Scalable Hybrid Implementation of Graph Coloring using MPI and OpenMP

Ahmet Erdem Sarıyüce*†, Erik Saule*, and Ümit V. Çatalyürek*‡
* *Department of Biomedical Informatics*
† *Department of Computer Science and Engineering*
‡*Department of Electrical and Computer Engineering*
The Ohio State University
Email: {aerdem,esaule,umit}@bmi.osu.edu

***Abstract*—Graph coloring algorithms are commonly used in large scientific parallel computing either for identifying parallelism or as a tool to reduce computation, such as compressing Hessian matrices. Large scientific computations are nowadays either run on commodity clusters or on large computing platforms. In both cases, the current target platform is hierarchical with distributed memory at the node level and shared memory at the processor level. In this paper, we present a novel hybrid graph coloring algorithm and discuss how to obtain the best performance on such systems from algorithmic, system and engineering perspectives.**

***Keywords*-Graph algorithm; Graph coloring; Distributed Memory; Shared Memory; Hybrid programming**

I. Introduction

Graph coloring is a combinatorial problem that consists in assigning a color, a positive integer, to each vertex of the graph so that every two adjacent vertices have a different color. The graph coloring problem has been shown to be a critical ingredient in many scientific computing applications such as automatic differentiation [1], printed circuit testing [2], parallel numerical computation problems [3], register allocation [4], and optimization [5].

Today's large scientific computing applications are typically executed on large scale parallel machines for mainly two reasons: to reduce the execution time by leveraging parallelism, and to process large volume of datasets that do not fit to the memory of a single node. While running such applications, in order to execute graph algorithms that are part of the application one can use one of the two following approaches. The graph can be collected on a single node, provided it is small enough to fit in the memory, and execute a sequential version of the algorithm. Or execute a distributed memory version of the graph algorithm. In many cases, the former is either infeasible due to memory limitations, or not efficient [6].

The advent of multicore architectures significantly increased the number of processing units within a single machine. Most supercomputers nowadays provide more than four processing cores per node, and eight to sixteen cores per node are fairly common as well[1]. Intel recently announced the Many Integrated Core (MIC) architecture which should provide more than fifty cores within a single chip. These architectural developments shifted the supercomputer from distributed memory machines to hierarchical memory machines where the memory is distributed at the node level but shared at the core level.

To keep the performance best, one can not ignore the improvement made possible by having multiple processing units within a single node. Hybrid systems have flourished in computation-intensive areas such as linear algebra [7], multiple sequence alignment [8] and parallel matrix-vector multiplication [9] which report significant performance improvements. To the best of our knowledge, graph algorithms have not been considered for scalable hybrid processing, which will be the main focus of this work. The reason to undertake such a challenging task is that distributed systems are not ideal platforms for graph algorithms [10], furthermore, distributed memory graph coloring techniques (in fact almost all graph algorithms) suffer severe performance drawbacks when trying to use all the processing units of multicore clusters using message passing libraries [11].

In this paper, we present the design and the development of a hybrid coloring algorithm. We provide a thorough experimental performance analysis of a careful implementation on a multicore cluster. Our study is performed within the framework of a widely used library, Zoltan [12]. We highlight the details that appear in a production quality general purpose library to illustrate the need for the algorithm engineering necessary to obtain the best possible performance in real world settings.

We discuss the related work in Section II. Then we present the internal architecture of the coloring module of Zoltan in Section III and explain how to adapt it for hybrid computation. Section IV presents our thorough experimental performance analysis starting with the different parameters one should consider when deploying an hybrid graph algorithm and how to get to a hybrid implementation that leads to 20% to 30% improvement over the distributed memory implementation. Finally, in Section V, we draw more general conclusion to hybrid graph algorithms and discuss future works.

[1]http://www.top500.org/

II. Preliminaries

A. Generalities

A coloring of a graph is an assignment of integers (called colors) to vertices such that no two adjacent vertices will have the same integer. The aim is to minimize the number of different colors assigned to the vertices. The problem has been known to be NP-Complete [13] and recently, it has shown that for all $\epsilon > 0$, it is NP-Hard to *approximate* the graph coloring problem within $|V|^{1-\epsilon}$ [14].

Yet simple algorithms are known to provide almost optimal coloring for a majority of common graphs [5]. The sequential greedy coloring presented in Algorithm 1 is the most popular technique for graph coloring [15], [16]. This algorithm simply visits the vertices of the graph in some order and assign to each vertex the smallest permissible color. The order of traversal of the graph is known to be of importance for reducing the number of colors used and many heuristics have been developed on that premise [17], [1].

Algorithm 1: Sequential greedy coloring.

```
Data: $G = (V, E)$
for each $v \in V$ do
    for each $w \in adj(v)$ do
        forbiddenColors[color[$w$]] $\leftarrow v$
    color[$v$] $\leftarrow \min\{i > 0 :$ forbiddenColors[$i$] $\neq v\}$
```

B. Parallel Graph Coloring Algorithms

We can classify the parallel graph coloring algorithms in three categories.

The first category of algorithms relies on finding a maximal independent set of vertices in the graph. An independent set of vertices does not contain any two vertices having an edge between them; such a set is said to be maximal if no other vertices can be added to that set while keeping it independent. Luby's algorithm [18] starts by assigning a random number to each vertex; then it finds a vertex such that its random number is larger than all of the neighbors, removes it and removes its neighbors. Instead of removing that vertex, one can give it the smallest color, and the algorithm becomes a parallel graph coloring algorithm. Many of distributed memory parallel graph coloring algorithms [3], [19], [20] relies on this technique.

The second category of coloring algorithms relies on speculative coloring technique [21], [22], [6], [23]. In the simplest form [21], each processor tentatively colors parts of the graph independently of the other ones using the sequential greedy algorithm. Once the graph has been fully colored, all the vertices of the graph are considered once again to make sure all adjacent vertices are colored with different colors. In case a conflict is detected, a global ordering on the vertices (usually determined by random numbers) is used to mark one of the vertex to be recolored. And lastly, marked vertices are recolored sequentially. The method was refined in [22] by using smart block based partitioning and by executing both coloring and conflict detection phase using parallel OpenMP construct. [23] presents more improvement to that shared-memory algorithms by introducing the computation of efficient ordering in parallel and applying the algorithm to distance-2 coloring.

