Integrating OpenMP into the Charm++ Programming Model

Seonmyeong Bak
University of Illinois at Urbana-Champaign
sbak5@illinois.edu

Harshitha Menon
Lawrence Livermore National Laboratory
gopalakrishn1@llnl.gov

Sam White
University of Illinois at Urbana-Champaign
white67@illinois.edu

Matthias Diener
University of Illinois at Urbana-Champaign
mdiener@illinois.edu

Laxmikant Kale
University of Illinois at Urbana-Champaign
kale@illinois.edu

ABSTRACT

The recent trend of rapid increase in the number of cores per chip has resulted in vast amounts of on-node parallelism. These high core counts result in hardware variability that introduces imbalance. Applications are also becoming more complex themselves, resulting in dynamic load imbalance. Load imbalance of any kind can result in loss of performance and decrease in system utilization. In this paper, we propose a new integrated runtime system that adds OpenMP shared-memory parallelism to the Charm++ distributed programming model to improve load balancing on distributed systems. Our proposal utilizes an infrequent periodic assignment of work to cores based on load measurement, in combination with tasks created via OpenMP's parallel loop construct from each core to handle load imbalance. We demonstrate the benefits of using this integrated runtime system on the LLNL ASC proxy application Lassen, achieving speedups of 50% over runs without any load balancing and 10% over existing distributed-memory-only balancing schemes in Charm++.

CCS CONCEPTS

• **Computer systems organization → Multicore architectures**; **Distributed architectures**; • **Software and its engineering → Runtime environments**;

KEYWORDS

Charm++, OpenMP, Load Balancing

ACM Reference Format:
Seonmyeong Bak, Harshitha Menon, Sam White, Matthias Diener, and Laxmikant Kale. 2017. Integrating OpenMP into the Charm++ Programming Model. In *ESPM2'17: ESPM2'17: Third International Workshop on Extreme Scale Programming Models and Middleware, November 12–17, 2017, Denver, CO, USA.* ACM, New York, NY, USA, 7 pages.
https://doi.org/10.1145/3152041.3152085

Publication rights licensed to ACM. ACM acknowledges that this contribution was authored or co-authored by an employee, contractor or affiliate of the United States government. As such, the Government retains a nonexclusive, royalty-free right to publish or reproduce this article, or to allow others to do so, for Government purposes only.
ESPM2'17, November 12–17, 2017, Denver, CO, USA
© 2017 Copyright held by the owner/author(s). Publication rights licensed to Association for Computing Machinery.
ACM ISBN 978-1-4503-5133-1/17/11…$15.00
https://doi.org/10.1145/3152041.3152085

1 INTRODUCTION

Several trends in high-performance computing are converging to drive applications and systems software to rely on multi-threading in each node's shared memory, rather than running an independent process on each CPU core. Increasing per-chip concurrency creates pressure on system memory, system software, and application design. The number of cores and threads in each chip is increasing rapidly. Within each node, increased hardware parallelism entails reduced per-core/thread memory capacity and bandwidth. Therefore, many applications have been refactored to use multithreading to share the common resources within a process.

In addition to increased hardware variability, many parallel applications no longer operate in a regime where work and data can be neatly divided into uniform chunks distributed to each processor. This trend encompasses unstructured computations, data-dependent iterative methods, variable resolution, multi-physics simulations, and multi-phase execution. Load balancing in various forms can be applied to aid these applications, which often coarsens the problem to the node level to avoid considering an excessive number of cores. Within-node balancing can smooth out imbalances with lower overhead than is possible with global load balancing. Additionally, light weight balancing strategies within a process can supplement global load balancing across nodes, so long as the within-node load balancing does not compromise data locality or introduce large new bottlenecks or overheads. The MPI+X model has been used to improve load imbalance within a node but both programming models run in separate runtimes so there's no combined and adaptive scheduling of the fine-grained tasks considering locality and low overhead.

In this paper, we present a combination of an asynchronous many task distributed programming model with OpenMP that addresses many of these challenging trends with a low-overhead and locality-conscious design. We use Charm++ [1] as our distributed tasking model for its built-in support for across-node load balancing.

Charm++ periodically performs coarse load balancing in terms of objects that encapsulate associated work and data together, and assigns them to cores with good balance among nodes. These objects then adaptively share work with other cores in the same process, exposing fine-grained OpenMP tasks only to the extent that otherwise idle cores are available to help execute them. Thus, our design ensures locality as well as a low and proportionate scheduling overhead.

