Visualizing, Measuring, and Tuning Adaptive MPI Parameters

Matthias Diener(✉), Sam White, and Laxmikant V. Kale

University of Illinois at Urbana-Champaign, Urbana, USA
{mdiener,white67,kale}@illinois.edu

Abstract. Adaptive MPI (AMPI) is an advanced MPI runtime environment that offers several features over traditional MPI runtimes, which can lead to a better utilization of the underlying hardware platform and therefore higher performance. These features are overdecomposition through virtualization, and load balancing via rank migration. Choosing which of these features to use, and finding the optimal parameters for them is a challenging task however, since different applications and systems may require different options. Furthermore, there is a lack of information about the impact of each option. In this paper, we present a new visualization of AMPI in its companion Projections tool, which depicts the operation of an MPI application and details the impact of the different AMPI features on its resource usage. We show how these visualizations can help to improve the efficiency and execution time of an MPI application. Applying optimizations indicated by the performance analysis to two MPI-based applications results in performance improvements of up 18% from overdecomposition and load balancing.

Keywords: MPI · Load balancing · AMPI · Migration · Overdecomposition

1 Introduction

Improving the performance of parallel applications that are based on the MPI programming model is an important aspect of High-Performance Computing. Compared to traditional MPI runtimes, Adaptive MPI (AMPI) [6] offers several advanced, unique features, the most important of which are: *overdecomposition through virtualization* and *load balancing through rank migration*. These features can be used to improve performance portability of MPI-based applications. AMPI itself is implemented on top of the Charm++ runtime system [1,10] and makes use of several of its features, including support for migration of threads, comprehensive scheduling and load balancing frameworks, and optimized communication within and between cluster nodes.

The key difference between AMPI and most other MPI implementations is that AMPI virtualizes ranks as lightweight, migratable user-level threads (instead of operating system processes). The Charm++ runtime system can

A. Bhatele et al. (Eds.): ESPT/VPA 2017/2018, LNCS 11027, pp. 219–230, 2019.
https://doi.org/10.1007/978-3-030-17872-7_13

schedule multiple virtual ranks per core based on message delivery, to overlap communication and computation and to enable a more fine-grained decomposition of work. This *overdecomposition* can also help with cache and NUMA locality, since smaller subdomains of a problem might fit more easily into caches.

The AMPI runtime also provides support for *migrating ranks* between address spaces at runtime, both within a cluster node and between separate nodes. This feature can be used for the purposes of load balancing or fault tolerance, among others. Charm++ contains many different load balancing strategies that can be selected by the user or automatically [18], resulting in substantial performance gains for many parallel applications [4,9].

These load balancing strategies are based on actual measurement of load information at runtime, and on migrating computations from heavily loaded to lightly loaded Processing Elements (PEs, Charm++'s terminology for OS processes). Figures 1 and 2 illustrate overdecomposition and rank migration in AMPI. The only changes necessary to existing MPI applications to run them on AMPI with virtualization and migration are related to privatizing global and static variables to AMPI's user-level threads [6]. All AMPI programs are valid MPI programs, besides any calls they might contain to AMPI's several extension APIs.

Using AMPI's high-level features efficiently is not straightforward, however. Users of MPI applications running on AMPI need to determine whether an application can benefit from each feature, as well as the optimal configuration (such as degree of overdecomposition and load balancing frequency) of each feature. Previously, the impact of these features could only be observed indirectly, by running an application with various parameters and observing its execution time. It was therefore difficult to determine the best configuration without extensive experiments, to understand application performance, as well as to explain the reasons for possible performance gains.