Bozdağ et al. [6] made multiple improvements to the speculative coloring idea to make it suitable for distributed memory architectures. One of the extensions was replacing the sequential recoloring phase with a parallel iterative procedure. Many of the other extensions were driven by performance needs in a distributed-memory setting, like during the coloring, exchange of coloring information is done in a bulk-synchronous way to reduce the communication overhead. The implementation in Zoltan [12], and hence our work is based on the coloring framework developed by Bozdağ et al. [6] and is explained in details in Section III.

The third category, which includes most recent development in coloring algorithms, is dataflow coloring algorithms which has been originally designed for Cray XMT [24]. The main difference of dataflow coloring algorithm is how the coloring of a vertex is initiated or *triggered* (and in some versions, how these coloring tasks are assigned to a processing element). By establishing a total ordering on vertices (such as using vertex IDs) one can color the vertex with highest priority (say vertex with lowest ID) first. In other words, a vertex can only be colored when its neighbors with higher priority have been colored. Algorithmically, coloring a vertex triggers the coloring of the adjacent vertices with lower priorities (that is vertices with higher IDs). We should note that vertices with lower priorities can be concurrently colored with higher priority vertices, as long as they are not depending on them. In other words, coloring of the vertices are driven by the flow of the data, i.e., when a color is assigned to a vertex. Although this algorithm is more work efficient than speculative algorithms by avoiding the conflict resolution phase, it relies on low-level hardware specific intrinsics to be efficient which does not exists on the architectures targeted in this study.

C. Hybrid Algorithms

Many algorithms have been developed for hybrid systems [7], [25], [26], [8]. Baker et al. [7] experiments algebraic multigrid on a hybrid platform and discusses the challenges faced. They introduce the first comprehensive study of the performance of algebraic multigrid on three leading HPC platforms. They claim that a general solution for obtaining best multicore performance is not possible without taking into account the specific target architecture including node architecture, interconnect and operating system capabilities. White et al. [26] discusses overlapping the computation and communication on hybrid parallel computers for the advection problem. Overlapping MPI communication with computation does not give significant performance improvements for their test case, but tuning the number of OpenMP threads per MPI process is important for performance. Macedo et al. [8] present a multiple sequence alignment problem in hybrid context and provide a parallel strategy to run a part of their algorithm in multicore environments. They also discuss the need for powerful master node, which is responsible for communication, and appropriate task allocation policy. Schubert et al. [9] discuss parallel sparse matrix-vector multiplication for hybrid MPI/OpenMP programming. They analyze single socket baseline performance with respect to architectural properties of multicore chips.

As an hybrid approach for graph algorithm, Kang and Bader [25] investigate the large scale complex network analysis methods on three different platforms: a MapReduce cluster, a highly multithreaded system and a hybrid system that uses both simultaneously. In that work, Kang and Bader show different approaches for the different architectures. They explain that performance and program complexity are highly related with the conformity of workload's computational requirements, the programming model and the architecture. Their work shows the synergy of the hybrid system in the context of complex network analysis and superiority of hybrid system over simply using a MapReduce cluster or a highly multithreaded system. Their work discusses a graph problem in hybrid context, but does not give scalability results in terms of distributed settings.

To the best of our knowledge, there is no work applied on a graph algorithm investigating the scalability on multicore hybrid systems. From that perspective, our work is the first hybrid parallel graph algorithm study investigating the scalability.

III. Algorithms

A. Distributed-Memory Coloring

Bozdağ et al. [6] present a distributed-memory parallel graph coloring framework which was the first to show a parallel speedup. Our work is based on the implementation of that algorithm in Zoltan [12], an MPI-based C library for parallel partitioning, load balancing, coloring and data management services for distributed memory systems. We implemented the hybrid graph coloring algorithm inside the distributed coloring framework of Zoltan. Here we will explain the distributed algorithm first and then we will give details about the implementation of the shared memory algorithms which are important for a proper execution on a hybrid system.

The graph is first built by Zoltan using programmer defined callbacks inside each MPI process. That is to say, the graph is distributed onto the MPI processes according to the distribution of the user's data. In other words, Zoltan does not choose the distribution unless it is requested by the application. In the coloring framework, each vertex belongs to a single MPI process. The information of an edge is available to an MPI process only if one of the end point vertices of the edge is owned by that process. In other words, if both vertices of an edge are owned by same process, then only that process has the information of that edge. Otherwise, if the edge is connecting two vertices owned by different processes, then both processes have the information about that edge. In the course of the algorithm, each process is responsible for coloring its own vertices. If all the neighbors of a given vertex are owned by the same process, then this vertex is an *internal* vertex. Otherwise, if any of the neighbor vertex belongs to a different MPI process, then this vertex is a *boundary* vertex.

The vertices can be colored in five different vertex visit orders [6], [11]. In this work, we focus on the ordering called *Internal First* which is the fastest one. It consists in coloring first the internal vertices in each process. Since they are internal, their coloring can be done concurrently by each process without risking to make an invalid decision.

The coloring of the boundary vertices is done in multiple rounds. In each round, each process greedily colors all of its uncolored vertices. Then, possible conflicts at boundary vertices are detected by each process. If a conflict is detected at an edge, one of the vertices of that edge is selected to be recolored in the next round. This selection is based on random keys that are associated with each vertex. This association of the random key is done before the coloring; each vertex is guaranteed to have a unique random key so that it provides a total ordering on the vertices in all the processes.

To reduce the number of conflicts at the end of each round, communication must be frequent between the MPI processes, but it should not be too frequent, otherwise the communication latency of the distributed system will be the bottleneck. Therefore, the coloring of the vertices are organized in *supersteps*. In each superstep, each MPI process colors a given number, called *superstep size*, of its own vertices, then exchange with other MPI processes the current color of the boundary vertices. The procedure is said to be *synchronous*, if all the processes are coloring the same superstep at each time. In other words, one process does not start one superstep before its neighbors finished the previous superstep. In our work, coloring is done in synchronous supersteps[2]. The superstep size has an important impact on the quality and the runtime of the entire coloring procedure since a too small value leads to too many synchronizations, while a too high value increases the number of conflicts and therefore the amount of redundant work.

