Reducing Data Movement on Large Shared Memory Systems by Exploiting Computation Dependencies

Isaac Sánchez Barrera
Barcelona Supercomputing Center (BSC)
Universitat Politècnica de Catalunya (UPC)
isaac.sanchez@bsc.es

Miquel Moretó
Barcelona Supercomputing Center (BSC)
Universitat Politècnica de Catalunya (UPC)
miquel.moreto@bsc.es

Eduard Ayguadé
Barcelona Supercomputing Center (BSC)
Universitat Politècnica de Catalunya (UPC)
eduard.ayguade@bsc.es

Jesús Labarta
Barcelona Supercomputing Center (BSC)
Universitat Politècnica de Catalunya (UPC)
jesus.labarta@bsc.es

Mateo Valero
Barcelona Supercomputing Center (BSC)
Universitat Politècnica de Catalunya (UPC)
mateo.valero@bsc.es

Marc Casas
Barcelona Supercomputing Center (BSC)
Universitat Politècnica de Catalunya (UPC)
marc.casas@bsc.es

ABSTRACT

Shared memory systems are becoming increasingly complex as they typically integrate several storage devices. That brings different access latencies or bandwidth rates depending on the proximity between the cores where memory accesses are issued and the storage devices containing the requested data. In this context, techniques to manage and mitigate non-uniform memory access (NUMA) effects consist in migrating threads, memory pages or both and are generally applied by the system software.

We propose techniques at the runtime system level to further mitigate the impact of NUMA effects on parallel applications' performance. We leverage runtime system metadata expressed in terms of a task dependency graph, where nodes are pieces of serial code and edges are control or data dependencies between them, to efficiently reduce data transfers. Our approach, based on graph partitioning, adds negligible overhead and is able to provide performance improvements up to 1.52× and average improvements of 1.12× with respect to the best state-of-the-art approach when deployed on a 288-core shared-memory system. Our approach reduces the coherence traffic by 2.28× on average with respect to the state-of-the-art.

CCS CONCEPTS

• **Computing methodologies** → *Parallel computing methodologies*; • **Computer systems organization** → Multicore architectures; • **Mathematics of computing** → Graph algorithms;

KEYWORDS

shared memory, scheduling, task-based programming model, NUMA

ICS '18, June 12–15, 2018, Beijing, China

ACM ISBN 978-1-4503-5783-8/18/06.
https://doi.org/10.1145/3205289.3205310

ACM Reference Format:
Isaac Sánchez Barrera, Miquel Moretó, Eduard Ayguadé, Jesús Labarta, Mateo Valero, and Marc Casas. 2018. Reducing Data Movement on Large Shared Memory Systems by Exploiting Computation Dependencies. In *ICS '18: 2018 International Conference on Supercomputing, June 12–15, 2018, Beijing, China.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3205289.3205310

1 INTRODUCTION

Since the end of Dennard scaling and the subsequent stagnation of the CPU clock frequency, computing infrastructures can only increase their peak performance by augmenting the number of computing units. This trend has brought an increase in the count of hardware components in parallel systems as well as their heterogeneity. As such, shared memory systems are experiencing an increase in the number of sockets they integrate. Besides the benefits in terms of a unified flat memory address space and large core counts, integrating many sockets into the same node exacerbates its *non-uniform memory access* (NUMA) effects, which become a serious performance bottleneck if they are not properly handled [10, 40]. For example, synchronization operations or barriers can seriously slow down the whole execution if software components that access locally stored data remain idle while waiting for other components accessing remote data to reach the barrier.

In order to mitigate NUMA effects, techniques for migrating threads, memory pages or both already exist [15, 16, 39]. These techniques aim to move computation near to data or vice versa in order to reduce memory access time. Although they effectively mitigate NUMA effects, they do not exploit any kind of application-specific information to predict accesses to remotely allocated data before a particular software component starts displaying this behavior. As such, already proposed OS-level thread or page migration techniques can only take action when the application is already suffering from remote memory accesses, which ends up bringing suboptimal solutions in most cases. Oppositely, other approaches transfer the NUMA management responsibility to the programmer [1, 41], exploiting information at the application source code

level to carry out NUMA-aware scheduling decisions. However, these approaches require significant code refactoring and programmer effort to be effective.

