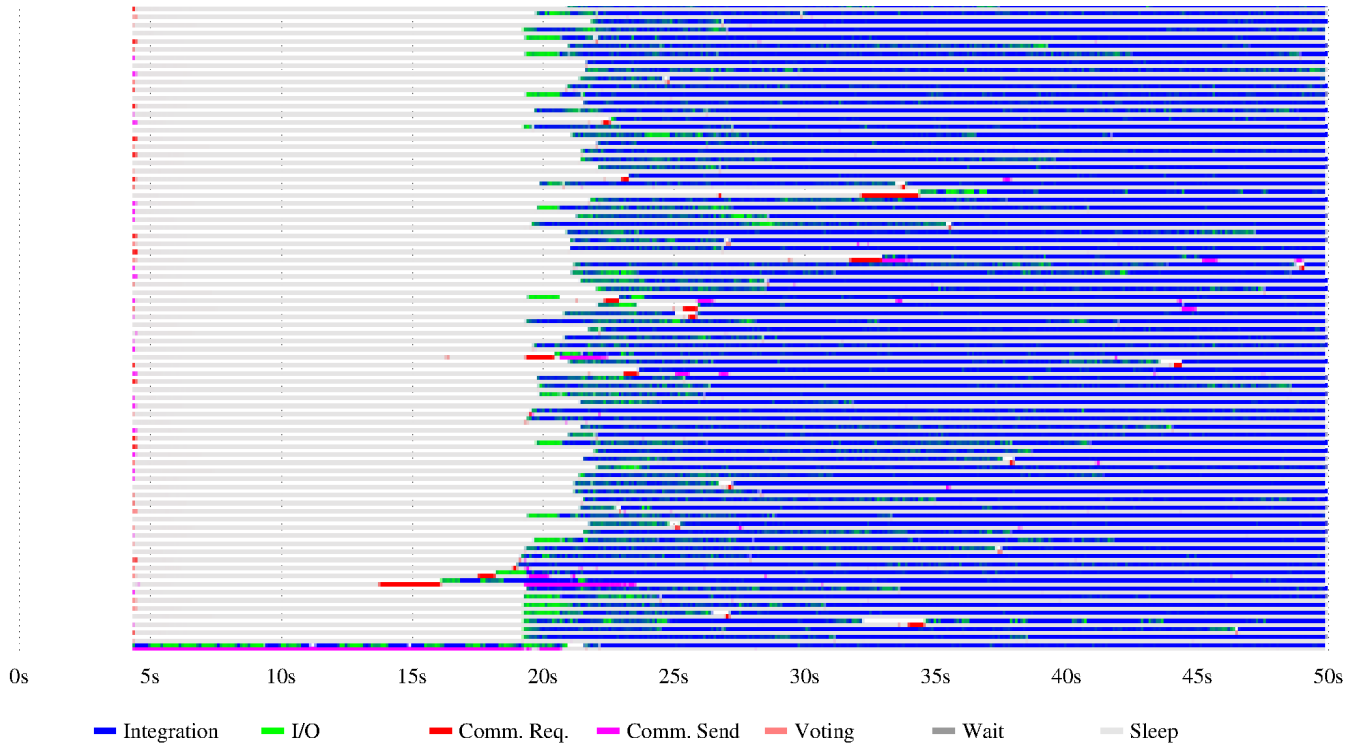

Fig. 7. Gantt chart of the work requesting distribution test. Only the first 50 seconds of the total runtime of 479 seconds is shown for only about one thirds of the tasks. After a few seconds of sending the work items, they are distributed between every task so the calculation can continue normally.

their integration. Only a few tasks need more time to find work; the last task needs about 30 seconds to find work. After that, the work items are distributed good enough so that the run looks like the test run where the work items were equally distributed at the beginning.

The efficiency of the distribution test is less than the test with equally distribution because of the additionally time needed to exchange the work items at the start of the test. But it only drops from 98.4% to 94.7% which is still better then the equally distribution parallelize-over-seeds algorithm.