It is important to understand that each MPI process

[2]Zoltan also supports asynchronous supersteps, but we do not investigate this possibility in this work. The interested reader is referred to [6].

communicates with its neighbor processes using dedicated messages. Different communication settings were investigated in [6] and found that a customized communication scheme leads to the best performance on medium to large scale systems.

An important implementation detail concerns the order of the vertices in memory. Since the graph comes directly from the user, there is no guarantee on the relative order of internal and boundary vertices. Having the internal vertices numbered one after the other, and similarly having boundary vertices numbered consecutively helps in utilizing caches and also could help reducing the amount of work, by only traversing the vertices that are needed to be processed. Therefore, before doing anything else, the coloring framework in Zoltan starts by reordering the graph in memory, that is to say it puts the boundary vertices first and the internal vertices last. In the process, the list of neighbors of a single node is also rearranged in that order. That way, it is easy to access only the internal neighbors of a vertex, or only to its local boundary neighbors, or its external boundary neighbors (neighbors that are not owned by the current process). This phase is relatively expensive but it is important to achieve the highest performance.

Zoltan has a random key construction phase before doing coloring to obtain a new total ordering among vertices that is not influenced by the natural ordering. This construction is done by first hashing the global ids of the vertices and then calling a random function with that hash as a seed. If two vertices happen to have the same random key, the tie is broken based on the global ids. Recall that the random keys are used to decide how conflicts are resolved. The overall algorithm is prone to some worst case that depends on how conflicts are resolved. Using randomized values makes the worst cases less likely.

B. *Hybrid Coloring*

There are many sources of shared memory parallelism within the scope of one MPI process. Exploiting properly each source of parallelism proved to be key in achieving the best performance. In our implementation, the parallelism inside an MPI process is achieved with OpenMP.

First of all, the construction of the random keys is done in parallel using the OpenMP parallel for construct. Originally, Zoltan was using a stateful random generator which was not thread-safe. We improved the random generator and made it thread-safe.

Reordering the vertices can also be done using multiple threads. In reordering, there are three main operations: determining and counting the boundary vertices, clustering the visit array so that boundary vertices are placed first and internal vertices are placed last, and changing the adjacency lists of each vertex so that boundary vertices appear first. The first and third operation can be executed concurrently by multiple threads provided they operate on different vertices. A simple parallel for construct allows to process them in parallel. The boundary vertices are determined and computed and stored independently by the threads. After the parallel execution of the loop their number is summed. To be able to execute the second operation efficiently in parallel, one must enforce the allocation of iterations to the threads to be static while counting the boundary vertices. Indeed, the position where a thread should move a vertex is easily computed if the number of boundary vertices with an ID smaller than the ID of the vertex being considered is known. The best way to obtain that information is to keep a static scheduling policy and to reuse the information contained in the execution of the counting of the boundary vertices. Two more important details appear. The first and third operations are independent from each other and can therefore be merged into a single parallel loop in order to reduce scheduling overhead. And to avoid false sharing the count of the number of boundary vertices processed by each thread must be allocated on different cache lines.

Each time vertices are colored, they are colored with the same thread-parallel procedure we now describe. The coloring is simply done by partitioning the vertices to different threads using the parallel for construct. Each thread needs its own mark array and a variable to keep track of the highest color it uses. Keeping the memory used by each thread on different cache lines avoids false sharing.

If there is more than one thread in the process, we need to verify whether there are some conflicts or not. Each thread verifies a part of the vertices and if there is a conflict and its random key is less than the other vertex random key, it is marked for recoloring. A list of vertices to recolor is kept by each thread and is aggregated once all the threads are done detecting conflicts.

IV. Design and Experiments

A. *Experimental Settings*

All of the algorithms are tested on an in-house cluster composed of 64 computing nodes. Each node has two Intel Xeon E5520 (quad-core clocked at 2.27GHz) processors, 48GB of main memory, and 500 GB of local hard disk. Nodes are interconnected through 20Gbps DDR InfiniBand. They run CentOS 6.0 the Linux kernel 2.6.32. The code is compiled with Intel C Compiler 12.0 using the -O2 optimization flag. Two implementations of MPI are tested: MVAPICH2 version 1.6 and OpenMPI version 1.4.3. However, mainly MVAPICH2 is used. The experiments are run on up to 32 nodes except for Figure 2 where 64 nodes are used for the experiment. For each run, we present how many processes per node are used as well as how many threads per process. Each node has 2 sockets, each socket has 4 cores and each core has 2 hyperthreads. There is an L1 and L2 caches per each core of size 32 KB and 256 KB, respectively, and there is an 8 MB L3 cache per socket.

Name	$\|V\|$	$\|E\|$	Δ	#colors	seq. time
auto	448K	3.3M	37	13	0.1103s
bmw3_2	227K	5.5M	335	48	0.0836s
hood	220K	4.8M	76	40	0.0752s
ldoor	952K	20.7M	76	42	0.3307s
msdoor	415K	9.3M	76	42	0.1458s
pwtk	217K	5.6M	179	48	0.0820s

Table I
PROPERTIES OF REAL-WORLD GRAPHS

The experiments are run on six real-world graphs which come from various application areas including linear car analysis, finite element, structural engineering and automotive industry [22], [27]. They have been obtained from the University of Florida Sparse Matrix Collection[3] and the Parasol project. The list of the graphs and their main properties are summarized in Table I. The number of colors obtained with a sequential run is also listed in the table. Finally, the time to compute the coloring using a sequential greedy algorithm is given.

In a real-word application context, the application data are partitioned according to the user's need. For our test, we partition the graphs using two different partitioners built in Zoltan. The Parallel HyperGraph partitioner (PHG) uses an hypergraph model to produce well balanced partitions while keeping the total inter-process communication volume small. Block partitioning is a simple partitioner based on vertex IDs. Although it produces well-balanced partitions in terms of number of vertices, since the actual work depends on the size of the adjacency lists of vertices, the load can be imbalanced and also could incur high communication cost.