In this paper, we propose a novel approach to overcome the limitations of already existing methods. Our techniques automatically mitigate NUMA effects on nodes with multiple NUMA-domains leveraging runtime system metadata, with minimal programmer intervention and application source code changes. In a task-based data-flow programming model, the parallel execution is conceived as a directed acyclic graph where nodes are pieces of sequential code and edges are control and data dependencies between them. Such abstraction is supported in the latest releases of OpenMP [29], the most used shared memory programming environment. In this context, the sequential pieces of code are called tasks and there is runtime system support to schedule them without breaking the dependencies the programmer has specified in the code via #pragma annotations. While the application is running, the runtime system automatically builds a *task dependency graph* (TDG) to orchestrate the whole parallel execution. A key aspect of this data-flow model is the explicit knowledge the runtime system has about the ranges of memory addresses that are going to be accessed by tasks before they start running, enabling improvements in terms of data prefetching [30] or cache coherence protocol optimizations [25].

Our approach considers the information contained in this TDG data structure to drive two techniques applied at the runtime system level; they apply advanced graph partitioning algorithms to break down the TDG into several pieces or parts. These partitions aim at minimizing data transfers across the parallel system. All things considered, the contributions of this paper are the following:

- Two schemes that dynamically perform graph partitioning over the TDG: The *runtime-informed partitioning with dependency easy placement* (RIP-DEP) and the *runtime-informed partitioning with moving window* (RIP-MW). Both approaches partition an initial subgraph containing the firstly created tasks but propagate this partition in different ways: RIP-DEP exploits information regarding the allocation of tasks' input data while RIP-MW repartitions the initial TDG subgraph as new tasks are added.
- A complete performance evaluation of the proposed techniques against 3 other methods: an expert programmer-driven policy, a locality-unaware *distributed first-in-first-out* (DFIFO) approach and an implementation of a state-of-the-art technique [17, 18, 42], *dependency easy placement* (DEP), that automatically schedules tasks depending on where their input and output data are allocated. Our evaluations consider 8 different OpenMP codes and 2 different parallel systems with up to 288 cores. Our proposals incur minimal runtime system overhead while keeping the parallel workloads well balanced. Our experiments show how RIP-DEP achieves speedups of up to 1.52× and average improvements of 1.12× on 288 cores with respect to DEP, the best state-of-the-art approach.
- An exhaustive evaluation of the coherence traffic triggered by all the considered approaches. The evaluation includes categories like control traffic, which is composed of messages carrying coherence protocol signaling activities without a data payload, and data traffic, which is composed of messages carrying a single cache line payload. Our coherence traffic evaluation explains the performance benefits of our techniques as it demonstrates that RIP-DEP achieves outstanding coherence traffic reductions of 172.2× and 2.28× on average compared to DFIFO and DEP, respectively.

This paper demonstrates that, although simple NUMA-aware heuristics (e.g., DEP) provide reasonably good performance in workloads with simple TDGs, more advanced techniques based on graph partitioning algorithms (e.g., RIP-DEP) are required to avoid coherence traffic to become a significant performance bottleneck.

The rest of this paper is organized as follows: Section 2 introduces the possibilities offered by task-based programming models in shared memory systems with NUMA effects. Section 3 gives some insight on the graph partitioning problem and approaches to tackle it. Following this, Section 4 explains how graph partitioning techniques can be applied at execution time to mitigate NUMA effects. Next, Section 5 describes the experimental environment and Section 6 evaluates the proposed scheduling mechanisms in terms of decrease of execution time and coherence traffic, including a detailed analysis of the load balance and overheads of the proposals. Section 7 describes the related work and, finally, Section 8 concludes this work.

2 OPPORTUNITIES OF TASK-BASED PARALLELISM

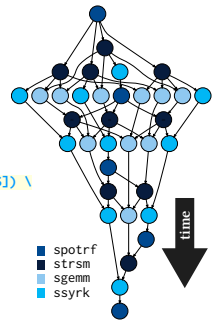
2.1 Task-Based Programming Models

The most common way to program shared memory nodes are thread-based programming models like OpenMP [29]. Recent versions of the OpenMP standard, starting from 4.0, have support for tasking and dependencies: the application source code is split into several pieces called *tasks*, which have their data or control dependencies explicitly indicated at the `#pragma` annotations. The programmer can indicate the data used by means of the `depend` clause, whether it is used as input or output and, optionally, its size. An example of this for the Cholesky matrix decomposition algorithm is shown in Listing 1. The sequential code is split into four task types: `spotrf` to calculate the Cholesky decomposition of the diagonal blocks, `strsm` to solve the linear systems that define the below-the-diagonal blocks, and `sgemm` and `ssyrk` to do matrix multiply and rank S operations to update the rest of the matrix.

