Characterizing MPI and Hybrid MPI+Threads Applications at Scale: Case Study with BFS

Abdelhalim Amer,* Huiwei Lu,† Pavan Balaji,† Satoshi Matsuoka*

*Tokyo Institute of Technology, {amer@matsulab, matsu@}is.titech.ac.jp
†Argonne National Laboratory, {huiweilu@mcs., balaji@}anl.gov

***Abstract*—With the increasing prominence of many-core architectures and decreasing per-core resources on large supercomputers, a number of applications developers are investigating the use of hybrid MPI+threads programming to utilize computational units while sharing memory. An MPI-only model that uses one MPI process per system core is capable of effectively utilizing the processing units, but it fails to fully utilize the memory hierarchy and relies on fine-grained internode communication. Hybrid MPI+threads models, on the other hand, can handle intranode parallelism more effectively and alleviate some of the overheads associated with internode communication by allowing more coarse-grained data movement between address spaces. The hybrid model, however, can suffer from locking and memory consistency overheads associated with data sharing.**

In this paper, we use a distributed implementation of the breadth-first search algorithm in order to understand the performance characteristics of MPI-only and MPI+threads models at scale. We start with a baseline MPI-only implementation and propose MPI+threads extensions where threads independently communicate with remote processes while cooperating for local computation. We demonstrate how the coarse-grained communication of MPI+threads considerably reduces time and space overheads that grow with the number of processes. At large scale, however, these overheads constitute performance barriers for both models and require fixing the root causes, such as the excessive polling for communication progress and inefficient global synchronizations. To this end, we demonstrate various techniques to reduce such overheads and show performance improvements on up to 512K cores of a Blue Gene/Q system.

I. Introduction

While parallel computers with millions of cores are already in production, the trend is toward higher core densities with deeper memory hierarchies, although other node resources are not scaling proportionally (e.g., memory capacity per core). Consequently, parallel programming models need to offer means to express communication, synchronization, and memory-consumption-reducing algorithms in order to efficiently run on these systems. The Message Passing Interface (MPI) [1] is the predominant programming system on these platforms, but MPI alone is often insufficient to take full advantage of the memory hierarchy in the system. Consequently, application developers are increasingly looking at using hybrid programming models to utilize the computational units while sharing memory.

Hybrid MPI+X programming, where "X" denotes a shared-memory programming model, has recently emerged as a viable model for many-core architectures. A common variant of this model is to use OpenMP [2] or Pthreads for "X," where multiple threads are used for intranode parallelism while internode communication is done with one or more communication threads. Such a model, however, has its own pros and cons compared with those of an MPI-only model. An MPI-only model that uses one MPI process per system core can effectively utilize the available processing units, but it fails to fully utilize the memory hierarchy. Specifically, each process memory is private, thus requiring message passing to move data between cores, and often involving duplication of data between processes to improve local access. The hybrid MPI+threads model, on the other hand, can handle intranode parallelism more effectively and can alleviate some of the interprocess data movement constraints associated with the MPI-only model by allowing a more coarse-grained "node-to-node" data movement (e.g., if threads share all memory on a node), rather than having to send a separate message to every core. The hybrid model, however, can suffer from locking and memory consistency overheads associated with data sharing. Moreover, the drawbacks of these models that may cause performance issues at small scales may differ from the ones that become prominent at large scales. For example, our experiments indicated that communication progress on requests whose number scales with the number of processes, which does not cause much overhead at small scales, can become a bottleneck at large scales.

In this paper, we use a distributed implementation of the breadth-first search (BFS) algorithm in order to understand the performance characteristics of MPI-only and MPI+threads communication models at scale. We start with the MPI-only implementation of BFS and then extend it to an MPI+threads implementation where threads independently communicate with remote processes while cooperating for local computation. Our extensions take advantage of both driving communication by multiple cores, offered by an MPI-only approach, and efficient shared-memory parallel computation, offered by threading models. At large scale, despite the fact that our hybrid method outperforms the MPI-only method, both implementations hit scalability barriers. Our analysis suggests that the inefficiencies stem from a number of aspects including eager polling for communication progress, inefficient global synchronization, and thread contention in MPI. We demonstrate that our hybrid design reduces some of these negative effects, because polling for communication and performing global synchronization scale with the number of processes P, where P is often larger for an MPI-only model than for a hybrid model. However, a hybrid approach only delays hitting the scalability wall and does not fix the root causes. We therefore propose various enhancements including a lazy polling approach and a nonblocking process synchronization (using the MPI-3 nonblocking barrier) to minimize the impact of such communication management, thereby improving the performance of both the MPI-only and the hybrid methods. Through detailed experimentation and analysis, we demonstrate that our techniques can reduce the cited overheads at a very large scale and improve the performance of BFS by *35-fold* on 16,384 cores while scaling to 524,288 cores of a Blue Gene/Q system.

II. Background

In the following we briefly introduce the MPI-only and hybrid MPI+threads models and the baseline BFS algorithm.

A. MPI-Only and MPI+Threads Paradigms

In Fig.1 we present a conceptual comparison between the MPI-only and the hybrid model. The MPI-only model has a finer-grained internode communication model and enforces explicit interprocess communication even within the same physical node. In addition, data sharing is not possible between address spaces, a feature that reduces further the ability of the model to exploit the memory hierarchy. Moreover, the available memory for the application is reduced compared with that of a hybrid model because of often-duplicated boundary data and the memory requirements of the MPI runtime. In contrast, the hybrid model moves data between address spaces in a coarser-grained manner while threads share the same address space. Arguably, however, the threads' shared-memory view may incur overheads in order to maintain consistent execution through locking, synchronization, and memory barriers.

The superiority of either model highly depends on the target machine, such as the performance and density of the cores, the memory and network performance, and the topology of the components, as well as the characteristics of the application, such as its computation and communication requirements, the amount of parallelism, and the application's ability to map to the machine topology. However, the technology trend suggests that hybrid models are likely to be better at handling large-scale machines. The reason is primarily the increasing core density in cluster nodes that requires efficient shared-memory parallel execution and, combined with the growth in the number of nodes, makes fine-grained core-to-core MPI operations difficult to scale in terms of performance and memory consumption. The implications of using either model are still not well understood, however, and require further investigation with various architectures, algorithms, and applications. Here we study these two models using BFS. In addition, we execute large-scale runs (up to 512K cores) and allow multithreaded concurrent access to the MPI runtime in order to expose thread-safety overheads of the hybrid model.