6 Discussion

There is certainly an overhead involved when using the work requesting algorithm. Firstly there is the obvious overhead of communication to request work, send the work items and the voting process to determine termination. But additionally, there is presumably more I/O than in the parallelize-over-seeds algorithm because of the ordering of the particles. One task in the parallelize-over-seeds algorithm calculates a number of particles which all start near each other. So it is likely that they use the same parts of the dataset, leading to fewer blocks to load for each task. On the other hand, the work requesting algorithm shuffles the particles between the different tasks on each transfer of work items. So theoretically each task may calculate each particle, needing to load arbitrary parts of the dataset.

Both types of overhead get bigger on a higher number of tasks, as there is more communication of work items between the tasks.

This overhead can be seen in the results, most clearly on the efficiency graph (Figure 3), which shows decreasing efficiency for bigger numbers of tasks. But it can also be seen that the overhead is relatively small, less than 10% even on 1024 tasks. It is also much smaller than the overhead of the parallelize-over-seeds algorithm, which consists mostly of the waiting of the tasks after they finished their work for the other tasks to finish.

The reason for using work requesting instead of work stealing is the distributed memory of high performance computing clusters. Our prototype uses one addition thread to handle the communication required by the work requesting algorithm, this is needed to not interrupt the integration work. However dedicated hardware on current and future-generation supercomputers which allows remote direct memory access is more and more common. With this, a partitioned global address space can be used to access the work queues of other tasks without interrupting them, therefore, work stealing could be implemented effectively. One would expect better performance and scalability, as no co-operation is needed from the victim. There are however indications (cf. [8]) that this difference is only important if the work items require very little computation time and work remains unbalanced throughout the computation, which is not typical for an integral curve problem.

Overall, we conclude that work requesting can be a viable load balancing scheme for integral curve computation. We have observed it to scale quite well, at a very modest cost to efficiency. Naturally, our experiments cover a simplified implementation, and many optimizations would be possible to bring performance up to higher levels.

7 Conclusion

Integral curves, while being one of the most important techniques for the visualization of vector fields, are hard to calculated efficiently in a parallel setting due to strong inherent data dependency. General-purpose load balancing techniques like work requesting have been proven to be able to balance irregular loads well. In this paper, we have investigated how well work requesting performs on the integral curve computation problem. For this, we have implemented a prototype to compare it to a baseline approach. We have conducted tests to analyze scalability and efficiency, as well as understand how it performs even on a badly distributed work loads. Our results show very good efficiency and scalability up to at least 1,024 processes. Therefore, work requesting promises to be an effective technique to balance integral curve computation load. Regarding future work, we would like to investigate in how far dynamic, adaptive algorithms such as stream surface [4] or streakline computations, which generate new integral curves at runtime, can be scheduled efficiently with our approach.

Acknowledgments

This work was funded in part by the Marie Curie Actions within the EU FP7 Programme under grant #304099. The authors wish to thank Hank Childs for an in-depth discussion of this work. We also thank the AHRP for uncomplicated access to the Elwetritsch cluster which we used for early tests. For compute time on Hopper, were all the test results presented in this paper were obtained, we thank Lawrence Berkeley National Laboratory, which was supported by the U.S. Department of Energy under Contract No. DE-AC02-05CH11231.

References

[1] R. D. Blumofe and C. E. Leiserson. Scheduling multithreaded computations by work stealing. *Journal of the ACM*, 46(5):720–748, Sept. 1999.

[2] K. Bürger, J. Schneider, P. Kondratieva, J. Krüger, and R. Westermann. Interactive Visual Exploration of Unsteady 3D Flows. In *Proceedings of the Joint EG/IEEE VGTC Symp. on Visualization*, pages 251–258, 2007.