A directed acyclic graph where nodes represent tasks and edges express dependencies between them is built and maintained by the runtime system to orchestrate the parallel execution. This graph is commonly referred as the application's task dependency graph, with an example accompanying Listing 1. The runtime system is in charge of managing the parallel execution, releasing the programmer from the burden of explicitly expressing task synchronization or scheduling at the application source code level.

The typical behavior of a thread in a task-based runtime system consists in requesting tasks to the scheduler, executing them and notifying about task completions to enable the wake-up and execution of dependent computations. Once a thread sends a request to the runtime system, a task is scheduled based on a certain policy.

```
void cholesky(int T, float *A[T][T], int S) {
  // each A[i][j] has size S*S
  for (int k = 0; k < T; ++k) {
#pragma omp task depend(inout: A[k][k][:S*S])
    spotrf(A[k][k]);
    for (int i = k + 1; i < T; ++i) {
#pragma omp task depend(in: A[k][k][:S*S]) \
                 depend(inout: A[k][i][:S*S])
      strsm (A[k][k], A[k][i]);
    }
    for (int i = k + 1; i < T; ++i) {
      for (int j = k + 1; j < i; ++j) {
#pragma omp task depend(in: A[k][i][:S*S], A[k][j][:S*S]) \
                 depend(inout: A[j][i][:S*S])
        sgemm(A[k][i], A[k][j], A[j][i]);
      }
#pragma omp task depend(in: A[k][i][:S*S]) \
                 depend(inout: A[i][i][:S*S])
      ssyrk(A[k][i], A[i][i]);
    }
  }
}
```



Listing 1: Task-based Cholesky decomposition written using OpenMP 4 and its corresponding TDG when T = 5.

2.2 Task Scheduling in NUMA Systems

Task programming models are especially well suited for large shared memory systems with NUMA effects. The specification of tasks' input and output dependencies provides the runtime system with information about what data is going to be accessed. Moreover, if the runtime system is aware of where data resides within the NUMA regions of the parallel architecture, a given task can be scheduled in a thread local to the NUMA region where its required data resides. This avoids the cost of remote memory accesses and bandwidth waste. This NUMA-aware scheduling also provides a higher probability for a task to hit its data in the cache of the processor if previous tasks using that data also ran in the same socket. Therefore, memory accesses to inputs and outputs will frequently access *close* memories during the execution of tasks, exploiting applications' data locality as a result.

This kind of NUMA-aware scheduling policies exploiting the task-based model has already been applied to parallel systems [1, 17, 18, 31]. To improve data locality, some authors propose to enrich the API of the runtime system so that the programmer can manage data placement and exploit data locality by specifying the NUMA region where tasks should be executed [1, 17]. These approaches also implement a distance-aware work-stealing method that steals tasks from the closest NUMA regions, which reduces load imbalance. However, they are not automatic, increasing the programmability burden in parallel systems. An automatic technique to mitigate NUMA effects on shared memory systems has also been proposed [18]. This technique is the most recent state-of-the art approach and it is further discussed in Section 4.1 and considered in the experiments described in Section 6. As these experiments demonstrate, this paper improves the state-of-the-art by leveraging the information contained in the TDG and using graph partitioning techniques to automatically mitigate NUMA effects.

3 GRAPH PARTITIONING

Throughout the literature [5, Section 2], the graph partitioning problem is defined as in Problem 1.

Problem 1 (Graph partitioning). Given a positive integer k and an undirected graph $G = (V, E)$ with positive edge weights $\omega\colon E \longrightarrow \mathbb{R}^+$, find a partition Π of the set of vertices V composed of k parts V_i with the following properties:

(1) $V_1 \cup \cdots \cup V_k = V$ (the parts cover all the vertices),
(2) $V_i \cap V_j = \emptyset$ if $i \neq j$ (the parts are disjoint).