B. Interoperation between MPI and Threads

The MPI standard defines four levels of threading support—`MPI_THREAD_SINGLE`, `MPI_THREAD_FUNNELED`, `MPI_THREAD_SERIALIZED`, and `MPI_THREAD_MULTIPLE`—to be specified when initializing MPI [1]. These levels are listed in increasing order of threading support, and the user can choose the desirable level depending on the needs of

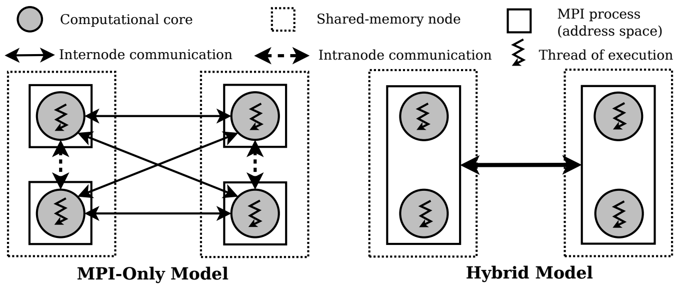


Fig. 1: Conceptual comparison between the MPI-only and the MPI+threads hybrid model.

the target application. Threads in our BFS implementation concurrently perform computation and communication in order to maximize throughput and minimize idleness. Thus, we require the `MPI_THREAD_MULTIPLE` threading support from the MPI library.

C. Breadth-First Search

Here we describe the breadth-first search algorithm and its baseline distributed implementation.

Given a graph $G(V, E)$, where V is a set of vertices and E a set of edges, and a root vertex $r \in V$, the BFS algorithm explores the edges of G, in order to traverse all the vertices reachable from r, and produces a breadth-first tree rooted at r. Most parallel graph traversal algorithms are performed level by level, where the vertices at the same level are equidistant from the root. Because synchronization is required between levels, this approach is referred to as *level-synchronized*. We choose the baseline BFS algorithm from the Graph500 benchmark [3] because it is widely used and studied [4], [5], [6]. Assuming P MPI processes, we describe the baseline MPI-only implementation in Fig.2 along with the details of the underlying routines provided in Fig.3 and Fig.4.[1]

We assume in Fig.2 that the graph is already partitioned between the processes. `CQ` and `NQ` are queues that store the vertices at the current level and the ones that will be visited at the next level, respectively. When a vertex is visited, it is marked in the `visited` vector, and its parent is recorded in the `pred` vector. Each process traverses the graph level by level starting from the root vertex at level 0. First, the root of the graph is enqueued in `CQ` (line 2). The graph traversal is done by the main loop on line 3 with each loop iteration corresponding to processing one level. Given a vertex in `CQ`, its neighbors are processed in the loop on line 7. The newly visited vertices will be enqueued in `NQ` (in `Update` on line 9). At the end of each level, `CQ` and `NQ` are swapped so that `CQ` contains the new vertices to be processed while `NQ` reuses `CQ`'s memory to contain the next-level vertices. The algorithm stops by breaking out of the main loop after `NQ`s of all processes are empty. Computation is carried out by the `Update` routine, and the bulk of interprocess communication is ensured by nonblocking point-to-point communication. The algorithm simulates an event-driven execution by polling for incoming communication (`CheckIncomMsgs`) and for pending send operations (`WaitPendSend`). After all the

[1]The communication routine names are similar to those in the MPI standard but are truncated in order to keep the pseudo-algorithm simple.

```
R ← RANK();                                          ▷ Get my rank
ENQUEUE(CQ, root);
while (True) do
    IRECV(AnySource);
    for (u ∈ CQ) do
        CHECKINCOMMSGS;
        for (v ∈ Neighbors(u)) do
            O ← Owner(v);                            ▷ Get owner of v
            if (R = O) then UPDATE(u,v) ;
            else
                if (Pending Send to O) then
                    WAITPENDSEND(O);
                Buffer(v, u, O);                     ▷ Put (v,u) in O's buffer
                if (O's buffer full or last message) then
                    Isend(O);                        ▷ Send buffer content to O
    SYNCHRONIZE;
    count ← |NQ|;
    ALLREDUCE(count);
    if (count = 0) then
        break;                                       ▷ NQ of all processes is empty
    SWAP(CQ, NQ);
```

Fig. 2: Pseudo-algorithm executed by each process for the Graph500 BFS reference simple implementation

```
UPDATE(u, v):
    if (visited[v] = 0) then
        visited[v] ← 1;
        pred[v] ← u;
        ENQUEUE(NQ, v);
CHECKINCOMMSGS:
    CHECKREQUESTS;
WAITPENDSEND(p):
    while (Pending Send to p) do
        CHECKREQUESTS;
SYNCHRONIZE:
    for (each remote process p) do
        ISEND(EmptyMessage, p)  ▷ Signal that I am done
    repeat CHECKREQUESTS until All processes done;
```

Fig. 3: Details of the routines from the algorithm in Fig.2

vertices of CQ are processed, the processes synchronize globally by exchanging empty messages (Synchronize). Progress on communication is ensured by polling for incoming and outgoing requests using the CheckRequests routine.

III. Design and Implementation

Our hybrid implementation builds on top of the baseline algorithm described in Section II. The main difference is that within the same node both computation and communication are driven by OpenMP threads instead of separate MPI processes.

```
CHECKREQUESTS:
    if (TESTRECV()) then
        for ((v, u) ∈ RecvBuff) do UPDATE(u,v) ;
        IRECV(AnySource);
    for (each remote process p) do
        if TESTSEND(p) then
            FREEBUFF(p)          ▷ Mark the buffer as free
```

Fig. 4: Communication progress routine

A thread is considered here as an independent entity with an execution flow similar to the process execution flow of the baseline method. We show in Fig.5 where the team of threads is spawned (line 3) and later joined. That is, threads do all the work in parallel except the Allreduce operation. We describe below how computation and communication are implemented. In the rest of the document, we refer to the problem size by the graph $scale$, where $scale = \log_2 |V|$. In addition, we use Kronecker graphs [7] as inputs for our experiments.