The contributions of this paper are:

- Integration of OpenMP with Charm++'s runtime system to enable fine-grained parallelism.
- Efficient implementation of dynamic scheduling of fine-grained tasks which uses an adaptive schedule based on the state of the system.
- An approach that combines infrequent distributed load balancing with shared-memory task parallelism to handle coarse-grained and fine-grained load balancing together.
- Performance improvements by using the integrated runtime system on the Lassen proxy application. We show a speedup of 50% over runs without load balancing, and a 10% improvement over those with existing global balancing strategies on Lassen.

2 CHARM++ PROGRAMMING MODEL FOR SHARED MEMORY

Charm++ is a parallel programming system which is based on an asynchronous message driven execution model. Each application's data and computations are encapsulated in entities called *chares*, which are C++ objects. The encapsulation of data and its computation into a chare, each of which is mapped to a specific core, inherently promotes data locality. An application written in Charm++ is over-decomposed into these objects. Chares interact via asynchronous method invocations and a method on a chare is executed when a message is received for it. Chare objects are assigned to a core by the runtime system.

In the message driven execution model of Charm++, the runtime system actively probes for incoming messages. On receiving a message, it identifies the corresponding *chare* which is targeted by the incoming message and schedules it. In Charm++, a *PE* refers to a processing element such as a *core* or a *hardware thread*, and we use these terms interchangeably.

The SMP mode of Charm++ takes advantage of multi-core shared memory processors. In this mode, a Charm++ OS process is launched with multiple threads and each thread is called a PE. In a typical configuration the number of threads launched by the Charm++ process is equal to the number of cores or hardware threads on a node. A PE is mapped to a separate core or a hardware thread, and PEs have CPU affinity. Each PE has a separate message queue and the scheduler on the PE picks up messages from the queue and handles it. Chares are mapped to PEs, and the PEs in a node can have multiple chares that they schedule in a message-driven manner. Running multiple threads in a single process enables work sharing without explicit inter-process data transfer.

3 OVERVIEW OF OUR PROPOSAL

The challenge, as outlined in Section 1, is to balance load across PEs while managing locality. A pure task model with randomized work stealing, or a pure dynamic schedule in OpenMP, sacrifices locality significantly to an extent that often nullifies the benefits of dynamic load balancing [9, 12]. Dynamic load balancing strategies are used to balance the load and redistribute the work at runtime. These load balancing strategies can incur a significant overhead due to the cost of computing a new assignment and the consequent data movement. If done less frequently, the overhead is reduced and locality is maintained, but dynamically emerging load imbalance may last longer before being corrected. With increasing number of cores within a node, intra-node load balancing becomes an effective way to reduce load imbalance.

The approach we propose is to utilize a relatively infrequent periodic assignment of work to cores based on load measurement, combined with user assisted creation of potential tasks from the work assigned to each core that the runtime can choose to make available to other cores. The periodic load balancing is based on *the principle of persistence*. Many HPC applications runs in a series of time steps and iterations, and have a repeated pattern of communication and computation which can be a good indicator to predict the future. Charm++ supports various load balancing strategies based on this principle. These strategies migrate chares across nodes based on load measurements of each PE.

Since inter-node load balancing is too costly to be called frequently, we utilize the idle cycles on other cores on a node with fine-grained tasks, which can redistribute loads within the node. We also need to make sure to not incur task creation overhead when tasks are not needed.

We support this approach with a method for users to create potential tasks. We use the term potential tasks to distinguish the application's OpenMP directives, which specify what can be parallelized, from our runtime implementation, which dynamically decides what is parallelized. This method builds on top of a task abstraction in Charm++ that integrates OpenMP with Charm++, such that each object can create potential tasks via OpenMP parallel loop constructs. Using this method, application developers can create potential tasks that can dynamically utilize all cores to restore load balance. We also develop multiple runtime scheduling strategies for managing these potential tasks.

4 OPENMP INTEROPERATION WITH CHARM++

In this section, we discuss the OpenMP thread model and our integration and optimization of its runtime features in Charm++.

4.1 Integration of OpenMP in Charm++

Common OpenMP runtime systems spawn their own threads independent of Charm++ worker threads. Without proper coordination between the two runtime systems the OpenMP and Charm++ threads will contend for hardware resources and lead to oversubscription of cores. To enable OpenMP to efficiently work with Charm++, we modified an OpenMP library to use Charm++ worker threads, so that the two runtimes can share resources.