In general, we want partitions that are balanced, that is $|V_i| \leq (1+\varepsilon)\lceil |V|/k \rceil$ for some $\varepsilon \geq 0$, and such that some metric is minimal. If we define the mapping $\varphi\colon V \longrightarrow 1, \ldots, k$ that assigns every vertex to the partition where it belongs, or $\varphi(v) = i$ if $v \in V_i$ in Π, we want to minimize the function

$$\sum_{\substack{uv \in E \\ \varphi(u) \neq \varphi(v)}} \omega(uv). \tag{1}$$

This function (1) is known as the *edge cut* of the solution and corresponds to the total weight of the edges connecting pairs of vertices from two different parts in Π.

Under these constraints, the problem is NP-hard, but there are known algorithms and heuristics for approximating it [5]. Some of the commonly used libraries in the HPC scenario are Metis [21, 22], Scotch [32, 33], Zoltan [4] and Metapart [36]. These libraries aim at reducing data transfers across parallel distributed memory systems by statically splitting input data like meshes or matrices.

3.1 Graph Partitioning Algorithms

In this work we use standard graph partitioning tools for undirected graphs to partition the TDG of applications, which are directed acyclic graphs. In particular, we partition the TDG in k parts, where k equals the number of NUMA regions (or sockets), and use the amount of transferred data between parts as edge cut function. While there is a wide range of graph partitioning algorithms (exact, recursive, greedy, local search...), this paper makes use of a multilevel approach combined with Dual recursive bipartitioning, Fiduccia-Mattheyses and Graph growing algorithms, which are summarized below. Complete details of these and other approaches are described in the literature [5].

Dual recursive bipartition is one of the most basic and used methods. It is a recursive divide-and-conquer algorithm that consists in doing a 2-partition of the set of parts, and a 2-partition of the set of vertices and map the last two to the pair of sets of parts. The mapping is done recursively until what is assigned is a set of tasks to a single part. The bipartitions are done using some heuristics that use the information from the edge weights to make good decisions.

Rather than a partitioning/mapping algorithm, *Multilevel mapping* is a scheme to do the partition in an easier way or with higher quality. It coarsens the input graph (makes it rougher, joining vertices), then applies the partitioning algorithm to the coarsened graph, projects back the partition to the original graph and refines it. This is well shown in Scotch User's Guide [34, Figure 3].

Fiduccia-Mattheyses is a local-search algorithm extended to not stall in a local minimum, and it is an evolution of the Kernighan-Lin method (another algorithm using local search). Starting with a given partition, it tries to improve it by moving vertices from one part to another or by swapping vertices in different parts. The selection is done with the vertices that make the edge cut decrease the most.

Graph growing algorithms are based on a breadth-first search that starts from some seed vertices and grows the parts greedily. The parts are grown in an order such that the next part to get a vertex is always the smallest one. Local search is then applied to balance the load of the parts, and new seed nodes are selected for the next step.

The ways in which we use these methods are detailed in Section 4.3, where we introduce and explain our proposals for reducing NUMA effects based on graph partitioning.

4 EXPLOITING THE TASK DEPENDENCY GRAPH TO MITIGATE NUMA EFFECTS

In order to automatically orchestrate a parallel execution while optimally mitigating NUMA effects on large shared memory nodes, we exploit the information contained in the TDG of the application. To do so, we consider either techniques that analyze the TDG by means of a simple heuristic or techniques based on advanced graph partitioning algorithms.

In order to be able to apply the proposed techniques, throughout the rest of the work, we assume a first-touch memory placement policy and page-aligned memory blocks. This means that a data page is physically allocated in memory the first time it is used, and the allocation is done in the NUMA domain of the core making the access, which is the default behavior in a Linux system.

4.1 Dependency Easy Placement (DEP)

By *dependency easy placement* (DEP), we refer to the approach proposed by Drebes et al. [17, 18] in terms of a dynamic task and data placement policy based on two concepts: i) *deferred allocation*, which implies that the memory to store task output data is not allocated until the task placement is known, and ii) *enhanced workpushing*, which means that tasks are scheduled to the NUMA region where most of their data dependencies are allocated. A similar method is proposed by Virouleau et al. [42]. In our context, the enhanced workpushing mechanism is implemented by means of a table to map the dependencies to sockets kept by the runtime system. The first address of a data dependency is used as its identifier; this way, we avoid invoking high cost system calls to figure out the sockets where the data is allocated. Also, data dependencies are allocated in the socket where the first task accessing them is executed, which is equivalent to the deferred allocation mechanism.