In this paper, we present the additions to AMPI and Projections that enable detailed performance analysis of applications running on AMPI, covering both normal MPI operations as well as AMPI's additions to the standard. With these

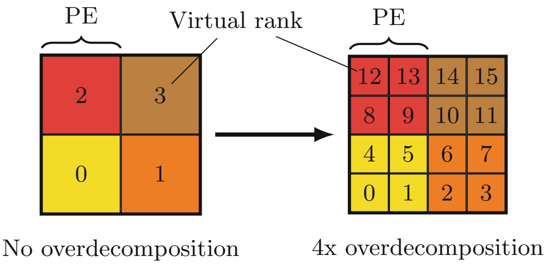

Fig. 1. Overdecomposition in AMPI. Colors indicate different PEs. The working set of a virtual rank in the *no overdecomposition* case might not fit into the cache, but it might fit in the *4x overdecomposition* case. (Color figure online)

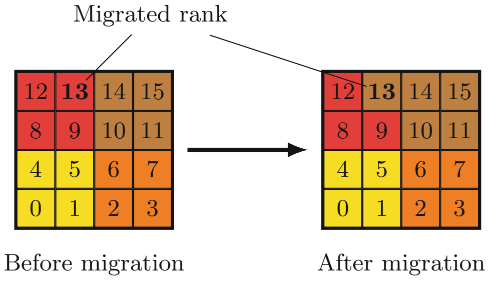

Fig. 2. Rank migration in AMPI. Colors indicate different PEs. Rank 13 is migrating from one PE to another. (Color figure online)

additions, it is possible to better understand the operation of an MPI-based application and its performance characteristics. Our tool can point out possible inefficiencies, their solutions, and can be used to evaluate and compare performance improvements.

In the second part of the paper (Sect. 3), we show how the information provided by AMPI and Projections can be used to optimize the performance of two MPI-based applications, LULESH [11,12] and PIC from the Intel Parallel Research Kernel suite [24]. Our results show that the performance analysis with the help of our additions to AMPI/Projections enabled us to achieve performance improvements of up to 18% from overdecomposition and load balancing. Furthermore, we show that performance gains are highly dependent on the characteristics of the application, such that different applications require using different AMPI features with different parameters.

2 Visualizing AMPI with Projections

This section briefly discusses how the operation of an MPI application running on top of AMPI is traced for visualization, and presents the main visualizations available to the application user in the Projections tool.

2.1 Implementation

Tracing and trace visualization in Charm++ and Projections is built around storing trace events in log files. Prior to version 6.8.0 of Charm++/AMPI, no special support for AMPI events was available, such that only events related to Charm++ were traced.

AMPI. In order to implement tracing of events in AMPI, we extended the support for *bracketed events* in the tracing framework in Charm++. Bracketed events are events that have a duration, that is, a starting time and end time. For every AMPI API function (standard MPI functions as well as AMPI extensions), an object is created on the stack as the first operation of that function. As part of the object's constructor, a time stamp of the function entry is stored. On function exit, this object is destroyed automatically, calculating the total time spent in the function and storing information about this event in the trace file. Information stored includes the event ID, function name, PE, virtual rank, and duration of the event. Previously, traces of AMPI programs only showed what task the AMPI implementation was executing at a given time on each core, providing no insight into what each virtual rank on a core was executing. Now, users can see what each virtual rank was doing at any given time.

Such an implementation via a stack-allocated object simplifies the support in AMPI, as well as seamlessly supporting nested events. The tracing framework itself is not limited to MPI, a user application can register and trace their own events in addition to the MPI functions. Furthermore, an application can also request more fine-grained traces by dynamically enabling and disabling tracing at runtime, via the `AMPI_Trace_begin()` and `AMPI_Trace_end()` functions.

Enabling tracing in Charm++ and AMPI applications has generally a negligible execution time overhead. For the applications discussed in this paper, the measured overhead was typically less than 3% of the total execution time. Trace files are kept in memory and are flushed to disk periodically and at the end of execution in a compressed format.

Projections. The Projections tool reads and evaluates the trace files after the execution of a Charm++ or AMPI application. We extended it with support for displaying virtual ranks for bracketed events, such that a user can see which rank has executed which MPI function. Furthermore, support was added to determine when and where virtual ranks are migrated, by showing the virtual rank numbers for traced events. As in Charm++ traces, MPI functions are grouped by color, such that it is easy to follow the operation of collective functions.

2.2 Visualizations

In the example in this section, we use an MPI application running on four Processing Elements (PEs) and eight virtual ranks (VPs) to illustrate the visualizations. Figure 3 depicts the visualization before the extensions described in this paper were applied, as presented in the original AMPI paper [6]. In the figure, a user can see that the application is running on four PEs and the percentage of time this PE was busy (that is, not blocked while waiting for communication, for example). This percentage is shown below each PE (left number in the parentheses). Furthermore, the figure illustrates at which times each PE was idle (in white) and busy (in red). Not presented in this figure are the virtual ranks of the application, and which operations they are performing.