A. Computation

Various optimizations were explored during the past decade to improve the efficiency of BFS on shared-memory architectures, among which several are exploited in our hybrid implementation. First, we inherit the visited vector bitmap representation to reduce the memory footprint of the graph and reduce the amount of data movement [8]. Second, we update the shared visited and pred vectors and the CQ and NQ queues in a lock-free and atomic-free manner. More precisely, the CQ is read-only so it does not cause any issue. Writing to NQ, however, needs to be protected, so we use private queues per thread and then merge them at the end of each level. This strategy also improves data locality because vertices visited by the same thread will be gathered contiguously in NQ, and it increases the opportunities for data reuse when reading the same vertices from CQ.

Our method shares many similarities to the shared-memory BFS implementation of Chhugani et al. [9]. One major

```
ENQUEUE(CQ, root);
while (True) do
    foreach thread parallel do
        IRECV(AnySource);
        for (u ∈ CQ) do
            ▷ Do local computation, send operations, and
            process incoming messages
        SYNCHRONIZE;
    count ← |NQ|;
    ALLREDUCE(count);
    if (count = 0) then
        break;                    ▷ NQ of all processes is empty
    SWAP(CQ, NQ);
```

Fig. 5: Multithreading extension to the baseline BFS 2

difference is that we do not maintain an array for the depth of the vertices that was exploited by their algorithm to confirm whether a vertex is visited. Instead, we initialize the parent of each vertex with a negative value and test against it to achieve the same atomic-free algorithm as shown in Fig.6. Although the algorithm exhibits possible data races, it guarantees the generation of a correct BFS tree on architectures that ensure atomic loads/stores (see [9] for a detailed explanation). Fig. 7a shows a single-node performance comparison between the baseline MPI-only design and our multithreaded implementation on a Blue Gene/Q node. While both methods scale with the number of cores, the multithreaded method achieves close to 2x better performance. Although the performance of the multithreaded computation can be improved, we do not optimize the computation part any further since we ignore whether it will constitute a bottleneck at large scale. In fact, we show later that the primary shortcomings of both implementations are related to the communication rather than the computation performance.

B. Communication

To avoid thread contention at the application level, we manage the communication following the same methodology as with the computation part. Since accessing the communication buffers is critical, we use thread-private buffers mapped to remote processes. This approach ensures asynchronous progress and thread independence where the only synchronization point is the implicit barrier at the end of the parallel region. In particular, synchronization inside the parallel region is avoided through the use of OpenMP `nowait` clauses. We note that the `Synchronize` step in Fig.5 is also performed by all threads. Each thread sends to all processes a termination message and expects the same type of message from each remote process. In the following, we model the communication analytically in order to compare the communication costs of running with only processes or with one process and multiple threads per node.

1) Communication Characterization

Assuming P processes, V a set of vertices in the graph, and an edge factor of e, we estimate $\mathcal{C}$, the total number of vertices communicated, as

$$\mathcal{C}(P,V) = 4e|V|\frac{P-1}{P}. \tag{1}$$

The derivative $\frac{\partial \mathcal{C}}{\partial P}$ is always positive and proves that $\mathcal{C}$ grows proportionally with the number of processes. If we assume that P in an MPI-only model is larger than that of a hybrid

```
1 UPDATE($u, v$):
2     if ($visited[v] = 0$) then
3         $visited[v] \leftarrow 1$;
4         if ($pred[v] = -1$) then
5             $pred[v] \leftarrow u$;
6             ENQUEUE($NQ_i$, $v$); ▷ Add $v$ to thread $i$'s NQ
```

Fig. 6: Multithreaded `Update` routine

model by a factor α, then the growth in communication can be estimated by

$$\frac{\mathcal{C}(\alpha P, V)}{\mathcal{C}(P,V)} = \frac{\alpha P - 1}{\alpha(P-1)}. \tag{2}$$

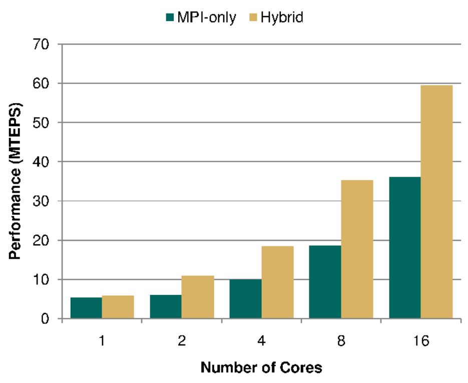
We note that for systems where the number of nodes is significantly larger than the core density ($P \gg \alpha$), the increase in communication of the MPI-only over a hybrid method is negligible. For a small-diameter cluster with high core-density nodes, however, the reduced internode communication of the hybrid model can be significant. For instance, with 128 processes and 16 threads per process, we estimate and confirm experimentally the total amount of communication and show the message count in Fig.7b and Fig.7c, respectively. Although the gain by the hybrid method in terms of reduced communication is not large in this case, the MPI-only method incurs a larger communication overhead by sending more messages. This message count increase stems from finer-grained graph partitioning between the processes and empty messages during the global synchronizations.

2) Scalability of the Global Synchronization

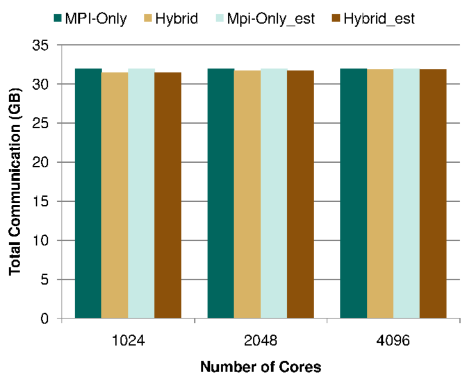
In our design, a thread sends an empty message to each remote process, to signal that it is done sending and expects to receive an empty message from each process. Although we could join the threads beforehand and perform the global synchronization by a single thread, we choose to involve all the threads in this step to process the incoming messages in parallel. Despite the fact that all threads participate in the synchronization, this method is more scalable than in the MPI-only case. Let us assume M cores per node, N nodes, one process per core for the MPI-only method, and one process per node and one thread per core for the hybrid method. Then the number of empty messages scales as $O(M^2N^2)$ for the MPI-only method and as $O(MN^2)$ for the hybrid method. As a result, the hybrid method reduces by M-fold the overhead of the global synchronization.