At the time of scheduling a task, the runtime explores its dependencies and weights the sockets using the size of the allocated dependencies (input and output), considering a virtual extra socket for unallocated data (also weighted using the size). Then, the task is scheduled to the socket with the highest weight. If the highest weight is for the virtual socket (unallocated data), the final socket is chosen via a discrete uniform distribution considering all the sockets available to the runtime system. In case of a tie, the socket is chosen via a discrete uniform distribution among the tied ones. Observe that DEP can also be seen as a *propagation* technique: once the data is placed physically in memory (using some kind of heuristic), tasks can be scheduled in cores that are near the data they use to be able to consume it faster.

This paper demonstrates in Section 6 how techniques based on graph partitioning achieve better performance and dramatically reduce the amount of data transfers carried out by techniques like the one proposed by Drebes et al. [18].

4.2 Considerations about Applying Graph Partitioning on Applications' TDGs

To exploit the structure of the application we use graph partitioning algorithms. Considering the whole TDG is not an option because partitioning schemes target undirected graphs, which implies that they typically split TDGs with deep task paths in a way that all potentially concurrent tasks are assigned to the same part. Intuitively, when the graph is wide rather than tall, the partitioning algorithm will decide that it is better to cut the edges (i.e., partition the graph) vertically because there will be fewer edges than horizontally. Using *hypergraph* partitioning software packages does not help in our context either since they use algorithms with high computational cost [11, 12, 23, 24] that require large distributed memory systems to run [4]. Also, since in practice the dependency graph is built simultaneously with the execution, the complete TDG is never available at runtime. In this context, the natural way to proceed is operating over small task subgraphs instead of over the whole TDG, that is, partitioning subgraphs and then extrapolating this partition to the upcoming tasks following a certain policy.

4.3 Runtime Informed Partitioning (RIP)

Under the *runtime informed partitioning* (RIP) family of policies, task scheduling decisions are based on graph partitioning techniques. The TDG is built at run time by leveraging information in terms of task dependencies. The graph is updated every time new tasks are instantiated, and partitioned once the execution goes through a barrier point or a limit in terms of the total number of tasks contained in the graph —called the *window size* limit— is reached. The partitioning algorithm uses the TDG as input, weights its edges depending on the amount of bytes they represent and assigns tasks to a particular part (corresponding with a specific socket) taking into account the machine NUMA distances.

In order to partition the initial subgraph, we use the dual recursive bipartition, the multilevel mapping and the Fiduccia-Mattheyses methods described in Section 3.1, and available in the Scotch [33] graph partitioning library, version 6.0.4. The graph growing algorithm, also described in Section 3.1, is available within the Metapart framework [36]. We represent the target architecture as a complete graph with as many vertices as NUMA domains, and with edge distances proportional to the NUMA distances measured as explained at the beginning of Section 5. For doing the partitions, the TDG is transformed to an undirected graph for Scotch. Once the complete graph that defines the target architecture is set, the initial partition is obtained by calling `SCOTCH_graphMap` with the default settings. Such default settings make use of a multilevel approach with dual recursive bipartitioning combined with the Fiduccia-Mattheyses local search algorithm.

We partition the initial subgraph given by the first *window size* tasks. The partitioning is done asynchronously while the runtime system creates new tasks. Once the initial subgraph has been partitioned, we consider two possible options to proceed: the first one consists in propagating the partition across the whole execution following a memory-allocation-aware policy, which corresponds to the RIP-DEP technique. The other alternative is to keep partitioning the different subgraphs the runtime system generates as the execution advances, which corresponds to the RIP-MW approach.

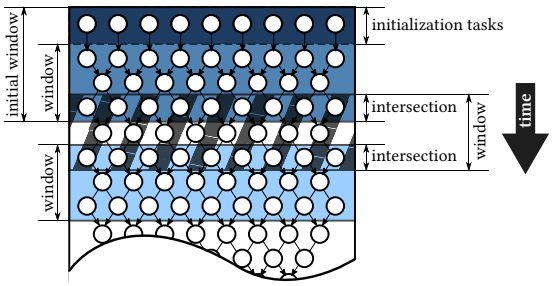
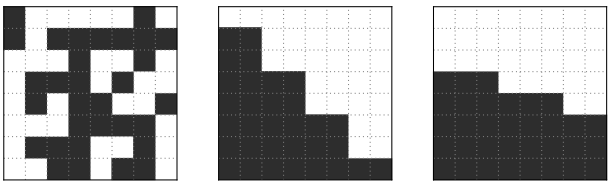


Figure 1: Diagram showing how RIP-MW works over time. The most relevant parameters for RIP-MW are represented.