Figures 4 and 5 depict the visualizations with the changes described in this paper. In addition to the information presented before, now the virtual ranks and the PE they are executing on are shown (two virtual ranks per PE in this example), as well as the operations the ranks perform, giving a detailed view of

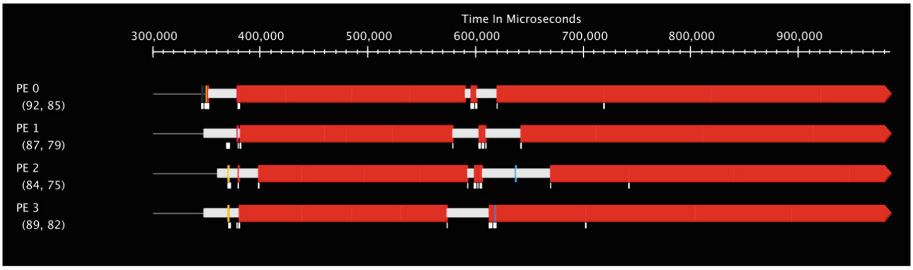

Fig. 3. Previous visualization of AMPI in Projections, as presented in the original AMPI paper [6]. The x-axis depicts time, while the y-axis shows the various processing elements (PE). Visible are the four processing elements, busy percentages (left value below each PE label), idle times (in white), and busy times (in red). Not visible are virtual ranks (two per PE), rank migrations, and which operation each rank is performing. (Color figure online)

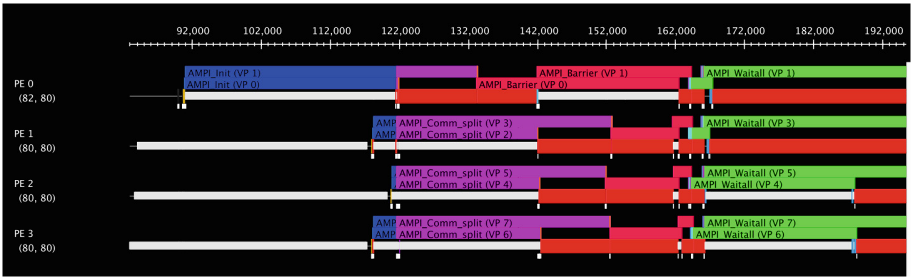

Fig. 4. New visualization of AMPI. In addition to the information shown in Fig. 3, virtual ranks (VPs) are depicted (including on which PE they are executing), as well as the operation performed by each rank.

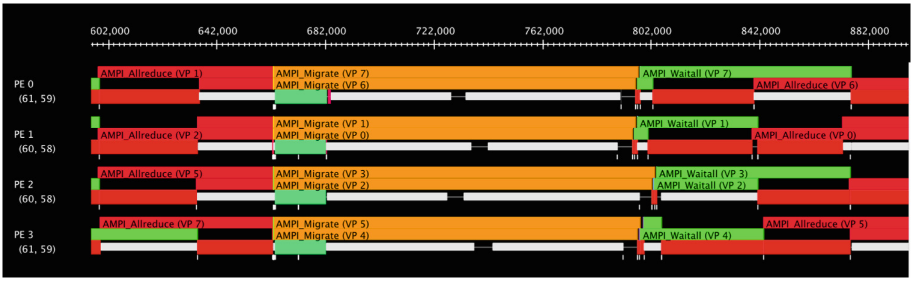

Fig. 5. Visualizing migrations in AMPI. The MPI extension *AMPI_Migrate()* shows where each rank is migrated. For example, VP 1 is migrated from PE 0 to PE 1.

an application's behavior. For example, in Fig. 4, it is possible to see that at the time between 142 ms and 162 ms, PE 0 was idle since both virtual ranks running on that PE (VP 0 and VP 1) were waiting in an `MPI_Barrier`. Starting at about 167 ms, PE 0 is busy with the execution of VP 0, while VP 1 is performing an `MPI_Waitall` operation. This shows how the overdecomposition can help reduce idle time.