IV. Evaluation and Analysis

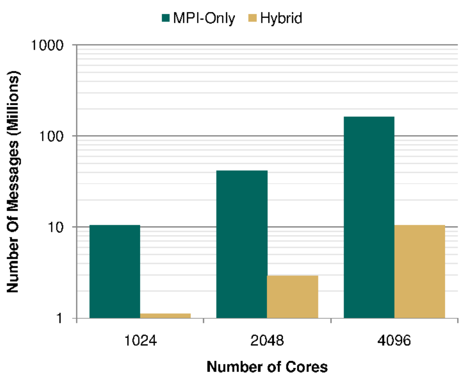
In this section we present our initial evaluation of both approaches, followed by a series of analyses of the bottlenecks and their corresponding optimizations. The evaluation was conducted on a Blue Gene/Q system (Table I) while interprocess communication was ensured by using MPICH 3.1.1. Because of the larger memory footprint and the suboptimal performance when using more that one process per core, we use only one hardware thread per core. For fairness reasons, the MPI-only method dictates the problem sizes and thread count per process for the hybrid method that uses one MPI process per node. The maximum achievable performance by the latter method is higher than what is shown in the present document because bigger problems can be run and better results were observed with 32 threads instead of 16. Thread safety in MPICH is guaranteed through a global critical section. Although MPICH supports fine-grained critical sections on Blue Gene systems, it has a higher overhead on the fast path, that is, the execution path free of lock contention. Hence, without proof of contention in the runtime, the application developer is advised to configure MPICH with the global critical section support.



(a) Single-node performance

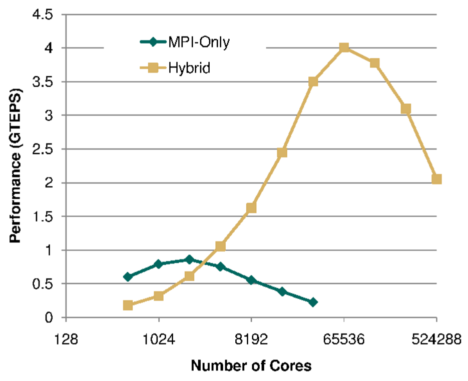


(b) Total communication

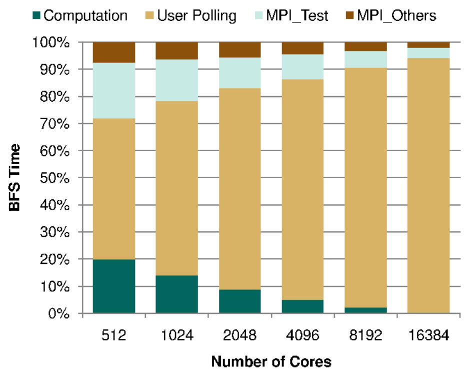


(c) Total message count

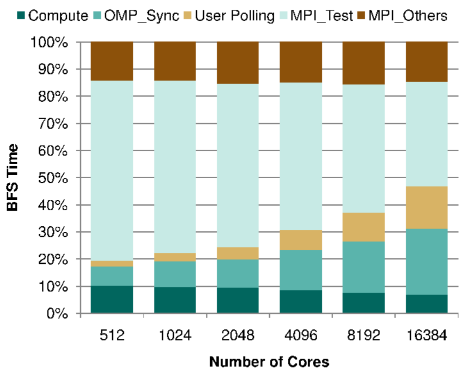
Fig. 7: (a) Performance comparison between the MPI-only and the hybrid implementations on a single BG/Q node with a problem size of 21. (b) Total amount of communication aggregated across all processes with a problem size 26, one process/thread per core for the MPI-only/hybrid method, respectively. (c) Corresponding total message count. `MPI-Only_est` and `Hybrid_est` are the estimated communication for the MPI-only and the hybrid methods, respectively.



(a) Performance comparison



(b) Profile for MPI-only



(c) Profile for Hybrid

Fig. 8: (a) Preliminary weak-scaling performance results with problem sizes from 25 to 35, with 16 processes per node (PPN) for MPI-only and 1 PPN and 16 TPN (threads per node) for the hybrid version. (b) Execution breakdown of the weak-scaling experiment for the MPI-only method. (c) Execution breakdown of the weak-scaling experiment for the hybrid method.

TABLE I: Target platform specification

Architecture	Blue Gene/Q
Processor	PowerPC A2
Clock frequency	1.6 GHz
Cores per node	16
HW threads/Core	4
Number of nodes	49152
Interconnect	Proprietary
Topology	5D Torus
Compiler	GCC 4.4.7
Network driver	BG/Q V1R2M1

A. *Preliminary Evaluation*

We performed a comparative analysis between the baseline process implementation and our hybrid method. We show in Fig.8a weak-scaling performance results. We observe that at small scales, the hybrid method performs worse than MPI-only. At larger scale ($\geq$4K cores), however, the MPI-only implementation stops scaling, whereas the hybrid method performs better and stops scaling only after $64K$ cores. We do not show data points after $32K$ cores with the reference implementation because it runs out of memory. This issue arises during the graph construction when using a flat-MPI model and has been reported by previous authors [6]. The trend is obviously toward worse performance. Most of the inefficiency of the hybrid method at small scale stems from runtime contention which is discussed in Section IV-D.

To understand the source of the performance breakdown, we profiled the the previous experiments and show the results in Fig.8b and Fig.8c. Thread load imbalance is estimated by the average time spent between the end of the global synchronization step and the end of the OpenMP parallel region and is shown as `OMP_Sync` in the hybrid model profiling figures. The non-overlapped communication cost is estimated by summing the time spent in the MPI runtime (`MPI_Test`, and `MPI_Others`[2]) and the time spent polling at the user level (outside the MPI runtime). Figure 8b shows that the main bottleneck in the reference implementation is polling for communication progress outside the MPI runtime. The analysis in Section III showed that the difference in communication

[2]`MPI_Isend`, `MPI_Irecv`, `MPI_Allreduce`, and other routines involved in the global synchronization implementation of Section IV-C

volume between the two methods is small and does not justify such a gap in the communication cost. We notice that the loop for checking outgoing requests in the communication progress routine (Fig.4) scales as $O(P)$. Although the same routine is used by the hybrid solution, it is more scalable because P is smaller by a factor equal to the number of cores per node. Nevertheless, since the hybrid method scales with the number of nodes, this issue is only delayed; and the hybrid approach breaks down at $64K$ cores. Hence, fixing the root issue is necessary.