For all the experiments, we will present either the runtime of the method or the number of colors it produces. Since all the graphs we consider show the same trends, their results are aggregated as follows. Each value is first normalized and then the normalized values are aggregated using a geometric mean. Normalization bases may vary from figure to figure, but it will be mentioned for each figure.

B. Scalability of the Distributed Memory Implementation

In a recent work, we showed that distributed memory graph coloring techniques suffer severely when trying to use all processing elements in a multicore cluster using a message-passing programming model [11]. Figure 1 shows the normalized time obtained when increasing the number of MPI processes using different process to processor allocation policies. Normalization is done with respect to the runtime obtained using one MPI process. The 1ppn allocation policy first allocates one process on a different node until 32 processes are used (where they are allocated on 32 different nodes) and then allocates a second process per node until 64 processes are used, and so on. The 2ppn allocation policy allocates processes by group of two so that when 4 processes

[3]http://www.cise.ufl.edu/research/sparse/matrices/

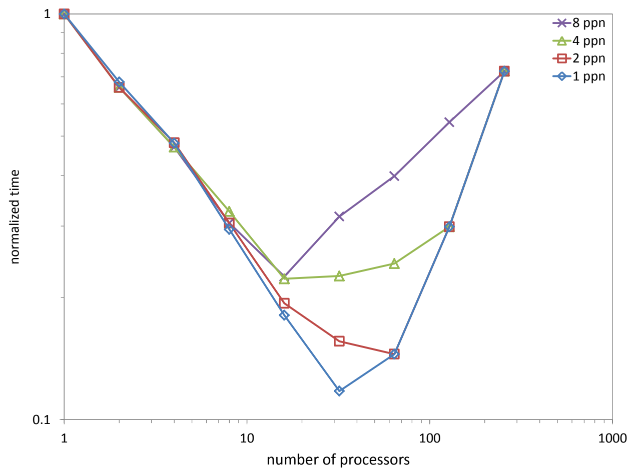


Figure 1. Impact of the distributed memory process allocation policies on real-world graph

are used only two nodes are used. Once it allocates 64 processes, it used all the nodes and start allocating processes to the first nodes again. (We will use ppn for "MPI processes per node" from now on). In this experiments, block partitioning is used and the superstep size is set to 1000. Figure 1 shows that the runtime of the 1ppn allocation policy dramatically increases as soon as more than one process per node is used (that is to say with 64 processes). The 8ppn allocation policy scales gracefully until 16 processes are used; that is to say, until two nodes are fully used. As soon as 32 processes are used, that is to say four nodes, the runtime starts to dramatically increase. For the very synchronized and small messages that are exchanged by the distributed memory coloring algorithm, the MPI subsystem is not capable of transferring the messages fast enough when multiple processes reside on a single node.

One can wonder whether it is a defect of a particular MPI implementation or whether the cause is deeper. We compared OpenMPI 1.4.3 and MVAPICH2 1.6 using the 1ppn and 8ppn process allocation policy; results are shown in Figure 2. Note that, in this experiment normalization is done with respect to the sequential greedy coloring time and experiment is conducted up to 64 nodes. The two implementations of MPI show some performance difference but the trend we are interested in is still present. Both show significant performance degradation when more than two nodes are used with more than one process per node.

The conclusion of those two experiments is that in the current state, the distributed memory-only implementation of the coloring algorithm can not efficiently use clusters of multicore. Hence, the importance of providing an efficient hybrid implementation that will allow to properly exploit the full potential of such systems.

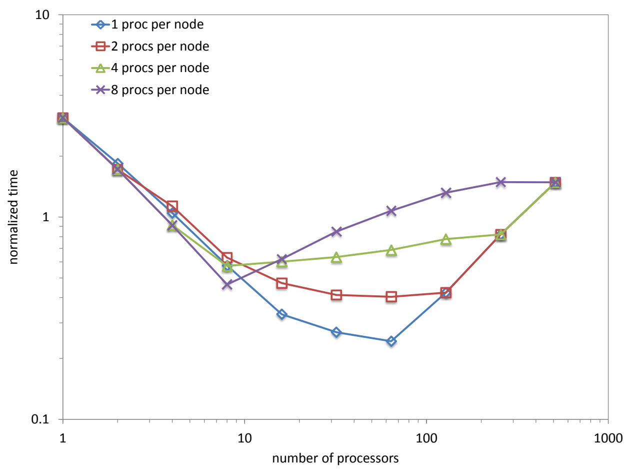


Figure 2. Comparison of different MPI implementations on 1 ppn and 8 ppn configurations

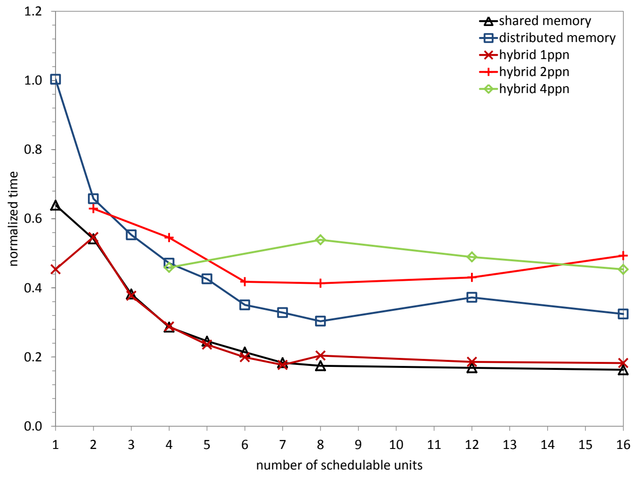


Figure 3. Comparison of shared memory, distributed memory and hybrid implementations with different ppns in a single node with block partitioning

C. *Single Node Experiments*

Figure 3 presents the results of the hybrid implementation in a single node compared to our shared-memory code and distributed-memory algorithm. The hybrid implementation is run with 1ppn, 2ppn and 4ppn configurations. The x-axis is named *number of schedulable units* (SU) which means either a thread in shared-memory and hybrid cases or a process in the case of distributed memory code. For instance, for the hybrid implementation with 2 ppn, if each process uses 4 threads, then 8 threads are used in total and the result is reported as 8 SUs. Therefore, some data point are not reachable, e.g., 1 SU for the hybrid 2ppn configuration. We did not run the distributed memory cases on more than 8 SUs since there are only 8 physical cores per node in our test cluster where 2 hyperthreads reside per core. In this experiment, block partitioning is used for distributed memory and hybrid implementations and the superstep size is set to 1000. Thread affinity and the OpenMP scheduling policy are left to their default values (which are no affinity is set and static scheduling policy). Normalization is done with respect to the execution of one MPI process on the distributed memory implementation.