We used GNU OpenMP 4.0, which is forked from GCC 4.9.3 and also implemented the same work using the LLVM OpenMP runtime library, which can work with GCC, ICC, and Clang on various environments. First, we modified the OpenMP runtime to use Charm++ threads to execute its tasks. Instead of spawning new threads for the execution of OpenMP tasks, our OpenMP runtime puts task descriptors into Charm++ messages. These messages are pushed onto a thread-local task queue that can be accessed by other threads on the same node. Idle threads steal tasks from this task queue. Because OpenMP is predominantly a synchronous programming model, all OpenMP programs have an implicit synchronization

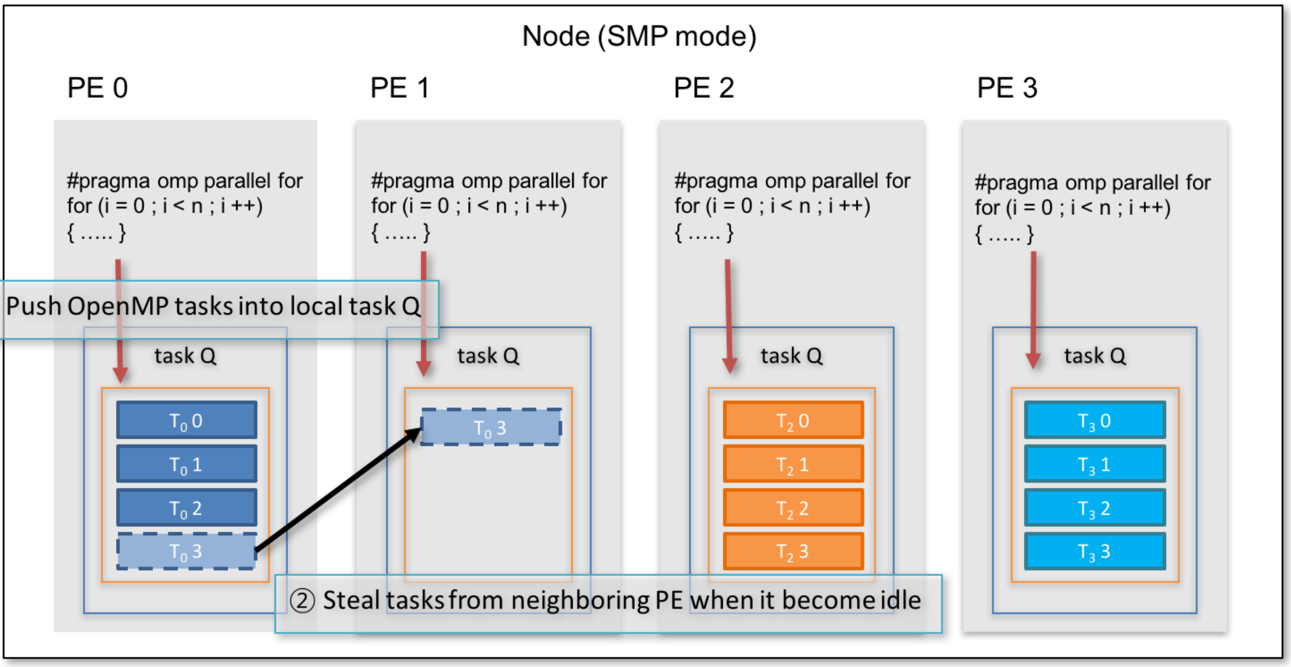

Figure 1: Implementation of OpenMP for Charm++ using the task API.

point in termination. Without removing these implicit synchronization points, the OpenMP tasks would make all Charm++ threads wait at a number of barriers.

As all threads in Charm++ are both worker as well as master threads, removing these barriers is necessary because otherwise this can lead to a hang. To solve this issue, we eliminate all barriers in OpenMP and replace them with atomic counters for each OpenMP task collection. When a chare generates OpenMP tasks, it records the number of tasks in its own team structure. Then, when other chares attempt to steal tasks from a busy thread, they decrement the appropriate counter to notify the master thread that its task is going to be executed. All OpenMP tasks pushed into the task queue can now be considered normal Charm++ messages, which can be executed and/or migrated within a node.