B. Reducing Synchronization between Endpoints

Polling for outgoing requests completion with `CheckRequests` is performed at several execution points. In short, the algorithm eagerly checks the requests in order to mark buffers as *free* as early as possible. We point out that doing so is not necessary, however, because a buffer may effectively be needed only at a later time. Although regularly checking for incoming messages is essential, for outgoing messages we propose a heuristic that delays polling until the buffer is needed. That is, we avoid checking outgoing requests at each iteration of the main loop and at the global synchronization step. In addition, waiting for a buffer to be freed involves only the corresponding outgoing request and avoids polling for $O(P)$ requests. We refer to this method as *lazy polling (LP)* and illustrate the changes to the routines in Fig.9. The performance gain of this optimization is shown in Fig.10a. We notice that with LP, both the MPI-only and MPI+threads methods are more scalable. The profiling results in Fig.10b and Fig.10c confirm that the *lazy polling* policy substantially reduces the polling overhead. Although not experimented here, it may be possible to reduce further the overhead of maintaining a large number of requests by eliminating the dependency on the number of processes and using a pool of a fixed number of requests. Instead, we will investigate another source of communication overhead since significant time is spent in communication with both methods.

C. Efficient Global Synchronization

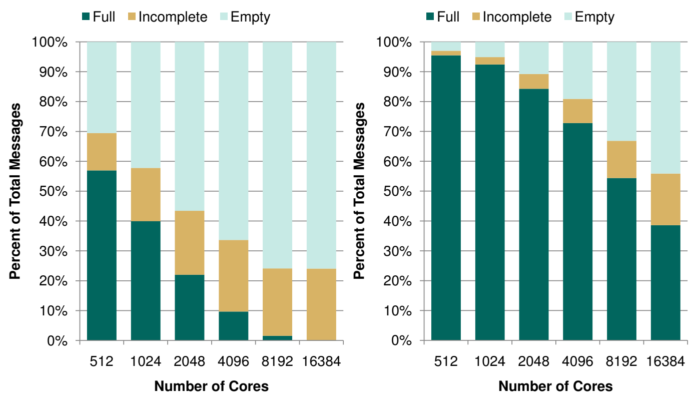
As mentioned in Section III, the original design incurs an overhead due to empty messages that scales as $O(M^2N^2)$. Although the hybrid method reduces this overhead by a factor M, it is still not scalable because the overhead grows quadratically with the number of nodes. In Fig.11, we depict the distribution of the messages in a BFS run according to their type—*full*, *incomplete*, and *empty*—in a weak-scaling experiment. Ideally only full messages would be exchanged. We observe, however, that the ratio of full messages decreases at scale. We also confirm experimentally that the multithreaded implementation inherits the same issue but incurs fewer empty messages than does the MPI-only method. These results encourage implementing a better global synchronization algorithm.

```
CHECKINCOMMSGS:
    if (TESTRECV()) then                         ▷ Test Recv requests
        for $((v,u) \in RecvBuff)$ do UPDATE(u,v); ;
        IRECV(AnySource) ;
WAITPENDSEND(p):
    while (Pending Send to p) do
        CHECKINCOMMSGS;              ▷ Make progress on Recv
        if (TESTSEND(p)) then                    ▷ Test Send request
            FREEBUFF(p);
```

Fig. 9: Lazy polling implementation

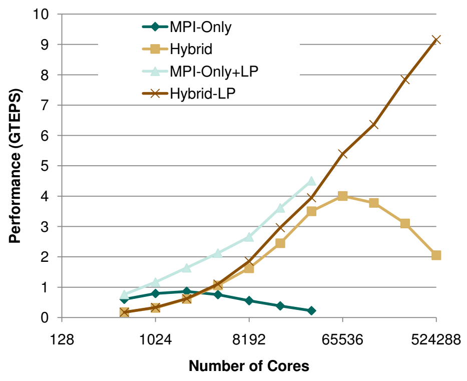
The difficulty here is to ensure a global barrier-like synchronization while processing incoming messages to avoid deadlocks. We propose the new `Synchronize` routine implementation in Fig.12. Our solution assumes the availability of an implementation of the recently released MPI-3 standard, which supports nonblocking barriers. Most supercomputers, including Blue Gene/Q, Cray, and InfiniBand platforms, support MPI-3 at this point. In addition, since handling incoming messages involves computation (`Update` operation) and internal MPI processing, the hybrid method uses multiple threads during this step. Here, a single thread is responsible for calling the nonblocking barrier. When one of the threads detects the barrier completion, it sets the `done` flag that signals the end of the synchronization step for the other threads. Assuming that a barrier implementation incurs $O(PLogP)$ message exchanges, where P equals the number of processes, we estimated the cost of the global synchronization as $NLogN$ instead of the original MN^2 for the hybrid implementation. We measured performance and profiling data after using the optimized global synchronization (with the suffix +IB short for `Ibarrier`) as shown in Fig.13. We observe in Fig.13a that the scalability of both methods has improved.

We note that the basic principle behind the LP and IB optimizations is to avoid algorithmic features that scale with the number of processes. This is an important rule that is applicable to BFS but also to other algorithms and applications.