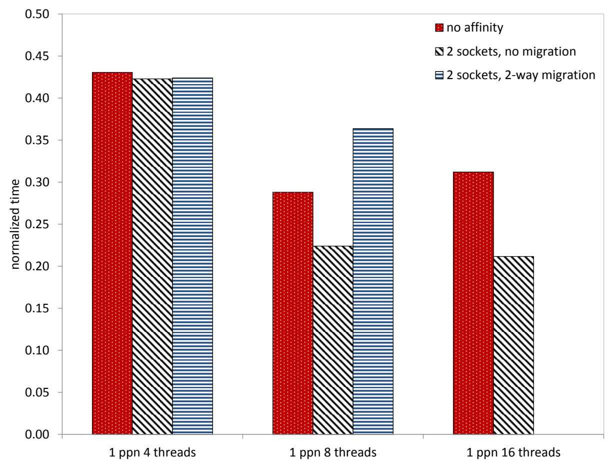
Figure 3 shows that hybrid implementation with 1ppn allocation policy gives very close results with shared memory implementation and they are clearly best. Then comes the distributed memory-only implementation, and finally hybrid implementations with the 2ppn and 4ppn. We should note that, the reordering process, which is explained in Section III-B, is disabled in when a single process is used. Hence the performance of hybrid implementation will be slightly worse when more processes are used.

Next, we investigate the use of thread affinity in a single node to decide whether migration and hyperthreading is beneficial for different configurations. Remember that, in our cluster machines each node has 2 sockets, each socket has 4 cores and each core has 2 hyperthreads. There is one L3 cache per socket and L1 and L2 caches per core. Figure 4(a) shows the results for the migration experiment. The ”no affinity” lets the operating system place the thread as it sees fit. The ”2 sockets, no migration” policy leaves no choice to the system scheduler by assigning each thread of a process to a different socket while fixing the (logical) threads to a given physical hyperthread. On the contrary, the ”2 sockets, 2-way migration” policy forces the threads of a process to all be scheduled on the different cores of the sockets while allowing the (logical) thread to migrate from one hyperthread to the other one. In the experiment, these configurations are compared under different number of threads.

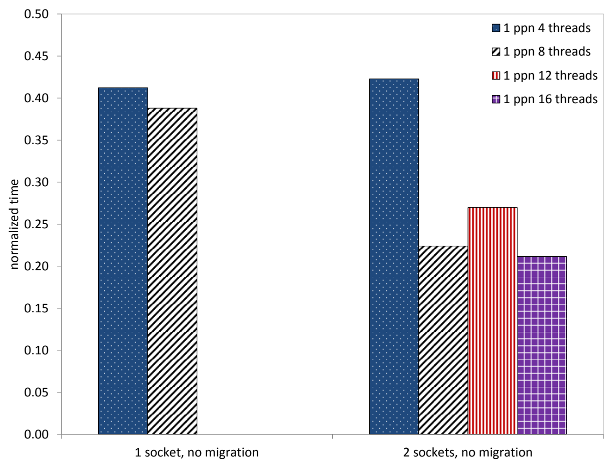
The results show that prevention of migration by pinning logical threads to hyperthreads gives the best performance; indeed allowing migration inside a core never improves performance. The improvement carried by setting the thread mapping over letting the system choose the thread mapping can be as high as 35%. The results for the hyperthreading experiment are shown in Figure 4(b). Using a single socket, the runtime decreases when using all the hyperthreads(1ppn 8 threads) compared to using a single hyperthread per core(1ppn 4 threads). Using 2 sockets, using hyperthreading (1ppn 16 threads) improves the runtime compared to not using it (1ppn 8 threads). However, partially using hyperthreading (1ppn 12 threads) degrades performance. This latter effect might be due to the static scheduling policy which induces load imbalance at the core level. Overall, using hyperthreading improves performance.

D. *Multiple Node Experiments*

When using OpenMP, it is usually important to properly set the scheduling policy. The next experiment investigates