(a) uniform distrib. **(b) optimal** **(c) RIP**

Figure 2: Task and data allocations into two sockets (dark and light) on the first iteration of Gauss-Seidel (8 × 8 grid).

The first technique aims at reducing the overhead due to graph partitioning as much as possible while RIP-MW aims at dynamically adapting the TDG partition during the parallel run. The overhead of partitioning the initial subgraph is small enough to be overcome by the benefits of graph partitioning (see Section 6 and Section 6.4 in particular, which provide performance results accounting for task creation and scheduling, as well as partitioning the TDG).

When tasks are ready to run (i.e., all their input dependencies are solved) but the partition is not done yet, they are stored in a temporary queue. Tasks are transferred to the ready queue as soon as they have been assigned to a socket. The temporary queue is not used often since, in general, the partition is obtained much before the tasks are ready.

4.3.1 RIP with Dependency Easy Placement (RIP-DEP). This technique based in graph partitioning consists in propagating the partition obtained from the initial subgraph by taking into account where the tasks data dependencies reside. More specifically, this approach uses DEP to propagate the partition to the rest of the graph, already described in Section 4.1. The main difference between DEP and RIP-DEP is the way of doing the initial partition: while DEP does the allocation using a uniform distribution, RIP-DEP partitions the TDG.

4.3.2 RIP with Moving Window (RIP-MW). In this case, the graph partitioning is performed many times throughout the execution of the program. Once the subgraph contains a particular amount of tasks —the *window size*—, or a barrier point is reached, the partitioning algorithm is run. Once a partition is obtained, the oldest tasks are flushed from the subgraph and a new one is built. As it is shown in Section 6.4, the overhead of graph partitioning is minimal (1.18 % on average). Moreover, the partitions are scheduled asynchronously as tasks, effectively overlapping the execution of the user-level tasks with the partitioning of new subgraphs. The user can set up the window size, an initial extra amount of tasks for the first window and the size of the intersection between two consecutive windows. This intersection is used by the partitioner to reduce the algorithmic complexity and preserve data locality from previous partitions.

Once the initial subgraph is partitioned in the way we describe above, RIP-MW keeps partitioning task subgraphs by calling the method `partitionGraphSCOTCHK` from Metapart, which uses the Graph growing algorithm with support for fixed vertices [36] in a multilevel framework combined with Fiduccia-Mattheyses. The reason for using Graph growing is that Dual recursive bipartitioning methods can perform badly under fixed vertex constraints [36, Fig. 1]. Using fixed vertices is required to exploit information from previous partitions and avoid as much as possible the mapping of tasks to NUMA domains distant from the data they consume.

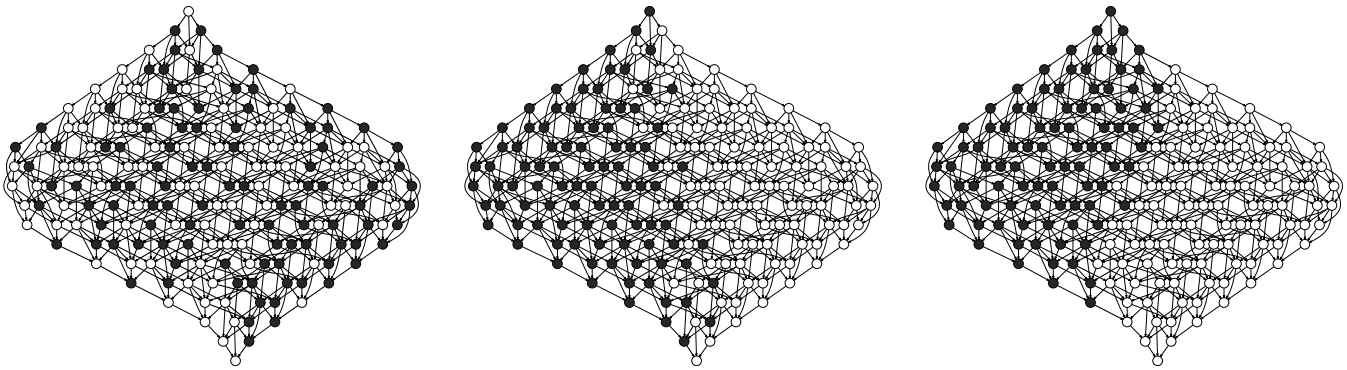
Figure 1 shows the way RIP-MW works. First, an initial subgraph composed of the initialization tasks plus the first window is partitioned. After this, a new subgraph is built, including the tasks in the intersection plus the new ones, until the window size is reached. Then, a second partition with fixed vertex constraints is carried out. The following subgraphs are built and partitioned in the same way.