In Fig. 5, the operation of a migration operation in AMPI is depicted. By looking the `AMPI_Migrate` event, a user can see which virtual ranks were migrated, and to which PE they were migrated to. In the example shown, VP 1 is migrated from PE 0 to PE 1.

Additional information that is provided by Projections, but not shown in the figures, are statistics related to the number of different events and the time spent for each event, among others.

3 Application Case Studies

This section presents two case studies using two different MPI-based applications in order to demonstrate how the visualizations presented in the previous section can help users and developers of MPI applications to optimize application performance and performance portability.

In this section, we discuss the overall load imbalance of an application using the average busy time and the *percent imbalance* metric λ [21], calculated over the busy time of all PEs using the following equation:

$$\lambda = \left(\frac{max(L)}{avg(L)} - 1 \right) \times 100\% \tag{1}$$

In the equation, L is a vector of the busy times of all PEs. If $\lambda = 0$, the application is perfectly balanced, while higher values of λ indicate increasing amounts of imbalance. The maximum value of λ with 8 PEs and possible values of 0–100 is 700%.

To keep the presentation of the visualizations at a reasonable size, we restrict them in this section to 8 PEs. Results are qualitatively similar to much higher numbers of PEs for both applications presented here.

For the performance experiments, we execute the applications on a system with an Intel Xeon E5-2680 v2 CPU (10-core, 2.8 GHz, SMT disabled) and 64 GByte of DDR3 main memory. The software environment consists of CentOS 7 with Linux kernel 2.6.32, gcc 4.8.2, and Charm++/AMPI 6.8.0.

3.1 LULESH

LULESH[1] is an LLNL proxy application for unstructured Lagrangian-Eulerian shock hydrodynamics [11,12]. We use the MPI implementation of LULESH 2.0 in the experiments.

[1] https://codesign.llnl.gov/lulesh.php.

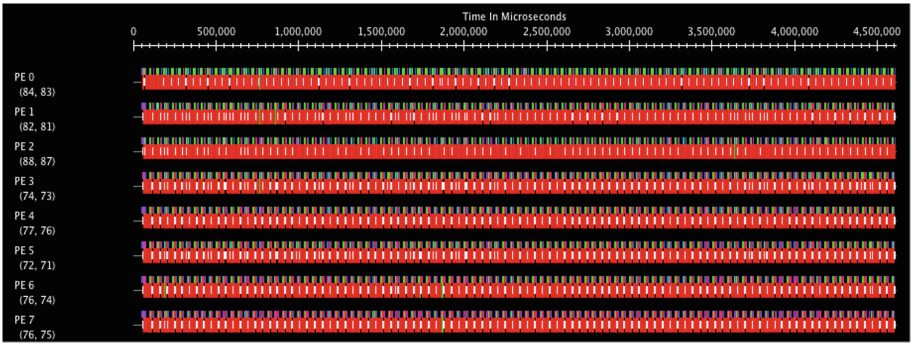

Fig. 6. Baseline execution of LULESH with neither overdecomposition nor load balancing.

