Adaptive Loop Scheduling with Charm++ to Improve Performance of Scientific Applications

Vivek Kale
University of Southern California
Marina del Rey, California

Harshitha Menon
Lawrence Livermore National Laboratory
Livermore, California

Karthik Senthil
University of Illinois at Urbana-Champaign
Urbana, Illinois

KEYWORDS

Load Balancing, Intelligent Parallel Runtime Systems, Multi-core

1 INTRODUCTION

Supercomputers today employ a large number of cores on each node, often with additional symmetric multi-threading on each core [1, 13]. At the same time, load imbalance is emerging as a performance issue. The issue is in part caused by static and dynamic speed variation across cores [7] and in part caused by sophisticated applications with dynamic adaptations such as mesh refinements [5].

Charm++ [8] has been highly effective at providing inter-node dynamic load balancing using an intelligent runtime system. However, modern multi-core nodes present new challenges and opportunities for Charm++ applications.

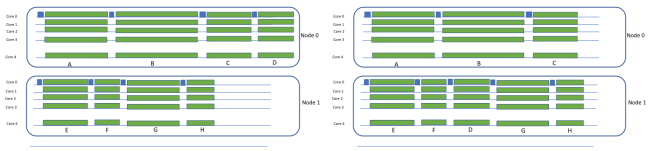
Two issues motivate the work presented in this poster:

(1) *The challenge of the cost of over-decomposition.*
(2) *The challenge and opportunity of exploiting multi-core nodes to mitigate imbalance.*

The baseline existing technique for load balancing in the Charm++ Runtime System (RTS) decomposes computation into a large number of chares so that there are tens of objects per core or hardware thread. It is arguable that such a high degree of over-decomposition may be infeasible or prohibitively expensive, especially with modern nodes with a large number of cores each. The alternative technique that we expand upon in this poster is to decompose the problem into a much smaller number of chares, so that there are only tens of chares per *node*, and to parallelize the execution of each chare's work across all the cores of a node using Charm++'s CkLoop library (loop-level parallelism). We will refer to this approach as *node-level over-decomposition*. We assume this approach as a *given* and focus the poster on how to effectively realize it. In addition, inter-node load balancing is applied periodically based on persistent behavior of objects.

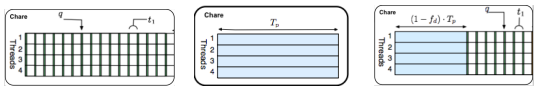
2 ADAPTIVE LOOP SCHEDULING WITH CHARM++

To realize the above approach as shown in Figures 1a and 1b, we must ensure that the work of each chare is spread equally across all cores without loss of efficiency. Our key idea is to use a tunable low-overhead loop scheduling strategy within each chare and then adjust Charm++'s load balancer's parameters. To formulate our technique for doing so, we define δ as the expected cost of load imbalance on a node to an application, α as the ratio of δ to the cost of an application, and $load_i$ as the load on the i^{th} core of a node on an arbitrary timestep.



(a) (b)

Figure 1: Application program's profile with intra-node load balancing (a) *without* and (b) *with* inter-node load balancing.



(a) Dynamic scheduling. **(b) Static scheduling.** **(c) Mixed static/dynamic scheduling.**

Figure 2: Motivation for mixed static/dynamic scheduling.

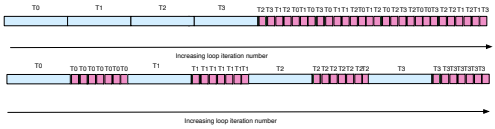


Figure 3: Optimization of mixed static/dynamic scheduling in the context of a Charm++ program.

Using ***static scheduling*** of loop iterations won't cause threads to incur overhead [3]. However, if the computational work of the iterations is non-uniform and/or the hardware creates imbalances because of noise [7, 12], using static scheduling will degrade performance because it will not complete until all cores finish their statically assigned iterations. Using ***dynamic scheduling***, as shown in Figure 2a, evens out load imbalances up to the quantization created by the loop's chunk size. However, this scheme suffers from two costs: 1) synchronization overhead due to dynamic assignment [2] and 2) loss of spatial locality due to iterations executed by a core in time sequence being noncontiguous [10].

The key idea of ***mixed static/dynamic scheduling*** is to realize that every iteration (or chunk) doesn't need to be scheduled dynamically for effective dynamic load balancing. Only a fraction of iterations are scheduled dynamically and the remaining ones can be statically assigned to each core as a single chunk. We define the portion of dynamically scheduled work as the *dynamic fraction* (f_d) and the remaining static portion as the *static fraction* (f_s = 1.0 - f_d). Across all chares, if the maximum imbalance expected across cores a chare uses, i.e., $\frac{max\ load_i}{average\ load_i}$, is $1 + \alpha$, then $\frac{1}{1+\alpha}$ fraction of work can be executed with static scheduling, thus reducing the cost

of dynamic scheduling significantly. Of course, doing this requires heuristics for a good estimate of the value of α to choose the ideal static fraction [9].