Figure 1 shows how OpenMP interoperates with Charm++ when Charm++ runs on a node with 4 PEs and use *static* scheduling to split each chare's task into OpenMP tasks. For the purpose of simplicity, we show how the static schedule of OpenMP works in this integrated runtime system. First, each chare splits its task into as many OpenMP tasks as there are PEs on a node. The OpenMP runtime puts each OpenMP task in a Charm++ message and pushes all of the messages into the thread local task queue. An idle thread can potentially steal a task from one of the busy threads on the same node, thereby distributing the work.

4.2 Task Queue

To support tasks, we created a task queue [13] on each PE, which is distinct from the normal message queue. The messages in the message queue are meant for that specific PE, whereas the tasks in the task queue can be stolen by different cores on a node. The scheduler on the PE polls the local task queue and the message queue for messages. We chose not to have a centralized task queue at the node level because then we lose locality information and there could be potential contention for the centralized queue. We have a separate task queue on each PE, which is a single producer multiple consumer queue for the fine-grained tasks. Whenever a PE becomes idle, it randomly chooses a PE and steals tasks from that PE's task queue. This is similar to Cilk's workstealing [4], except that our scheduler also polls other queues, including a PE-specific message queue for messages to chares assigned to that PE by the periodic load balancer.

The task queue is implemented using the Chase-Lev [7] non-blocking algorithm. The task queue is a double-ended queue. A push(t) call enqueues a task at the tail of the queue. A pop() call dequeues a task from the tail of the queue. A steal() call dequeues from the head of the queue. The queue is a cyclic array of task pointers with non-wrapping head and tail indices. A worker does a push(t) by adding the task at the tail of the queue and increments T, the tail pointer. A worker does a pop() by decrementing T. If it detects that there could be a conflict, then it uses compare and swap (CAS) to handle the conflict. A thief reads H and T and uses CAS to atomically increment H and obtains the task.

The task descriptor contains details about the task such as the object pointer, function pointer, parameters and an atomic variable. The message enqueued into the task queue contains range parameters and a pointer to the common task descriptor. To minimize the overheads of creating messages and task descriptors, we keep a pool of task messages and descriptors which are reused.

4.3 Scheduling schemes of OpenMP for Charm++

4.3.1 Basic scheduling schemes for OpenMP. The number of messages created for OpenMP tasks resulted in overheads in message creation and queue contention. We identified various opportunities for performance improvement and implemented them as different scheduling schemes. In the OpenMP standard, there are four kinds of scheduling schemes for OpenMP tasks. The first and default scheduling policy in many implementations is *static* scheduling. *static* scheduling assigns the iterations of a for-loop to cores in blocks of size number of iterations divided by the number of physical threads in a node. This incurs no overhead due to task creation and contention because it is done by the compiler. In the *dynamic* schedule, threads in a team pick and execute next available iterations. Dynamic scheduling incurs some overhead due to task creation, contention of shared resources as well loss of locality. In the *guided* policy, each thread in the team is assigned a chunk of iterations proportional to the number of unassigned iterations divided by the number of threads in a team. Whenever each thread in a team finishes its assigned task, the next assigned chunk is determined in this way. User can specify the minimum size of chunk in the *guided* policy. The *auto* policy is specific to each implementation.

4.3.2 Changing the portion of stealable OpenMP tasks. We first consider static scheduling and show how we minimize the overheads of our task scheduler. Although static scheduling avoids the runtime overhead of dynamic and guided policies, static scheduling can still cause significant overhead by the creation of excessive numbers of messages. To minimize overheads of accessing the local task queue, we make all threads keep a history vector to record the ratio of stolen tasks to locally executed tasks. Using the moving average of the previous ratios in the history vector helps each thread decide how many of the generated tasks it should push into its local task queue to expose for work stealing. This reduces the overhead for each thread to push and pop its own OpenMP messages into its local task queue.

4.3.3 Changing the number of OpenMP messages created. We use an atomic counter for the number of idle threads in the Charm++ runtime to prevent each thread from creating more messages than the number of idle threads. This can reduce overheads in creating messages significantly and efficiently. When the OpenMP runtime splits each thread task into OpenMP tasks, it first inspects the idle counter maintained by the runtime system. In addition to this value, the OpenMP runtime also looks at the local history record of previous ratios of work stolen. These ratios represents how many of tasks have been stolen by other threads. Then, when each thread needs to split their task into at least the number of messages proportional to the average of these previous ratios. In our integration of OpenMP for Charm++, we use a bigger value of average ratio in the history vector and the number of idle threads in the atomic counter to decide how many messages to create. Using only the counter may restrict parallelism at times because each thread may lose the opportunity to receive help from other threads becoming idle while its tasks are being executed.