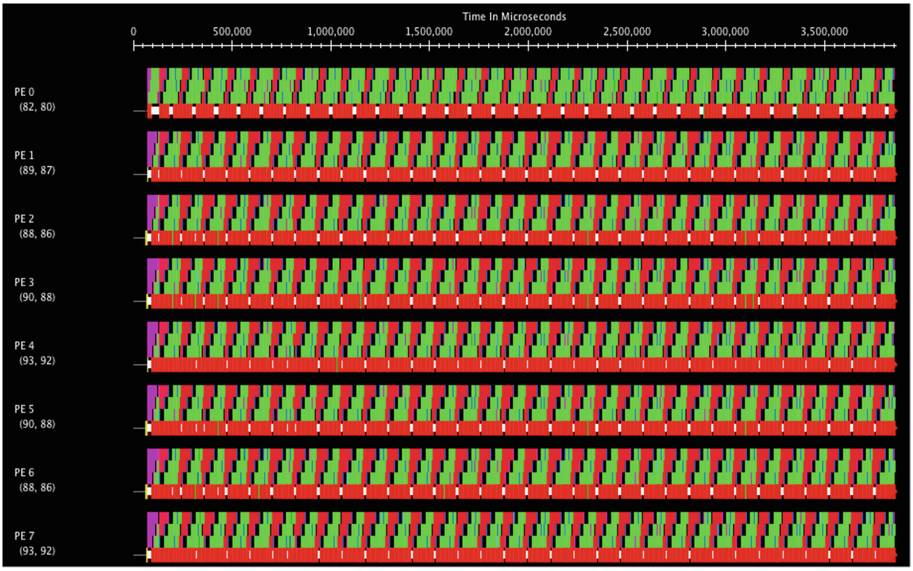

Fig. 7. Execution of LULESH with 3.4x overdecomposition (8 PEs, 27 virtual ranks) and no load balancing.

Figure 6 depicts the operation of LULESH with 8 PEs/ranks, no overdecomposition and no load balancing. As can be seen from the figure, the application is not imbalanced, with similar busy times and load distribution among all PEs. The average busy percentage is 78.6%, with an imbalance of $\lambda = 11.9\%$. Due to the low busy percentage, this application may benefit from overdecomposition. On the other hand, load balancing appears not to be profitable due to the low imbalance.

Figure 7 shows the performance graph of LULESH with a 3.4x overdecomposition (27 virtual ranks running on 8 PEs). As we expected, the busy time of all PEs is increased substantially in this scenario, reaching an average of 89.1%, while also improving the load balance of the application slightly ($\lambda = 4.3\%$).

The impact of these improvements can be seen on the execution time, which was reduced from 4.61 s in the baseline experiment to 3.85 s with overdecomposition (∼16% improvement).

3.2 Particle-in-cell

The *Particle-in-cell (PIC)*[2] application is part of Intel's Parallel Research Kernels [24]. We used version 2.17 of the AMPI implementation of PIC.

Figure 8 shows the performance behavior of the PIC application baseline, with 8 PEs/ranks and no load balancing. Several things need to be noted here. First of all, the application is substantially imbalanced. About half of the PEs have a significantly lower busy time than the other half, leading to an overall imbalance of $\lambda = 22.5\%$. Furthermore, since some of the PEs are idle for large amounts of time, the overall busy time is only 75.6%.

The first natural step to fix this behavior is to balance the load between the PEs. For this, we use AMPI's load balancing feature, specifically the RefineLB load balancer mechanism, which has shown good load balancing results with a reasonable overhead [2]. The result of this experiment is presented in Fig. 9. Since overdecomposition is required for load balancing, we selected the smallest reasonable degree of overdecomposition (2x, 16 virtual ranks on 8 PEs) for this experiment. Note that in order to reduce the size of the figures, we are not showing the individual virtual ranks in Figs. 9 and 10.

As can be seen in Fig. 9, the RefineLB load balancer is able to balance the load among the PEs successfully, resulting in an overall imbalance of only $\lambda = 7.3\%$. However, although the work is better distributed, the average busy time (77.4%) increases only slightly compared to the baseline execution, despite the slightly higher overdecomposition. Therefore, we can not expect significant performance improvements compared to the baseline. This is confirmed by the measurement of the execution time, which is reduced only from 3.96 s in the baseline to 3.94 s with load balancing.

The relatively high idle time of the load balanced version indicates that this application can benefit from overdecomposition in addition to load balancing. This intuition is verified with an experiment that uses a 6x overdecomposition (48 virtual ranks on 8 PEs) in addition to RefineLB. The results of this experiment is shown in Fig. 10. Here, we can see that busy time has increased drastically, with an average of 92.4%. Furthermore, the application is also more balanced ($\lambda = 1.7\%$). These improvements lead to a total execution time of 3.26 s, about 18% less than in the baseline version of PIC.

[2] https://github.com/ParRes/Kernels.