4.4 Benefits of Graph Partitioning

While simple heuristics based on data locality, like DEP, are able to produce good partitions in some scenarios, in other cases they fail to optimally partition the graph. This is especially relevant as the number of NUMA regions in the system increases. At the same time, automatic mechanisms based on graph partitioning can make the codes more architecture-agnostic and easier to program than manual assignment of the tasks to the sockets.

As an example, we consider the stationary heat diffusion problem using the iterative Gauss-Seidel method with a 4-element stencil (top, bottom, left, right) in an 8×8 regular grid, which corresponds to the Gauss-Seidel application later described in Section 5.2. Each task operates over one cell of the grid. In each iteration, computations over every cell depend on the data of the four neighboring cells, the algorithm execution follows a wavefront scheme in the direction of the main diagonal, and tasks in the same anti-diagonal are independent between them. For this reason, when targeting two sockets, the optimal partition consists in dividing the domain along the main diagonal. As a result, at each instant, half of the anti-diagonal can be executed in a different socket. Figure 2 shows the allocation of the data and the corresponding tasks for Gauss-Seidel for a discrete uniform placement (e.g., DEP), the explained optimal partition and using a RIP method (equivalent for all RIP proposals in the case of the first iteration).

Figure 3 shows the same partitions expressed at the TDG level on three iterations of Gauss-Seidel. Clearly, data transfers among tasks assigned to different sockets are minimized in the expert programmer-given partition and the one obtained via graph partitioning (RIP-DEP): the graphs are cut almost *vertically*, increasing parallelism while grouping neighboring tasks in the same socket. In contrast, the DEP approach produces a sub-optimal partition, with more edges connecting different parts. The implications of these results in terms of the total performance are detailed in Section 6.



(a) uniform distribution + DEP **(b) programmer-given partition (optimal)** **(c) RIP-DEP**

Figure 3: Task dependency graph corresponding to three iterations of the Gauss-Seidel code comparing a uniform distribution placement with locality awareness (DEP) to a programmer-given partition and the RIP-DEP technique in a two-socket system.

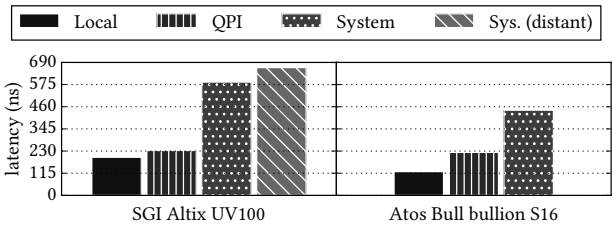


Figure 4: Measured memory latencies in milliseconds as we increase the working set size with LMbench `lat_mem_rd`.

4.5 Assumptions of the Proposals

Our proposals make some general assumptions. First of all, the system needs to have a first-touch policy with local allocation, the default in the Linux kernel, and data blocks have to be initialized in page-aligned addresses using tasks in order to take full benefit from this first-touch policy. In the case of the RIP-MW techniques, barriers in the middle of a window of tasks may reduce the quality of the subgraph to partition since no new tasks are created after a barrier until all previous tasks have finished. As such, this technique benefits from a reasonably high ratio of tasks per barrier point. The RIP-DEP and RIP-MW methods need the user to set the window size as a parameter for the runtime. This is simple to achieve as intuition and experiments show that it is enough to include the initialization tasks and the first couple of computation phases of the application (iterations in the case of iterative algorithms). An alternative solution is to apply existing techniques of automatic detection of the phases [9] and use this information to decide at execution time what is a correct window size. Another possible approach is editing the application source code by making a call to the runtime system API indicating that the partition must be done at the specific point of the call.

5 EXPERIMENTAL ENVIRONMENT

In all cases, we use the OpenMP programming model with a customized Nanos++ v0.10 runtime system and the companion Mercurium 2.0.0 (rev. c5a91d5) compiler [3, 38]. In the case of programs that need LAPACK, we use the open-source implementation from OpenBLAS 0.2.19 [28, 43] compiled for each architecture. Threading of the library is disabled so as not to interfere with OpenMP.