4.4 Overhead of the OpenMP integration

To measure the overhead of the fine-grained parallelism by the OpenMP integration, we run a simple stencil application with Charm++, Charm++/OpenMP on Intel Xeon E5-1620 v3. This machine has four physical cores and two threads per core. We applied best thread affinity configuration for each case. And the size of the matrix for stencil application is 4096 x 4096. This application decomposes the matrix into 16 submatrices, each of the matrices is 4096 x 256. Figure 2 shows the result of this experiment. There's no improvement of the Charm++/OpenMP over Charm++ because this stencil application has balanced loads across PEs. So, there's no room for performance enhancement by redistributing load imbalance through the fine-grained parallelism of the integrated OpenMP. However, this experiment shows that OpenMP integration doesn't incur a significant overhead over pure OpenMP and the techniques we mentioned above help minimize the overhead efficiently by creating OpenMP tasks on Charm++ only when they are considered to be needed.

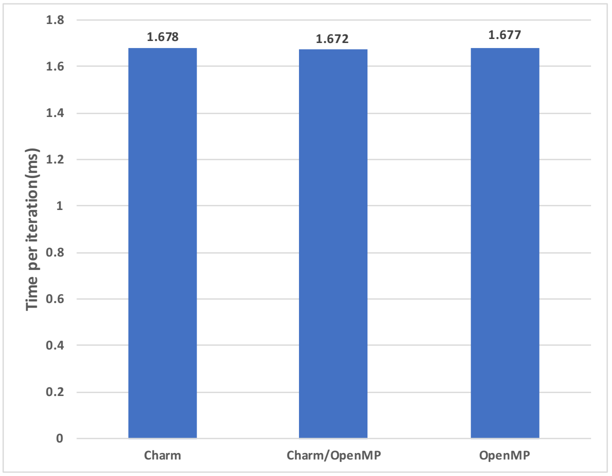

Figure 2: Performance of stencil application on Intel Xeon E5-1620 v3.

5 APPLICATION STUDY – LASSEN

We study the performance benefits of our new integrated runtime system that combines the Charm++ distributed memory model with the task model on a proxy application developed at Lawrence Livermore National Laboratory, called *Lassen*. We compare the performance of Lassen with and without the integrated OpenMP task model. We use the OpenMP integration implemented on GNU OpenMP library. And, the *history* scheme for OpenMP in conjunction with the *when idle* strategy, which resulted in the best performance. As explained in 4.3, the integrated OpenMP runtime uses a bigger value of the average number of OpenMP tasks stolen in the history vector and the number of idle threads through the idle threads counter as the number of OpenMP tasks to create. The use of these two values helps create OpenMP tasks only when needed without significant overhead.

Table 1: Values of the load imbalance metric λ of Lassen over the whole execution, at the cluster node and processing element (PE) levels. Lower values indicate better balance. For GreedyLB, a load balancing period of 50 iterations is used.

Level	No LB	GreedyLB	GreedyLB+OpenMP interop
Node	47.77 %	16.51 %	13.73 %
PE	227.60 %	110.43 %	42.59 %

Lassen is an LLNL ASC proxy application for simulation of detonation shock dynamics. It models wave propagation by tracking the wave front. This application has significant load imbalance where the load is concentrated just before and after the wave front. As the wave front moves, computation load also shifts. We used the Charm++ version of Lassen as our baseline for the experiments. The input to the application is a Cartesian mesh subdivided into domains and assigned to PEs. The number of domains used is 16 times the number of PEs.

We modified the baseline version of Lassen by adding two lines of `#pragma omp parallel for` to two 'for' loops. The reason why we chose only two loops is that these are compute intensive, while others are relatively more memory intensive. The parallelized loops calculate the distance between points or doing computations. More specifically, the loops not parallelized are moving data from one vector to another or just searching data over the vectors. For those memory intensive operations, more fine-grained parallelism can make them slower because of increased memory traffic and working set.