A way to optimize the mixed static/dynamic scheduling strategy is to stagger loop iterations [10] so that a thread's dynamically scheduled iterations are adjacent to its statically scheduled iterations as shown in Figure 3. This improves spatial locality in the common case when no dynamic imbalance arises during execution of the loop.

We refer to this scheduling strategy as *adaptive loop scheduling* and use a parametrized version of it to schedule loop iterations of a chare to cores. The best-performing collection of scheduler parameter values is the best-performing one, on average, for each chare. An optimal collection of scheduler parameter values, in isolation, isn't necessarily the best-performing, given an arbitrary collection of parameter values for the load balancer. Thus, our technique is to search for the collection of parameter values for the scheduler together with those for the load balancer that obtains best performance.

3 MODIFIED CHARM++ RTS

Usage of Library: Figure 4 shows the user code with the adapted CkLoop library using the function `CkParLoop()` for the computational region of the program. The user creates a kernel for the computation region and replaces the region with a call to `CkParLoop()` having the name of the kernel for the computation region as a parameter to the function. Here, the user calls the function on thread 0 providing the kernel to be called, the range of iterations of the loop and the number of chunks used in the loop. The user specifies a loop history variable that has a field for the static fraction within it. The lines of code in the user code with the library is 3% more than the original user code.

```
#include "charm++.h"
#include "CkLoopAPI.h"
void doDotProduct()
{
  float r = 0.0; int numberOfChunks= (probSize/numElems)/chunkSize;
  float* params[2]; params[0] = a; params[1] = b;
  CkParLoop(lh, dotP_chunked, 2, params, numberOfChunks, 0, (probSize/numElems),
          1, &r, CKLOOP_FLOAT_SUM);
  CkCallback cb(CkReductionTarget(Main, printResult), mainProxy);
  contribute(sizeof(float), &r, CkReduction::sum_float, cb);
}
```

Figure 4: User code for dot product implemented in Charm++ with `CkParLoop()`.

Implementation of Library: When `CkParLoop()` is invoked on core 0, our scheduler enqueues a message in every core's queue describing the loop. Each core enqueues its dynamic iterations into its own stealable task queue and then executes its static chunk by calling the user-supplied kernel function. It then executes dynamic tasks from its own stealable queue and then randomly steals from other cores' queues. The synchronization for loop termination and handling of reduction variables is also part of the implementation.

Charm++'s scheme to allocate chares, i.e., objects, to cores was modified so that objects are initially assigned to core 0 of each node. The dynamic load balancer was modified to measure the loads of objects correctly in presence of the scheduler and to ensure that objects weren't re-assigned to cores other than core 0.

4 EVALUATION

We now evaluate how our approach of combining the adaptive loop scheduling strategy and Charm++ load balancing performs. We consider the Particle-in-Cell, or PIC, code from the Parallel Research Kernels [4] implemented in Charm++ and adapted to incorporate CkParLoop. The application code performs a particle calculation followed by a loosely synchronous communication. Since particles migrate across chares, the code has significant dynamic imbalance. When Charm++ load balancing is used across all cores of a node, i.e., when *Charm++-only* is used, we know that load isn't balanced perfectly across the cores because of quantization, i.e., because there are only a few chares on each core. We confirmed this by timing the execution of a Charm++-only version of PIC.

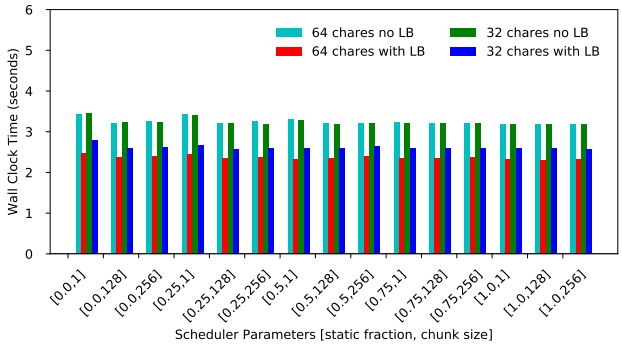


Figure 5: Impact of varying scheduler parameters and load balancer parameters when running PIC on Blue Waters.