[3] M. Bußler, T. Rick, A. Kelle-Emden, B. Hentschel, and T. Kuhlen. Interactive Particle Tracing in Time-Varying Tetrahedral Grids. In *Proceedings of the Eurographics Symposium on Parallel Graphics and Visualization*, pages 71–80, 2011.

[4] D. Camp, H. Childs, C. Garth, and D. Pugmire. Parallel stream surface computation for large data sets. In *Proc. Large Data Analysis and Visualization '12*, pages 39–47, Oct. 2012.

[5] D. Camp, C. Garth, H. Childs, D. Pugmire, and K. I. Joy. Streamline Integration Using MPI-Hybrid Parallelism on a Large Multicore Architecture. *IEEE Transactions on Visualization and Computer Graphics*, 17(11):1702–1713, 2011.

[6] D. Camp, H. Krishnan, D. Pugmire, C. Garth, I. Johnson, E. W. Bethel, K. I. Joy, and H. Childs. GPU Acceleration of Particle Advection Workloads in a Parallel, Distributed Memory Setting. In *Proceedings of the Eurographics Symposium on Parallel Graphics and Visualization*, 2013.

[7] L. Chen and I. Fujishiro. Optimizing parallel performance of streamline visualization for large distributed flow datasets. In *IEEE Pacific Visualisation Symposium*, pages 87–94, Mar. 2008.

[8] J. Dinan, D. B. Larkins, P. Sadayappan, S. Krishnamoorthy, and J. Nieplocha. Scalable work stealing. In *SC*. ACM, Nov. 2009.

[9] A. Knüpfer, H. Brunst, J. Doleschal, M. Jurenz, M. Lieber, H. Mickler, M. S. Müller, and W. E. Nagel. The vampir performance analysis tool-set. In *Tools for High Performance Computing*, pages 139–155. 2008.

[10] J. Krüger, P. Kipfer, P. Kondratieva, and R. Westermann. A Particle System for Interactive Visualization of 3D Flows. *IEEE Transactions on Visualization and Computer Graphics*, 11(6):744–756, 2005.

[11] T. McLoughlin, R. S. Laramee, R. Peikert, F. H. Post, and M. Chen. Over Two Decades of Integration-Based, Geometric Flow Visualization. *Computer Graphics Forum*, 29(6):1807–1829, 2010.

[12] Message Passing Interface Forum. MPI2: A message passing interface standard. *High Performance Computing Applications*, 12(1–2):1–299, 1998.

[13] B. Nouanesengsy, T.-Y. Lee, K. Lu, H.-W. Shen, and T. Peterka. Parallel Particle Advection and FTLE Computation for Time-Varying Flow Fields. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–11, 2012.

[14] B. Nouanesengsy, T.-Y. Lee, and H.-W. Shen. Load-Balanced Parallel Streamline Generation on Large Scale Vector Fields. *IEEE Transactions on Visualization and Computer Graphics*, 17(12):1785–1794, 2011.

[15] T. Peterka, R. Ross, B. Nouanesengsy, T.-Y. Lee, H.-W. Shen, W. Kendall, and J. Huang. A Study of Parallel Particle Tracing for Steady-State and Time-Varying Flow Fields. In *Proceedings of the Parallel Distributed Processing Symposium (IPDPS),*, pages 580–591, 2011.

[16] D. Pugmire, H. Childs, C. Garth, S. Ahern, and G. Weber. Scalable Computation of Streamlines on Very Large Datasets. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2009.

[17] M. Schirski, C. Bischof, and T. Kuhlen. Interactive Particle Tracing on Tetrahedral Grids Using the GPU. In *Proceedings of Vision, Modeling, and Visualization (VMV) 2006*, pages 153–160, 2006.

[18] H. Yu, C. Wang, and K.-L. Ma. Parallel hierarchical visualization of large time-varying 3d vector fields. In *Supercomputing, 2007. SC'07. Proceedings of the 2007 ACM/IEEE Conference on*, pages 1–12. IEEE, 2007.