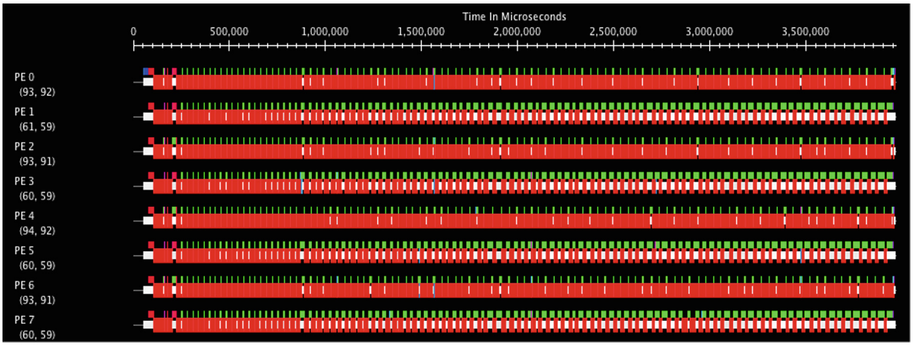

Fig. 8. Baseline execution of PIC with neither overdecomposition nor load balancing.

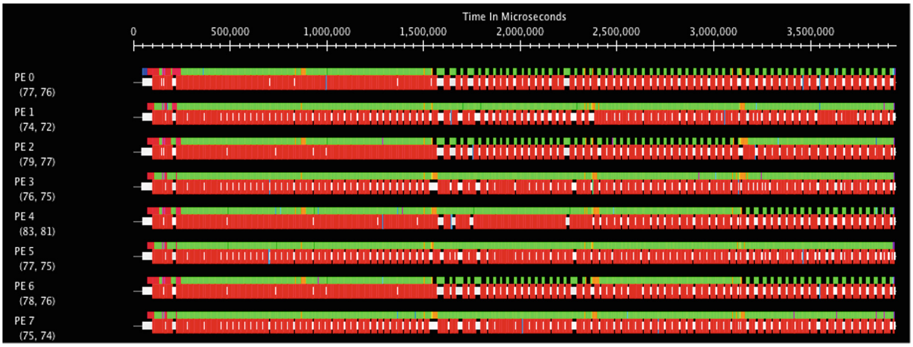

Fig. 9. Execution of PIC with load balancing (RefineLB) and 2x overdecomposition (8 PEs, 16 virtual ranks).

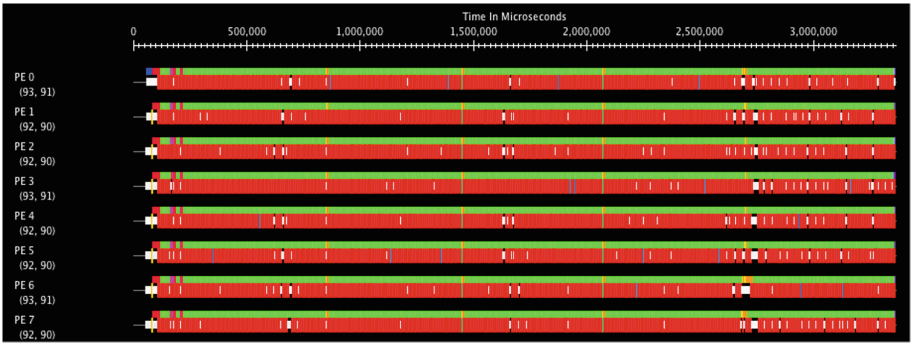

Fig. 10. Execution of PIC with load balancing (RefineLB) and 6x overdecomposition (8 PEs, 48 virtual ranks).

4 Related Work

Several prior tools exist to help with visualizing and understanding MPI application performance. These tools include Totalview [5], Allinea Map and DDT [17], Vampir [14]/Vampirtrace [20], Score-P [15], the HPCToolkit [3], Jumpshot [26], and Marmot [16]. Some of these tools provide visualizations of an application's MPI behavior that are very similar to the visualizations discussed in this paper.

Many proposed techniques exist for monitoring communication in MPI applications [23,25,27]. Tracing itself, as well as storing and analyzing large trace files, is a significant challenge [27]. Since a tracing API is directly integrated in Charm++/AMPI, the tracing overhead can be substantially lower than in external tools that rely on overriding particular MPI functions.

Other tools perform automatic detection of inefficiencies in certain MPI functions (such as send and receive) [22]. However, as these tools are not aware of AMPI's features that go beyond the MPI standard, their applicability in the context of the AMPI runtime is limited. Particularly, they can generally not be used for overdecomposition or migration, as they have no knowledge of virtual ranks.