5.1 Load Imbalance

To formalize the load balance improvements of our proposal, we measure the load imbalance of Lassen with different load balancers and calculate the *percent imbalance* λ [15] with the following equation:

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

In the equation, L represents the load vector of nodes or PEs. This equation indicates the amount of imbalance, with higher values of λ indicating a higher imbalance, while a value of 0 indicates perfect balance.

The values of this metric are shown in Table 1 over the complete execution of Lassen, at the cluster node and processing element (PE) levels.

Without a load balancer (*No LB*), there is significant load imbalance across nodes and the load shifts between iterations. This inter-node imbalance can be handled by running a coarse-grained load balancer, such as the *GreedyLB* load balancer that is part of Charm++.

However, while the load across nodes is relatively balanced when running with GreedyLB, the load of PEs within each node is still imbalanced as indicated by the values in the table. One possible way to handle this would be to call the load balancer more frequently. Table 2 shows the timing for Lassen with different load balancing periods of GreedyLB. We break down the total execution time into time per step and load balancing time. When the load balancer is called frequently, it improves the time per step but results in a significant overhead.

Our approach is very well suited to handle this load imbalance problem. Instead of performing load balancing more frequently, we use tasks generated via our OpenMP integration. Table 1 shows that the load distribution across nodes and PEs using our integrated run-time system is much better. We can see that excess load is spread to other PEs within the node, resulting in a much higher overall balance compared to running with No LB and GreedyLB only.

5.2 Performance Improvements

Performance experiments were performed on Blue Waters, which is a Cray XE machine located at the National Center for Supercomputing Applications (NCSA). Each node contains two AMD Interlagos 6276 processors with 8 Bulldozer cores. Each Bulldozer core compute unit has 16 integer cores and 8 floating point cores. We show results from our modified version of the GNU OpenMP runtime integration.

We run Lassen to show the benefit of our work on Blue Waters with and without GreedyLB. Figure 3 shows how much Lassen performance is improved. GreedyLB can distribute load imbalance across nodes quite well, as discussed in the previous subsection. However, as we noted before, coarse-grained load balancing alone cannot redistribute all of the existing load imbalance because of its more significant overhead.

Figure 3 shows how the OpenMP integration can help distribute this load imbalance within each node. Even only with the OpenMP

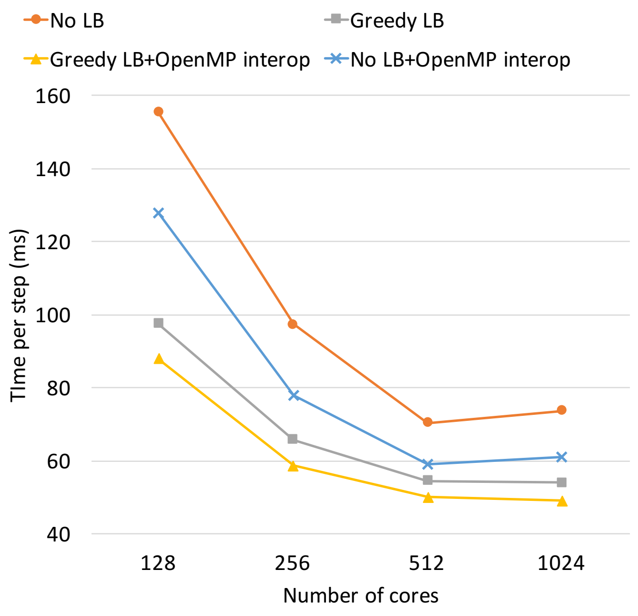

Figure 3: Strong scaling performance results of Lassen on Blue Waters, comparing the original Charm++ and OpenMP integration with GreedyLB and without LB.

Table 2: Timing results for Lassen, without load balancer (No LB), and with the coarse-grained GreedyLB balancer that is run with different periodicities (measured in time steps).

Measurement	No LB	GreedyLB, period 20	GreedyLB, period 50	GreedyLB, period 100
All application time steps	100 s	70 s	69 s	72 s
Load balancing time	0 s	19 s	8 s	5 s
Total execution time	100 s	89 s	77 s	77 s

integration, without any inter-node load balancing, the load imbalance in Lassen is quite well redistributed and the performance is improved by about 21% for 1024 cores. When combining the GreedyLB balancer with the OpenMP integration, performance is improved by 50% on 1024 cores. Compared to using GreedyLB only, performance gains are 10%. These results show that it is important to consider intra-node imbalance when running on large distributed systems. With our version of Charm++ integrated with OpenMP, users can easily resolve load imbalance in their application by adding simple flags of OpenMP, while they can redistribute load imbalance across nodes by using coarse-grained load balancing manually.