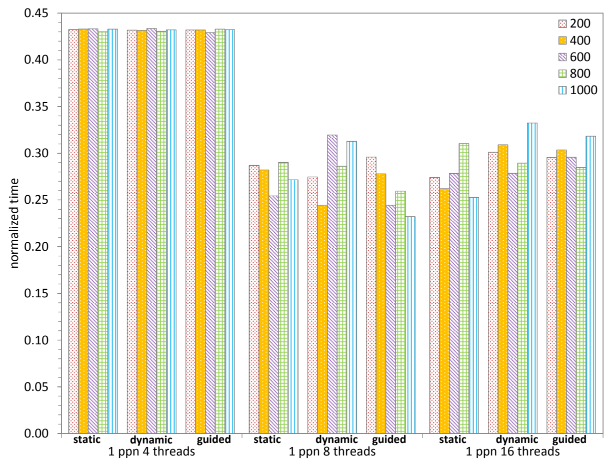


(a) Impact of Migration

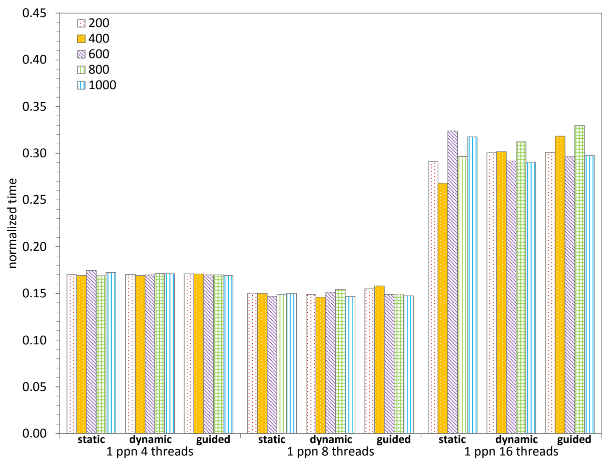


(b) Impact of Hyperthreading

Figure 4. Impact of affinity policies on some configurations in a single node



(a) 1 node



(b) 8 nodes

Figure 5. Impact of OpenMP scheduling policy on hybrid implementation

the static, dynamic and guided scheduling policies with chunk sizes of 200, 400, 600, 800 and 1000 on 1 node and 8 nodes for the 1ppn 4 threads, 1ppn 8 threads and 1ppn 16 threads cases. In this experiment, the superstep size is 1000, no affinity is set and normalizations are done with respect to the execution time of one MPI process using the distributed memory-only implementation. Results are presented in Figure 5.

On 1 node, Figure 5 shows that the OpenMP scheduling policy does not make much of a difference when 4 threads are used. Using 8 threads, some differences appear and the runtime based "Guided, 1000" and "Dynamic, 400" give best results. Using 16 threads, "Static, 1000" leads to better results. Overall the differences are difficult to predict. On 8 nodes, the scheduling policy does not bring major differences unless 16 threads are used where the runs are quite difficult to predict again.

Figure 6 shows the results of the study of the impact of the different supersteps sizes up to 8 nodes with 1ppn 16 threads case where affinity is set properly. Normalizations are done with respect to one MPI process of the distributed memory-only implementation. As can be seen from the figure, superstep size 500 is slower. Superstep sizes of 1000, 2000 and 4000 leads to only marginally different runtimes despite a larger superstep size make the run faster. We also know that, increasing superstep size brings more conflicts in our algorithm and tend to degrades the quality of coloring. For this reason, we believe a superstep size of 1000 balances reasonably the quality of the coloring and runtime performance.

Until now, we have experimented the variations of several parameters to see their effects on hybrid coloring. From

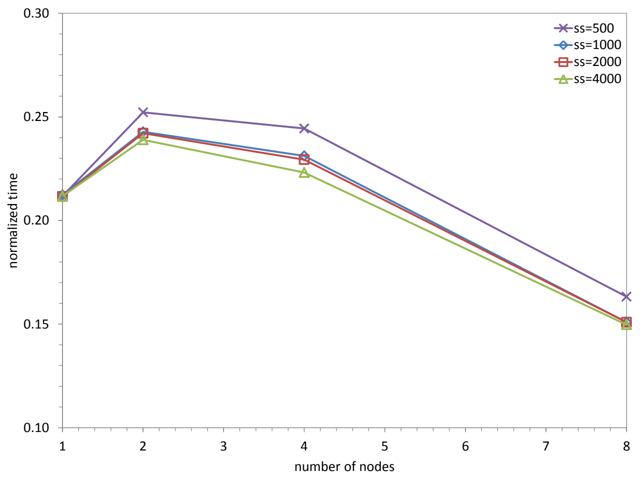


Figure 6. Impact of superstep size on hybrid implementation up to 8 nodes

now on, we combine the best results we obtained by tuning the mentioned parameters. For example, when we present distributed memory-only result on say 4 nodes, we have tested distributed memory code from 4 processes (1ppn policy) to 4×8 processes (8ppn policy) and simply present best possible result one could achieve with distributed memory code on 4 nodes. In other words, configuration selected for distributed memory-only code from one node to another could be different. Indeed, for 1, 2, 4, 8, 16 and 32 nodes, best configurations are 8ppn, 8ppn, 4ppn, 2ppn, 2ppn and 1ppn, respectively. We would like to be fair to both implementations and compare only their best performance.

All previous experiments used block partitioning. The next experiments present the impact of the graph partitioning of the runtime of the hybrid coloring algorithm. Figure 7 shows these results up to 32 nodes. This chart simply compares the best results one can obtain with hybrid 1ppn, hybrid 2ppn and distributed memory-only implementation on a given number of nodes. The two partitioners tested are PHG (Parallel Hypergraph Partitioning) and block partitioning. Normalizations are done with respect to eight MPI processes in one node case using the distributed memory-only implementation. The result indicates that PHG partitioning provides a better runtime performance for hybrid 1ppn and distributed memory-only by about 25% and about 20% respectively. Similar values are observed for the hybrid 2ppn configuration.

A typical parallel program, running on a multi-core cluster, is expected to utilize all the processing units available. So, for our cluster, an MPI program is usually run with 8 processors per node configuration and a hybrid program is run with x processors per node and y threads per each process, where x * y is 8. We compared the hybrid and distributed memory-only implementations of graph coloring in Figure 7. This figure is the first one showing the performance of hybrid algorithm in large scale. When we compare the typical configurations, hybrid implementations are far better than distributed memory-only 8ppn implementations, 8x faster for block partitioning and 6x faster for PHG partitioning. Furthermore, hybrid implementations are also better than distributed memory-only 1ppn implementations. For block partitioning, hybrid 1ppn outperforms the distributed memory-only 1ppn implementation by 6% on 8 nodes, and up to 24% on 32 nodes. The hybrid 2ppn configuration outperforms the distributed memory-only implementation in almost all number of nodes, by 47% on 8 nodes and by 15% on 32 nodes. Notice that because the partitioning is not taken into account in the runtime, using the PHG partitioner gives an advantage to the distributed memory-only implementation. Still, the hybrid 1ppn and hybrid 2ppn configurations obtain better runtime than the distributed memory implementation on 32 nodes and 16 nodes, respectively, with PHG partitioning. This result expresses in our opinion the superiority of hybrid graph algorithms in a large scale setting.