We vary the scheduler parameters and Charm++ RTS parameters, without and with Charm++ load balancing with a varying number of chares. Figure 5 shows results for the experiment running on 4 nodes of the Blue Waters supercomputer using 100 timesteps. The frequency of Charm++ load balancer calls can be reduced since within-node load balancing mitigates imbalances that arise [6, 11]. The scheduler parameters, i.e., the number of chunks and static fraction, are varied to obtain the execution times shown in the plot. From these results, we see that using a modified Charm++ inter-node load balancing strategy (using 32 chares instead of 64 chares and with low-frequency load balancing) improves performance over the case of using fully dynamic scheduling without any load balancing strategy by 17.20%. We expect that the performance gains will be higher for runs on a larger number of nodes.

5 CONCLUSION

We have explained the need to combine low-overhead loop scheduling strategies and Charm++'s inter-node load balancing strategies. We integrated our scheduling strategies in Charm++ via its CkParLoop construct. Using this technique, we demonstrate performance improvement of 17.2% in the PIC code run on 4 nodes of Blue Waters. For future work, we will 1) integrate our technique with an OpenMP library such as LLVM's OpenMP library and 2) explore a more sophisticated and intelligent set of scheduling and load balancing strategies based on a broader set of applications.

REFERENCES

[1] 2012. Intel Delivers New Architecture for Discovery with Intel Xeon Phi Coprocessors. https://newsroom.intel.com/news-releases/intel-delivers-new-architecture-for-discovery-with-intel-xeon-phi-coprocessors/. (Nov. 2012). Accessed: 2017-03-30.

[2] Eduard Ayguade, Bob Blainey, Ro Duran, Jesus Labarta, Francisco Martinez, Xavier Martorell, and Raul Silvera. 2003. Is the Schedule Clause Really Necessary in OpenMP?. In *Proceedings of the International Workshop on OpenMP Applications and Tools 2003, volume 2716 of Lecture Notes in Computer Science*. Toronto, Canada, 69–83.

[3] Leonardo Dagum and Ramesh Menon. 1998. OpenMP: An Industry-Standard API for Shared-Memory Programming. *IEEE Computational Science & Engineering* 5, 1 (January-March 1998).

[4] Evangelos Georganas, Rob F. Van der Wijngaart, and Timothy G. Mattson. 2016. Design and Implementation of a Parallel Research Kernel for Assessing Dynamic Load-Balancing Capabilities. *Parallel and Distributed Processing Symposium, International*. https://doi.org/10.1109/IPDPS.2016.65

[5] Brian T.N. Gunney, Andrew M. Wissink, and David A. Hysom. 2006. Parallel Clustering Algorithms for Structured AMR. *J. Parallel and Distrib. Comput.* 66, 11 (2006), 1419 – 1430. https://doi.org/10.1016/j.jpdc.2006.03.011

[6] Harshitha Gopalakrishnan. 2016. *Adaptive Load Balancing for HPC Applications*. Ph.D. Dissertation.

[7] T. Hoefler, T. Schneider, and A. Lumsdaine. 2010. Characterizing the Influence of System Noise on Large-Scale Applications by Simulation. In *International Conference for High Performance Computing, Networking, Storage and Analysis (SC'10)*. New Orleans, LA, USA.

[8] L.V. Kalé and S. Krishnan. 1993. CHARM++: A Portable Concurrent Object Oriented System Based on C++. In *Proceedings of OOPSLA'93*, A. Paepcke (Ed.). ACM Press, 91–108.

[9] Vivek Kale, Simplice Donfack, Laura Grigori, and William D. Gropp. 2014. Lightweight Scheduling for Balancing the Tradeoff Between Load Balance and Locality. (2014).

[10] Vivek Kale, Amanda Peters Randles, and William D. Gropp. 2014. Locality-Optimized Scheduling for Improved Load Balancing on SMPs, In Proceedings of the 21st European MPI Users' Group Meeting Conference on Recent Advances in the Message Passing Interface. *EuroMPI/ASIA '14* 0, 1063–1074.

[11] Harshitha Menon, Seonmyeong Bak, Phil Miller, Sam White Nitin Bhat, and Laxmikant Kale. 2016. *Handling Transient and Persistent Imbalance Together in Distributed and Shared Memory*. Technical Report.

[12] Alessandro Morari, Roberto Gioiosa, Robert Wisniewski, Francisco J. Cazorla, and Mateo Valero. 2011. A Quantitative Analysis of OS Noise. In *Proceedings of the IEEE International Parallel and Distributed Processing Symposium*. Anchorage, AK, USA.

[13] Scott Vetter. 2013. Architecture of the IBM POWER7+ Technology-Based IBM Power 750 and IBM Power 760. http://www.redbooks.ibm.com/Redbooks.nsf/RedbookAbstracts/tips0972.html. (Feb. 2013).