6 RELATED WORK

There has been extensive work studying the interoperation of MPI and OpenMP (e.g. [18, 19]). The MPI+X model on its own has been shown to improve load balance within each node [8]. We combine a periodic measurement-based inter-node load balancing scheme to attain approximate uniformity, with dynamic shared-memory execution to smooth out residual imbalances. Recent work has explored the hybrid model in more detail, mixing static and dynamic scheduling of work among cores on a node to improve the tradeoffs among overhead, locality, and load imbalance [11, 12]. Our work extends these ideas by adaptively tuning the level of dynamic scheduling to match its potential utility, thus reducing overhead further.

More recently, the hybrid model has been increasingly used with other shared memory programming models to handle within node parallelism. OmpSs [6] introduced concurrent tasks on top of OpenMP, with data dependences satisfied by MPI communication operations and coordinated by its runtime system. Recent versions of MPC bind an implementation of MPI that supports multiple ranks in each OS process [17] to multi-threading via POSIX threads, OpenMP, and Intel TBB. This paper moves in a similar direction, by directly scheduling execution of various shared-memory tasks to run on normal Charm++ worker threads, overlaid on the work/data mappings generated by Charm++'s distributed memory load balancing infrastructure.

The approach of work-stealing task scheduling has been used in Cilk [4], Intel TBB [16], OpenMP 3.0 [14] and Habanero [3]. The randomized work-stealing used in Cilk can result in loss of locality. TBB has a mechanism to bind each loop iteration to the same worker thread that previously executed that iteration, thereby favoring temporal cache-reuse. The Habanero runtime system has an adaptive locality-aware work-stealing scheduler [10] to increase temporal data reuse.

7 CONCLUSIONS AND FUTURE WORK

In this paper, we proposed a new integrated runtime system that combines the OpenMP runtime with the Charm++ distributed programming model, with a focus on handling load imbalance. Our proposal utilizes a relatively infrequent periodic assignment of work to cores based on load measurement, in combination with user created tasks to handle coarse-grained and fine-grained load balancing together.

We integrate OpenMP with Charm++ so as to enable objects to create potential tasks via OpenMP's parallel loop construct. We have shown that OpenMP can be embedded successfully to handle fine-grained load balancing. Our experiments show that a combination of within-node and across-node load balancing can improve the performance of the Lassen proxy application by 10% to 20% at 1,024 cores compared to when one of them is used alone. Overall, when using our integrated OpenMP runtime with Charm++'s existing global load balancing support, Lassen performs 50% better than without any load balancing at all.

This work has many opportunities for further improvement. First, the task generation scheme we used currently supports relatively flat set of tasks generated by parallel loops. A possible future extension is to add support for tasks with dependencies, similar to OmpSs [6], StarPU [2], and PaRSEC [5]. In addition, this work uses randomized work-stealing algorithm that Cilk [4] proposed. This algorithm is good for multicore processors having a few number of cores but not optimal for manycore processors consisting of tens of cores in hierarchical topology due to increasing migration and interconnection cost across cores. The randomized work-stealing can be replaced by topology aware hierarchical stealing.

ACKNOWLEDGMENT

This work was performed under the auspices of the U.S. Dept. of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344 (LLNL-CONF-728277).

REFERENCES

[1] Bilge Acun, Abhishek Gupta, Nikhil Jain, Akhil Langer, Harshitha Menon, Eric Mikida, Xiang Ni, Michael Robson, Yanhua Sun, Ehsan Totoni, Lukasz Wesolowski, and Laxmikant Kale. 2014. Parallel Programming with Migratable Objects: Charm++ in Practice. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC '14)*. 647–658.

[2] Cédric Augonnet, Samuel Thibault, Raymond Namyst, and Pierre-André Wacrenier. 2010. StarPU: A Unified Platform for Task Scheduling on Heterogeneous Multicore Architectures. *Concurrency and Computation: Practice and Experience, Euro-Par 2009 best papers issue* (2010). Accepted for publication, to appear.