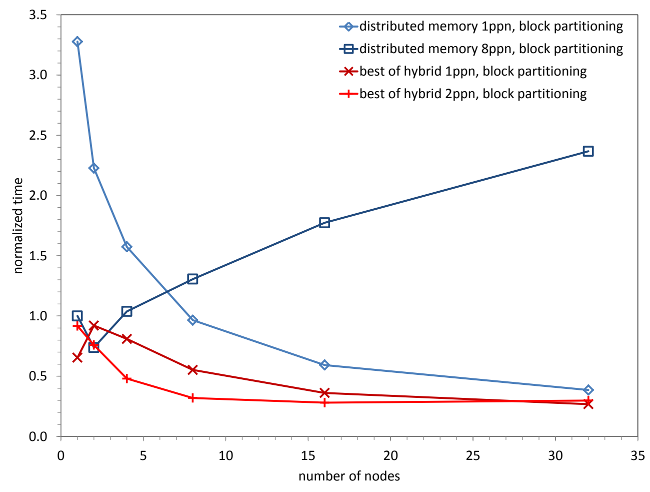
E. *Overall comparisons*

Figure 8 presents the runtime comparison between the typical distributed memory-only configuration (8 ppn), the best distributed memory-only configuration and the best hybrid configuration. The normalizations are performed with respect to the runtime of eight MPI processes on one node using the distributed memory-only implementation. Here, the best configuration is picked and might use different number of processes. For example, using 8 processes per node shows the best performance for the distributed memory implementation on 1 node while the hybrid implementation is better using a single process but 16 threads. The hybrid implementation is far better than typical distributed memory-only implementation. It also leads to better runtime than the best distributed memory-only implementation on all number of nodes (except on 2 nodes) with 20% to 30% of improvement.

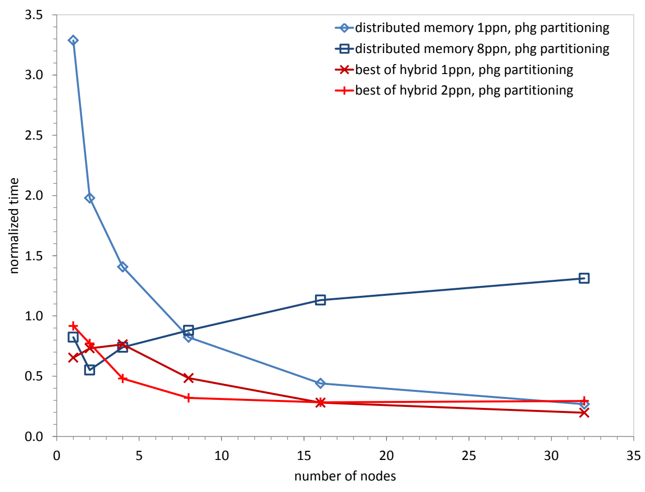
One can be interested to verify that the hybrid implementation did not significantly worsen the number of colors obtained by the algorithm. Actually, the hybrid implementation provides 4% to 7% less number of colors than the distributed memory-only implementation at large scale.

V. Conclusion

In this paper, we investigated the proper implementation of a graph coloring algorithm for hybrid systems. We showed how an existing distributed memory code base was carefully adapted to provide better performance for hierarchical multi-core architectures. The parameters affecting the performance of the hybrid execution have been investigated one by one in order to obtain the best possible performance. Despite the shared memory algorithm is not work efficient and the distributed memory algorithm benefits from a free partitioning,



(a) Block Partitioning



(b) PHG partitioning

Figure 7. Impact of partitioning types on hybrid implementation up to 32 nodes

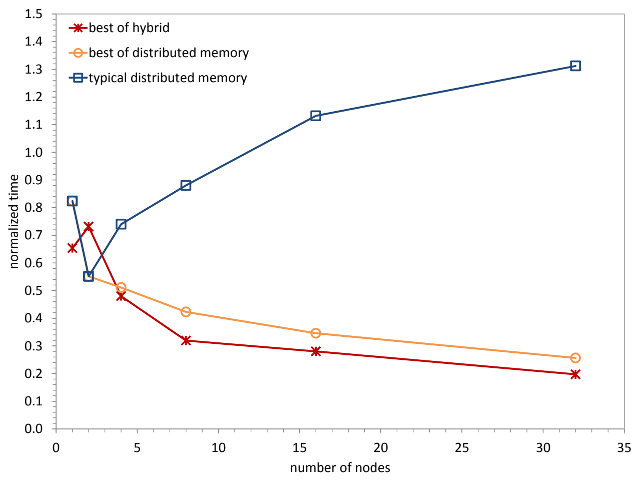


Figure 8. Runtime comparison of best of distributed memory-only and best of hybrid configurations

a careful implementation of the program for hybrid system and a proper evaluation of the parameters of the execution platform allow to obtain better performance over typical distributed memory-only usage by 6 times to 8 times. Hybrid coloring is even better than 1ppn distributed memory-only implementation by 20% to 30% while obtaining better number of colors. To the best of our knowledge, this paper is the first work on the hybrid implementation of a graph algorithm with full scalability tests.

We would like to highlight the importance of properly setting thread affinity. Letting the operating system schedule threads typically reduces performance significantly. Scheduling the threads of a single process so that they share a common cache usually improves performance. Also, it is important to note that hyperthreading is beneficial for hybrid parallelization of graph coloring.

Now that it is possible to utilize all the parallelism contained within a node without suffering from high communication cost, we plan to investigate ways to improve the quality of the solution in a hybrid setting. Implementation of ordering techniques such as Largest First and Smallest Last for hybrid systems will bring some new challenges such as coloring the graph with a pre-computed ordering. We are also planning to include the recoloring procedure presented in [11]. We only investigated the distance-1 coloring problem and the proper implementation of distance-2 coloring should be investigated as well. Last but not least, the implementation for hybrid system that perform coloring and distributed memory communication simultaneously has the potential for overlapping both operation and achieving higher performance.

Acknowledgment

This work was partially supported by the U.S. Department of Energy SciDAC Grant DE-FC02-06ER2775 and NSF grants CNS-0643969, OCI-0904809 and OCI-0904802.