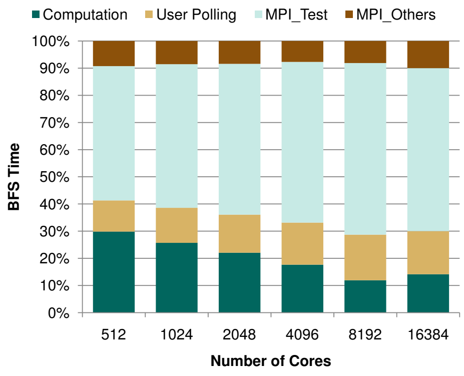


(a) MPI-only (b) Hybrid

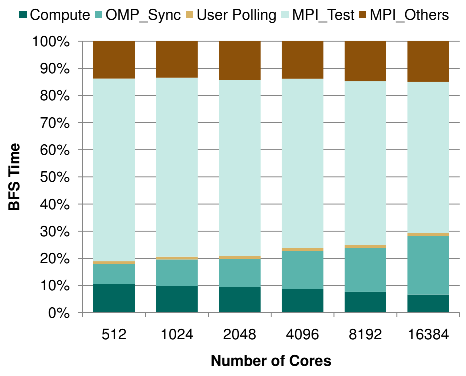
Fig. 11: Distribution of messages according to their content in a weak-scaling experiment with problem sizes from 24 to 30 and with 256 edges per message. *Empty* messages stem from `Synchronize`, and *incomplete* messages result from flushing the last vertices inside the buffers at the end of each level.



(a) Performance comparison

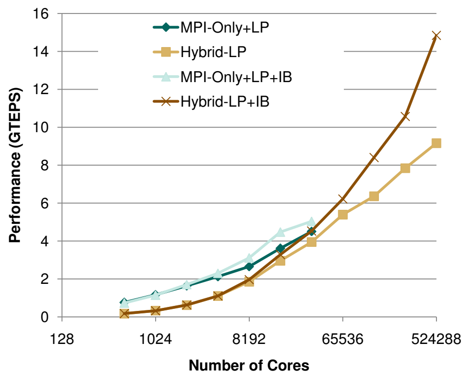


(b) Profile for MPI-only

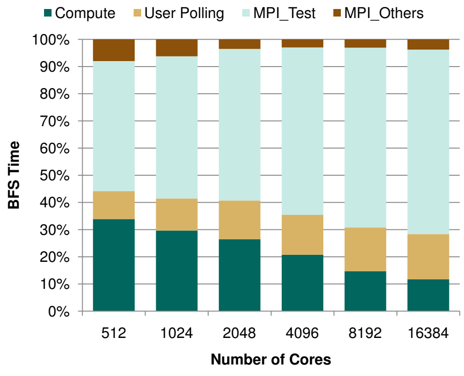


(c) Profile for Hybrid

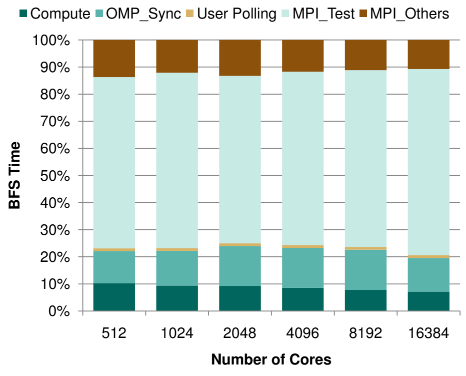
Fig. 10: Weak-scaling results and profiling after using the lazy polling method



(a) Performance comparison



(b) Profile for MPI-only



(c) Profile for Hybrid

Fig. 13: Weak-scaling performance and profiling results after using a nonblocking barrier to implement the global synchronization step

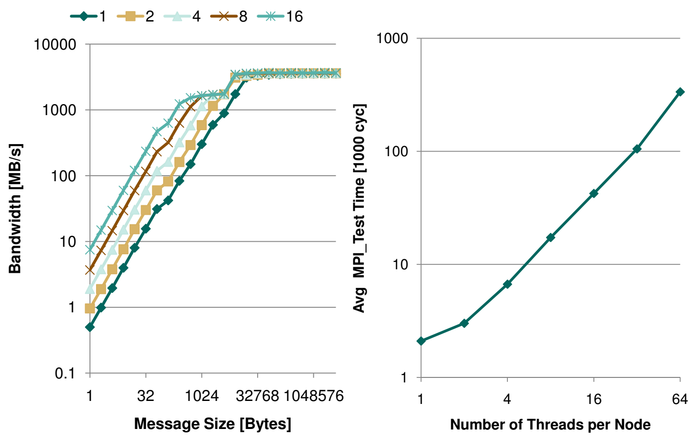
```
1 SYNCHRONIZE:
2     $done \leftarrow$ False;
3     OMPBARRIER();        ▷ Threads must complete sending
4     IBARRIER();          ▷ Executed by a single thread
5     while ($done$ = False) do
6         CHECKINCOMMSGS(); ▷ Make progress on Recv
7         $done \leftarrow$ TESTIBARRIER();     ▷ Single-threaded
8                                 ▷ barrier completion test
```

Fig. 12: New global synchronization implementation with the necessary threading extensions for the hybrid approach.

D. Reducing MPI Runtime Contention

Although the hybrid implementation is more scalable, it performs worse than the MPI-only method at a smaller scale. Despite the reduction in execution time, however, the execution breakdown in Fig.13b and Fig.13c shows that significant time is still spent in communication. In particular, the communication costs for both methods are comparable, a result that contradicts our analysis, in which we concluded that the communication costs should be higher for the MPI-only method.

We therefore hypothesized that either the multitheaded communication is serialized at the network level or the threads suffer contention at the software stack. To test the first hypothesis, we measured the concurrent bandwidth when varying the number of cores (one process per core). The results are shown in Fig.14a. We observe that by driving the communication using multiple cores, throughput can be improved by more than an order of magnitude as compared with a single core. This result disproves the first hypothesis. To test the second hypothesis, we measured the average latency of an `MPI_Test` call during a BFS run and plotted the graph in Fig.14b. We notice that the latency of `MPI_Test` is proportional to the number of threads and suggests contention in the software stack. We remind the reader that MPICH was built by using a *global critical section (CS)* internally to optimize the fast path. However, the proof of contention encourages the use of fine-grained critical sections. In our previous work, we had experimented with this concept in MPICH and obtained significant improvement in both multithreaded communication throughput and latency [10], [11], [12]. Because of the cost of implementing fine-grained critical sections, however, most MPI implementations rely on coarse-grained locking; to the best of our knowledge, only MPICH on Blue Gene systems supports fine-grained locking in production environments. To enable a more efficient multithreaded runtime on a wider range of architectures including commodity HPC clusters,



(a) Concurrent Bandwidth (b) `MPI_Test` performance

Fig. 14: (a) Point-to-point bandwidth on a Blue Gene/Q system with respect to the number of cores involved in the communication (data obtained by using the `osu_mbw_rm` benchmark from the OSU microbenchmarks suite, http://mvapich.cse.ohio-state.edu/benchmarks/). (b) `MPI_Test` performance with the BFS hybrid method when scaling the number of TPN with 512 cores

we investigated a different approach that reduces contention regardless of the CS granularity [13]. We showed that by using locks with CS arbitration policies different from that of Pthread mutex, contention can be reduced. Indeed, up to 5x performance improvement was observed with several benchmarks and applications.