[3] Rajkishore Barik, Zoran Budimlic, Vincent Cave, Sanjay Chatterjee, Yi Guo, David Peixotto, Raghavan Raman, Jun Shirako, Sağnak Taşırlar, Yonghong Yan, et al. 2009. The Habanero multicore software research project. In *Proceedings of the 24th ACM SIGPLAN conference companion on Object oriented programming systems languages and applications*. ACM, 735–736.

[4] Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. 1995. Cilk: An Efficient Multithreaded Runtime System. In *Proc. 5th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP'95*. Santa Barbara, California, 207–216. MIT.

[5] George Bosilca, Aurelien Bouteiller, Anthony Danalis, Mathieu Faverge, Thomas Hérault, and Jack J Dongarra. 2013. PaRSEC: Exploiting heterogeneity to enhance scalability. *Computing in Science & Engineering* 15, 6 (2013), 36–45.

[6] Javier Bueno, Luis Martinell, Alejandro Duran, Montse Farreras, Xavier Martorell, Rosa M Badia, Eduard Ayguade, and Jesús Labarta. 2011. Productive cluster programming with ompss. In *Euro-Par 2011 Parallel Processing*. Springer, 555–566.

[7] David Chase and Yossi Lev. 2005. Dynamic circular work-stealing deque. In *Proceedings of the seventeenth annual ACM symposium on Parallelism in algorithms and architectures*. ACM, 21–28.

[8] Julita Corbalan, Alejandro Duran, and Jesus Labarta. 2004. Dynamic load balancing of MPI+OpenMP applications. In *Parallel Processing, 2004. ICPP 2004. International Conference on*. IEEE, 195–202.

[9] Simplice Donfack, Laura Grigori, William D Gropp, and Vivek Kale. 2012. Hybrid static/dynamic scheduling for already optimized dense matrix factorization. In *Parallel & Distributed Processing Symposium (IPDPS), 2012 IEEE 26th International*. IEEE, 496–507.

[10] Yi Guo, Jisheng Zhao, Vincent Cave, and Vivek Sarkar. 2010. SLAW: a scalable locality-aware adaptive work-stealing scheduler for multi-core systems. In *ACM Sigplan Notices*, Vol. 45. ACM, 341–342.

[11] Vivek Kale, Simplice Donfack, Laura Grigori, and William D Gropp. 2014. Lightweight Scheduling for Balancing the Tradeoff Between Load Balance and Locality. (2014). Poster presented at SC'14.

[12] Vivek Kale, Amanda Randles, and William D Gropp. 2014. Locality-Optimized Mixed Static/Dynamic Scheduling for Improving Load Balancing on SMPs. In *Proceedings of the 21st European MPI Users' Group Meeting*. ACM, 115.

[13] Harshitha Menon. 2016. *Adaptive Load Balancing for HPC Applications*. Ph.D. Dissertation. Dept. of Computer Science, University of Illinois.

[14] OpenMP ARB. 2008. OpenMP application program interface version 3.0. In *The OpenMP Forum, Tech. Rep.*

[15] Olga Pearce, Todd Gamblin, Bronis R. de Supinski, Martin Schulz, and Nancy M. Amato. 2012. Quantifying the effectiveness of load balance algorithms. In *26th ACM international conference on Supercomputing (ICS '12)*. 185–194.

[16] Chuck Pheatt. 2008. Intel® threading building blocks. *Journal of Computing Sciences in Colleges* 23, 4 (2008), 298–298.

[17] Marc Pérache, Patrick Carribault, and Hervé Jourdren. 2009. MPC-MPI: An MPI Implementation Reducing the Overall Memory Consumption. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface, Proceedings of the 16th European PVM/MPI Users' Group Meeting (EuroPVM/MPI 2009)*, Matti Ropo, Jan Westerholm, and Jack Dongarra (Eds.). Lecture Notes in Computer Science, Vol. 5759. Springer Berlin Heidelberg, 94–103.

[18] Rolf Rabenseifner, Georg Hager, and Gabriele Jost. 2009. Hybrid MPI/OpenMP Parallel Programming on Clusters of Multi-Core SMP Nodes. In *Proceedings of the 2009 17th Euromicro International Conference on Parallel, Distributed and Network-based Processing (PDP '09)*. IEEE Computer Society, Washington, DC, USA, 427–436.

[19] Lorna Smith and Mark Bull. 2001. Development of Mixed Mode MPI / OpenMP Applications. *Scientific Programming* 9, 2,3 (Aug. 2001), 83–98.