References

[1] A. H. Gebremedhin, F. Manne, and A. Pothen, "What color is your jacobian? Graph coloring for computing derivatives," *SIAM Review*, vol. 47, no. 4, pp. 629–705, 2005.

[2] M. Garey, D. Johnson, and H. So, "An application of graph coloring to printed circuit testing," *Circuits and Systems, IEEE Transactions on*, vol. 23, no. 10, pp. 591–599, Oct. 1976.

[3] J. Allwright, R. Bordawekar, P. D. Coddington, K. Dincer, and C. Martin, "A comparison of parallel graph coloring algorithms," Northeast Parallel Architectures Center at Syracuse University (NPAC), Tech. Rep. SCCS-666, 1994.

[4] G. J. Chaitin, “Register allocation & spilling via graph coloring,” *SIGPLAN Not.*, vol. 17, pp. 98–101, Jun. 1982.

[5] T. F. Coleman and J. J. More, “Estimation of sparse Jacobian matrices and graph coloring problems,” *SIAM Journal on Numerical Analysis*, vol. 1, no. 20, pp. 187–209, 1983.

[6] D. Bozdağ, A. Gebremedhin, F. Manne, E. Boman, and Ü. Çatalyürek, “A framework for scalable greedy coloring on distributed memory parallel computers,” *Journal of Parallel and Distributed Computing*, vol. 68, no. 4, pp. 515–535, 2008.

[7] A. H. Baker, T. Gamblin, M. Schulz, and U. M. Yang, “Challenges of scaling algebraic multigrid across modern multicore architectures,” in *IPDPS*, 2011, pp. 275–286.

[8] E. de Araujo Macedo and A. Boukerche, “Hybrid MPI/OpenMP strategy for biological multiple sequence alignment with DIALIGN-TX in heterogeneous multicore clusters,” in *IPDPS Workshops*, 2011, pp. 418–425.

[9] G. Schubert, G. Hager, H. Fehske, and G. Wellein, “Parallel sparse matrix-vector multiplication as a test case for hybrid MPI+OpenMP programming,” *CoRR*, vol. abs/1101.0091, 2011.

[10] A. Lumsdaine, D. Gregor, B. Hendrickson, and J. W. Berry, “Challenges in parallel graph processing,” *Parallel Processing Letters*, vol. 17, no. 1, pp. 5–20, 2007.

[11] A. E. Sarıyüce, E. Saule, and U. V. Çatalyürek, “Improving graph coloring on distributed-memory parallel computers,” in *High Performance Computing (HiPC), 2011 18th International Conference on*, Dec. 2011, pp. 1 –10.

[12] E. Boman, K. Devine, R. Heaphy, B. Hendrickson, V. Leung, L. A. Riesen, C. Vaughan, Ü. Çatalyürek, D. Bozdağ, W. Mitchell, and J. Teresco, *Zoltan 3.0: Parallel Partitioning, Load Balancing, and Data-Management Services; User's Guide*, Sandia National Laboratories, Albuquerque, NM, 2007, tech. Report SAND2007-4748W.

[13] M. R. Garey and D. S. Johnson, *Computers and Intractability*. Freeman, San Francisco, 1979.

[14] D. Zuckerman, “Linear degree extractors and the inapproximability of max clique and chromatic number,” *Theory of Computing*, vol. 3, pp. 103–128, 2007.

[15] D. W. Matula, G. Marble, and J. Isaacson, “Graph coloring algorithms,” *Graph Theory and Computing*, pp. 109–122, 1972.

[16] A. V. Kosowski and K. Manuszewski, “Classical coloring of graphs,” *Graph Colorings*, pp. 1 – 19, 2004.

[17] J. C. Culberson, “Iterated greedy graph coloring and the difficulty landscape,” University of Alberta, Tech. Rep. TR 92-07, Jun. 1992.

[18] M. Luby, “A simple parallel algorithm for the maximal independent set problem,” *SIAM Journal on Computing*, vol. 15, no. 4, pp. 1036–1053, 1986.

[19] M. Jones and P. Plassmann, “A parallel graph coloring heuristic,” *SIAM Journal on Scientific Computing*, vol. 14, no. 3, pp. 654–669, 1993.

[20] R. K. Gjertsen Jr., M. T. Jones, and P. Plassmann, “Parallel heuristics for improved, balanced graph colorings,” *Journal of Parallel and Distributed Computing*, vol. 37, pp. 171–186, 1996.

[21] A. H. Gebremedhin and F. Manne, “Parallel graph coloring algorithms using OpenMP (extended abstract),” in *In First European Workshop on OpenMP*, 1999, pp. 10–18.

[22] A. Gebremedhin and F. Manne, “Scalable parallel graph coloring algorithms,” *Concurrency: Practice and Experience*, vol. 12, pp. 1131–1146, 2000.

[23] M. Patwary, A. Gebremedhin, and A. Pothen, “New multithreaded ordering and coloring algorithms for multicore architectures,” in *Euro-Par 2011 Parallel Processing*, E. Jeannot, R. Namyst, and J. Roman, Eds. Springer Berlin / Heidelberg, 2011, pp. 250–262.

[24] Ü. V. Çatalyürek, J. Feo, A. H. Gebremedhin, M. Halappanavar, and A. Pothen, “Graph coloring algorithms for multicore and massively multithreaded architectures,” *Parallel Computing*, 2012, (to appear).

[25] S. Kang and D. A. Bader, “Large scale complex network analysis using the hybrid combination of a MapReduce cluster and a highly multithreaded system,” in *IPDPS Workshops*, 2010, pp. 1–8.

[26] J. White and J. Dongarra, “Overlapping computation and communication for advection on hybrid parallel computers,” in *Parallel Distributed Processing Symposium (IPDPS), 2011 IEEE International*, may 2011, pp. 59 –67.

[27] M. M. Strout and P. D. Hovland, “Metrics and models for reordering transformations,” in *Proc. of Workshop on Memory System Performance (MSP)*, June 8 2004, pp. 23–34.