Hence, since our platform is a Blue Gene/Q system, we rebuilt MPICH to support *fine-grained* (FG), or per-object, critical sections. We profiled the performance of `MPI_Test` with our hybrid method and show comparative results in Fig.15a. We notice that the performance improves considerably, with the cost of polling with `MPI_Test` being almost constant. The overall performance is improved, and the approach outperforms the MPI-only approach (Fig.15b), providing a total performance improvement of *35-fold* on 16K cores over the original MPI-only approach. The final profiling data in Fig.15c shows that the communication cost is reduced considerably, while the time spent outside the MPI runtime dominates the overall execution time. We note, however, that for fewer than 4K cores the hybrid approach is slightly worse than MPI-only, indicating that the hybrid method may still suffer from contention and incurs overheads, such as thread load-imbalance, shown in Fig.15c.

V. Related Work

Considerable studies have been conducted on the pros and cons of using MPI-only and shared-memory methods and their hybrid derivatives. Experiments with the NAS parallel benchmarks showed some conditions that favor the hybrid model, such as the amount of intranode parallelism and the speed of the network [14], [15], [16], [17], [18]. Chorley et al. analyzed the DL POLY molecular dynamic application and showed that the superiority of the hybrid model at large scale is due primarily to reduced communication [19]. The main conclusion we draw from this history is that the number of cases in favor of the hybrid model are increasing over time and are proportional to the scale of the testbed and the abundance of intranode parallelism.

However, studies where threads participate in communication are scarce, although this ability is gaining importance given the benefit of driving the network through multiple endpoints. For instance, Cappello et al. showed how threads can help with the internal MPI computation [14]. Others have proposed a solution to the idleness of threads during communication in a hybrid MPI-OpenMP model by exposing and exploiting computation-idle threads in order to improve communication performance [20]. In contrast to previous works, we have provided insight into multithreaded communication performance issues in the hybrid model. In addition, we stressed the core-to-core communication model of the MPI-only model and showed its overhead analytically and experimentally.

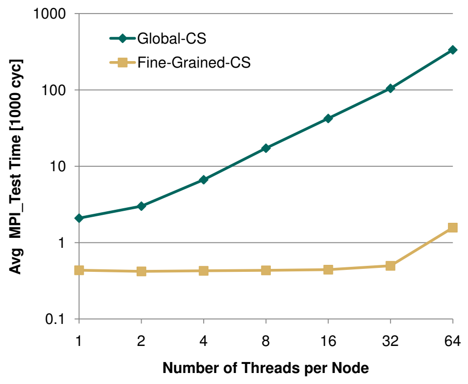
We considered MPI-only as using a single-copy model on shared memory. The recent MPI-3, however, supports better shared-memory through zero-copy extensions [21]. MPI shared-memory extensions offer some advantages similar to those of multithreading. A comparison between these models, however, requires thorough investigation and is out of the scope of this paper.

Many parallel BFS implementations have emerged during the past few decades, most of which use a hybrid MPI-threads model [6], [22], [23]. The goal of those works is to improve the performance of BFS, whereas we seek to characterize the trade-offs of different models. Moreover, those works require the `FUNNELED` mode, which from a programmability perspective adds a layer of complexity because of the heterogeneity created by classes of communication and computation threads and the producers-consumer relationship between them. We also point out that there are some performance drawbacks associated to this mode, such as losing some threads for computation and possibly underutilizing the network resources, since not all cores are driving the communication.

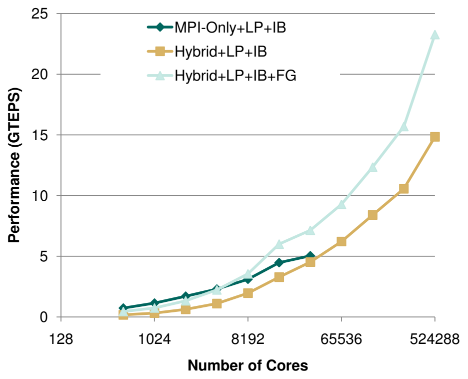
VI. Concluding Remarks

In this work, we studied the MPI-only and hybrid MPI+threads models using the BFS algorithm at very large scale. In our hybrid BFS implementation, threads are the main entities that handle both computation and communication concurrently. The motivation behind this design is to offer a better trade-off between memory usage, intranode parallelism, and communication performance than what the MPI-only model offers. As a result, we exposed many important parameters that users need to take into account when choosing the best model that fits the application and target platform. Such parameters are the scale of the target machine, where large-scale experiments favor the hybrid model because of the superior node-to-node communication model, and the drawbacks of runtime contention when communication is driven by multiple threads.

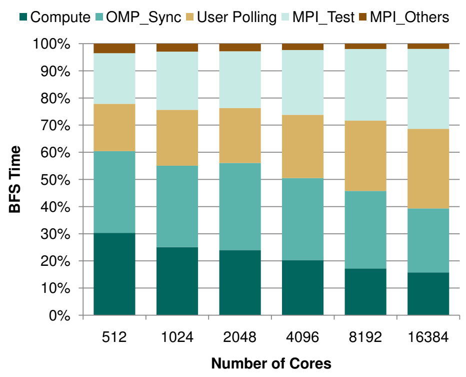
However, the hybrid model is not a silver bullet that fixes all scalability issues. Although it reduces some of the overheads associated with interprocess communication, thus delaying some of the scalability bottlenecks, it does not completely avoid them. We therefore proposed various enhancements and, through a detailed experimentation and analysis, demonstrated that our techniques can reduce the overheads at very large



(a) `MPI_Test` performance



(b) Performance comparison



(c) Profile for Hybrid

Fig. 15: (a) `MPI_Test` performance when scaling the number of TPN with 512 cores. (b) Weak-scaling performance comparison after using fine-grained critical sections (FG). (c) Execution breakdown after using fine-grained critical sections

scale and improve the performance of BFS by *35-fold* cores on 16K cores while scaling to 512K cores of a Blue Gene/Q system.