Many performance analysis tools for MPI use the Profiling MPI (PMPI) standard [13,19], which provides a coarse-grained way to override standard MPI functions with custom versions that can be used for tracing, analysis, and visualization. More recently, the MPI_T interface [7,8] was added to the MPI standard [19]. It allows more fine-grained access to performance counters provided by the environment. Currently, AMPI does not support PMPI or MPI_T, but an implementation is planned for the near future. With such support, AMPI could expose information about overdecomposition and migrations to other external tools.

5 Conclusions

Adapting MPI applications to the underlying hardware platform and guaranteeing performance portability on different systems is a challenging task. In this context, the Adaptive MPI (AMPI) runtime provides several features that can help with this task, the two most important of which are *overdecomposition through virtualization* and *load balancing through rank migration*. Correct usage of these features requires a deep understanding of the application performance, as well as information about inefficient behavior displayed by the application.

In this paper, we presented extensions to the Projections tool to help with the performance analysis of applications running on AMPI. We added tracing capabilities to AMPI, covering standard MPI functions and AMPI's extensions, and added their visualization to Projections. Furthermore, we extended AMPI and Projections to support visualization of virtual ranks as well as rank migrations at runtime. With our extensions, Projections can be used to understand application behavior, point out possible inefficiencies and their solutions, and evaluate improvements in load balance, overdecomposition, and performance. We applied this analysis to two MPI-based applications, and achieved improvements of 16%–18% with overdecomposition and/or load balancing.

The changes discussed in this paper have been integrated into version 6.8.0 of Charm++/AMPI, and are available online[3]. Projections is available at the same location. For the future, we intend to integrate support for PMPI and MPI_T into AMPI in order to better support traditional performance analysis tools. Furthermore, we want to improve how rank migrations are displayed in Projections, and implement automatic suggestions for performance improvements in AMPI and Projections.

Acknowledgments. This paper is based in part upon work supported by the Department of Energy, National Nuclear Security Administration, under Award Number DE-NA0002374.

References

1. Acun, B., et al.: Parallel programming with migratable objects: Charm++ in practice. In: SC (2014). https://doi.org/10.1109/SC.2014.58
2. Acun, B., Kale, L.V.: Mitigating processor variation through dynamic load balancing. In: 2016 IEEE International Parallel and Distributed Processing Symposium Workshops, pp. 1073–1076. IEEE (2016)
3. Adhianto, L., et al.: HPCToolkit: tools for performance analysis of optimized parallel programs. Concurr. Comput.: Pract. Exp. **22**(6), 685–701 (2010). https://doi.org/10.1002/cpe.1553
4. Bhandarkar, M., Kalé, L.V., de Sturler, E., Hoeflinger, J.: Adaptive load balancing for MPI programs. In: Alexandrov, V.N., Dongarra, J.J., Juliano, B.A., Renner, R.S., Tan, C.J.K. (eds.) ICCS 2001. LNCS, vol. 2074, pp. 108–117. Springer, Heidelberg (2001). https://doi.org/10.1007/3-540-45718-6_13
5. Gottbrath, C.: Automation assisted debugging on the Cray with TotalView. In: Proceedings of Cray User Group (2011)
6. Huang, C., Lawlor, O., Kalé, L.V.: Adaptive MPI. In: Rauchwerger, L. (ed.) LCPC 2003. LNCS, vol. 2958, pp. 306–322. Springer, Heidelberg (2004). https://doi.org/10.1007/978-3-540-24644-2_20
7. Islam, T., Mohror, K., Schulz, M.: Exploring the capabilities of the new MPI_T interface. In: Proceedings of the 21st European MPI Users' Group Meeting, p. 91. ACM (2014)
8. Islam, T., Mohror, K., Schulz, M.: Exploring the MPI tool information interface: features and capabilities. Int. J. High Perform. Comput. Appl., pp. 212–222. (2016). https://doi.org/10.1177/1094342015600507
9. Jeannot, E., Meneses, E., Mercier, G., Tessier, F., Zheng, G.: Communication and topology-aware load balancing in Charm++ with TreeMatch. In: 2013 IEEE International Conference on Cluster Computing, CLUSTER, pp. 1–8. IEEE (2013)
10. Kale, L.V., Krishnan, S.: CHARM++: a portable concurrent object oriented system based on C++. In: Conference on Object-Oriented Programming Systems, Languages, and Applications, OOPSLA, pp. 91–108 (1993)
11. Karlin, I., et al.: Exploring traditional and emerging parallel programming models using a proxy application. In: 27th IEEE International Parallel & Distributed Processing Symposium, IEEE IPDPS 2013, Boston, USA, May 2013

[3] https://charm.cs.illinois.edu/software.