As future directions, we consider enlarging the study to other platforms, such as commodity HPC systems; investigating other shared-memory models (e.g., MPI-3 shared-memory extensions); and using other benchmarks and applications as case studies.

Acknowledgment

This material is based upon work supported by JST, CREST (Research Area: Advanced Core Technologies for Big Data Integration), by JSPS KAKENHI Grant Number 23220003, and by the U.S. Department of Energy, Office of Science, under Contract DE-AC02-06CH11357.

References

[1] "MPI: A Message-Passing Interface standard version 3.0," September 2012.

[2] L. Dagum and R. Menon, "OpenMP: An industry standard API for shared-memory programming," *IEEE Computational Science & Engineering*, vol. 5, no. 1, pp. 46–55, 1998.

[3] R. C. Murphy, K. B. Wheeler, B. W. Barrett, and J. A. Ang, "Introducing the Graph 500," *Cray User's Group (CUG)*, 2010.

[4] "The Graph 500 list," http://www.graph500.org/.

[5] T. Suzumura, K. Ueno, H. Sato, K. Fujisawa, and S. Matsuoka, "Performance characteristics of Graph500 on large-scale distributed environment," in *2011 IEEE International Symposium on Workload Characterization (IISWC)*, 2011, pp. 149–158.

[6] K. Ueno and T. Suzumura, "Highly scalable graph search for the Graph500 benchmark," in *Proceedings of the 21st International ACM Symposium on High-Performance Parallel and Distributed Computing*, 2012, pp. 149–160.

[7] J. Leskovec, D. Chakrabarti, J. Kleinberg, and C. Faloutsos, "Realistic, mathematically tractable graph generation and evolution, using Kronecker multiplication," in *2005 Knowledge Discovery in Databases (PKDD)*, 2005, pp. 133–145.

[8] V. Agarwal, F. Petrini, D. Pasetto, and D. A. Bader, "Scalable graph exploration on multicore processors," in *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, 2010, pp. 1–11.

[9] J. Chhugani, N. Satish, C. Kim, J. Sewall, and P. Dubey, "Fast and efficient graph traversal algorithm for CPUs: Maximizing single-node efficiency," in *2012 IEEE 26th International Parallel & Distributed Processing Symposium (IPDPS)*, 2012, pp. 378–389.

[10] P. Balaji, D. Buntinas, D. Goodell, W. Gropp, and R. Thakur, "Toward efficient support for multithreaded MPI communication," in *Recent*

[10] P. Balaji, D. Buntinas, D. Goodell, W. Gropp, and R. Thakur, "Toward efficient support for multithreaded MPI communication," in *Recent Advances in Parallel Virtual Machine and Message Passing Interface*. Springer Berlin Heidelberg, 2008, pp. 120–129.

[11] ——, "Fine-grained multithreading support for hybrid threaded MPI programming," *International Journal of High Performance Computing Applications*, vol. 24, no. 1, pp. 49–57, 2010.

[12] G. Dózsa, S. Kumar, P. Balaji, D. Buntinas, D. Goodell, W. Gropp, J. Ratterman, and R. Thakur, "Enabling concurrent multithreaded MPI communication on multicore petascale systems," in *Recent Advances in the Message Passing Interface (EuroMPI'10)*, 2010, pp. 11–20.

[13] A. Amer, H. Lu, Y. Wei, P. Balaji, and S. Matsuoka, "MPI+ threads: Runtime contention and remedies," in *The 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'15)*, 2015.

[14] F. Cappello and D. Etiemble, "MPI versus MPI+ OpenMP on the IBM SP for the NAS benchmarks," in *ACM/IEEE 2000 Conference on Supercomputing*, 2000, pp. 12–12.

[15] G. Jost, H. Jin, D. an Mey, and F. F. Hatay, "Comparing the OpenMP, MPI, and hybrid programming paradigms on an SMP cluster," in *European Workshop on OpenMP (EWOMP)*, vol. 3, 2003, p. 2003.

[16] E. Lusk and A. Chan, "Early experiments with the OpenMP/MPI hybrid programming model," in *OpenMP in a New Era of Parallelism*, 2008, pp. 36–47.

[17] R. Rabenseifner, G. Hager, and G. Jost, "Hybrid MPI/OpenMP parallel programming on clusters of multi-core SMP nodes," in *17th Euromicro International Conference on Parallel, Distributed and Network-based Processing*, 2009, pp. 427–436.

[18] X. Wu and V. Taylor, "Performance characteristics of hybrid MPI/OpenMP implementations of NAS parallel benchmarks SP and BT on large-scale multicore supercomputers," *ACM SIGMETRICS Performance Evaluation Review*, vol. 38, no. 4, pp. 56–62, 2011.

[19] M. J. Chorley and D. W. Walker, "Performance analysis of a hybrid MPI/OpenMP application on multi-core clusters," *Journal of Computational Science*, vol. 1, no. 3, pp. 168–174, 2010.

[20] M. Si, A. J. Peña, P. Balaji, M. Takagi, and Y. Ishikawa, "MT-MPI: Multithreaded MPI for many-core environments," in *Proceedings of the 28th ACM international conference on Supercomputing (ICS)*, 2014, pp. 125–134.

[21] T. Hoefler, J. Dinan, D. Buntinas, P. Balaji, B. Barrett, R. Brightwell, W. Gropp, V. Kale, and R. Thakur, "MPI+ MPI: A new hybrid approach to parallel programming with MPI plus shared memory," *Computing*, vol. 95, no. 12, pp. 1121–1136, 2013.

[22] N. Satish, C. Kim, J. Chhugani, and P. Dubey, "Large-scale energy-efficient graph traversal: A path to efficient data-intensive supercomputing," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis (SC)*, 2012, p. 14.

[23] H. Lv, G. Tan, M. Chen, and N. Sun, "Understanding parallelism in graph traversal on multi-core clusters," *Computer Science-Research and Development*, vol. 28, no. 2-3, pp. 193–201, 2013.