12. Karlin, I., Keasler, J., Neely, R.: Lulesh 2.0 updates and changes. Technical report LLNL-TR-641973, August 2013
13. Karrels, E., Lusk, E.: Performance analysis of MPI programs. In: Environments and Tools for Parallel Scientific Computing, pp. 195–200 (1994)
14. Knüpfer, A., et al.: The Vampir performance analysis tool-set. In: Resch, M., Keller, R., Himmler, V., Krammer, B., Schulz, A. (eds.) Tools for High Performance Computing, pp. 139–155. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-68564-7_9
15. Knüpfer, A., et al.: Score-P: a joint performance measurement run-time infrastructure for periscope, Scalasca, TAU, and Vampir. In: Brunst, H., Müller, M., Nagel, W., Resch, M. (eds.) Tools for High Performance Computing, pp. 79–91. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-31476-6_7
16. Krammer, B., Bidmon, K., Müller, M.S., Resch, M.M.: MARMOT: an MPI analysis and checking tool. In: Advances in Parallel Computing, vol. 13, pp. 493–500 (2004)
17. Lecomber, D., Wohlschlegel, P.: Debugging at scale with Allinea DDT. In: Cheptsov, A., Brinkmann, S., Gracia, J., Resch, M., Nagel, W. (eds.) Tools for High Performance Computing, pp. 3–12. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-37349-7_1
18. Menon, H., Chandrasekar, K., Kale, L.V.: POSTER: automated load balancer selection based on application characteristics. In: Proceedings of the 22nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, pp. 447–448. ACM (2017)
19. Message Passing Interface Forum: MPI: A Message-Passing Interface Standard (Version 3.0). Technical report (2012)
20. Müller, M.S., et al.: Developing scalable applications with Vampir, VampirServer and VampirTrace. In: PARCO, vol. 15, pp. 637–644 (2007)
21. Pearce, O., Gamblin, T., de Supinski, B.R., Schulz, M., Amato, N.M.: Quantifying the effectiveness of load balance algorithms. In: ACM International Conference on Supercomputing, ICS, pp. 185–194 (2012). https://doi.org/10.1145/2304576.2304601
22. Vetter, J.: Performance analysis of distributed applications using automatic classification of communication inefficiencies. In: Proceedings of the 14th International Conference on Supercomputing, pp. 245–254. ACM (2000)
23. Vetter, J.: Dynamic statistical profiling of communication activity in distributed applications. In: Proceedings of the 2002 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems, SIGMETRICS 2002, pp. 240–250. ACM, New York (2002). https://doi.org/10.1145/511334.511364
24. Van der Wijngaart, R.F., Mattson, T.G.: The parallel research kernels. In: 2014 IEEE High Performance Extreme Computing Conference, HPEC, pp. 1–6. IEEE (2014)
25. Wu, C.E., et al.: From trace generation to visualization: a performance framework for distributed parallel systems. In: Proceedings of the 2000 ACM/IEEE Conference on Supercomputing, SC 2000. IEEE Computer Society, Washington, DC (2000). http://dl.acm.org/citation.cfm?id=370049.370458
26. Zaki, O., Lusk, E., Gropp, W., Swider, D.: Toward scalable performance visualization with Jumpshot. Int. J. High Perform. Comput. Appl. **13**(3), 277–288 (1999)
27. Zhai, J., Sheng, T., He, J.: Efficiently acquiring communication traces for large-scale parallel applications. IEEE Trans. Parallel Distrib. Syst. (TPDS) **22**(11), 1862–1870 (2011). https://doi.org/10.1109/TPDS.2011.49