5.1 Considered Platforms

We evaluate the proposed techniques in two different platforms. The first machine is an SGI Altix UltraViolet 100 with 3 IRU (*internal rack units*) interconnected with NUMAlink at 15 GB/s. Each IRU contains two IP93 blades with two 8-core Intel Xeon E7-8837 CPU (Westmere-EX) at 2.66 GHz and 24 MB of shared last-level cache, and 16 DIMM of 16 GB DDR3 RAM. Sockets in the same blade communicate via Intel QPI (*Quick Path Interconnect*). The system runs SUSE Linux Enterprise Server 11 with Linux 2.6.32 kernel. We use GCC 5.1.0 as the backend compiler for Mercurium.

We use `lat_mem_rd` from LMbench [26] to measure the true memory latencies, shown in Figure 4, and pass that information to the partitioning library. In the Altix, accesses within the same blade have an increased latency of 17 % compared to local memory, while there is a significant latency penalty of 200 % to access data in other IRUs, and close to 240 % in the most distant sockets.

The other machine is an Atos Bull bullion S16 with 8 modules, each one with two 18-core Intel Xeon E7-8890 v3 sockets (Haswell) at 2.50 GHz and 45 MB of shared last-level cache. Each socket has 512 GB of local RAM and is connected via Intel QPI to the other socket in the module; modules are interconnected using the Bull Connecting Box and communicate using the BCS2 (*Bull Coherence Switch 2*) [2]. The system runs Red Hat Enterprise Linux 6.5 with Linux 2.6.32 kernel. We use GCC 4.8.2 as the backend for Mercurium. In this case, the access latency via QPI has an extra penalty of 79 % and of 260 % for remote accesses via the BCS, as shown in Figure 4.

5.2 Tested Applications

This section describes the parallel codes considered in this paper and also the source-code-level annotations that drive the Socket-Aware (SA) scheduler, described in Section 6. We test our proposals by considering this set of benchmarks, which is representative of typical parallel workloads.

Conjugate gradient (CG) is an iterative method for solving linear symmetric positive-definite systems of equations. It computes the solution by building a basis of orthogonal vectors each iteration. We use a sparse matrix version with the task decomposition described by Jaulmes et al. [19]. The manual scheduling assigns tasks to sockets in a round-robin fashion. The window size corresponds to

all tasks belonging to a single iteration. When applying the RIP-MW technique, the intersection is equivalent to half iteration.

Gauss-Seidel is an algorithm solving the stationary heat diffusion problem using the iterative Gauss-Seidel method with a 4-element stencil (top, bottom, left, right). The implementation is based on a task decomposition given by tiles with the tile contents contiguous in memory (instead of the rows) and halos between the tiles of the matrix to communicate the borders. The graph follows a wavefront shape, as shown in Figure 3. The source-code-level annotations divide the columns contiguously into as many groups as NUMA domains. The window size covers the tasks of three iterations, with an intersection of a whole iteration for RIP-MW.

The *Integral histogram* computes a cumulative histogram for each pixel of an image, using a cross-weave scan as described by Porikli [35]. In our case, the calculation of the histograms of different images are overlapped to increase parallelism. The vertical and horizontal halos used for the reduction of the histograms are allocated in a round-robin fashion, in both dimensions. The image data and scan tasks are assigned to a socket in a round-robin manner using the column identifier so that they match with the corresponding vertical halos. For the schedulers based on graph partitioning, the window size corresponds to the tasks of two iterations, with an intersection of a whole iteration.

Jacobi solves the stationary heat diffusion problem using the iterative Jacobi method with an implementation derived from the Charm++ project [13, 20]. This implementation uses a 5-element stencil (top, bottom, left, right, center) and a task decomposition given by blocks of rows. The source code level annotations for assigning these blocks to a socket follow a round-robin approach. The double-buffer nature of Jacobi gives an embarrassingly parallel algorithm inside every iteration with a very symmetric TDG, hence it becomes simple to partition in contrast to the Gauss-Seidel code that solves the same problem. The window size includes the tasks of two iterations, with an intersection of a whole iteration.

NStream is a synthetic benchmark to measure memory bandwidth based on STREAM [27]. This task-based parallel implementation works with N independent arrays (a multiple of the number of threads, usually). Its task graph is made of N isomorphic connected components, so partitioning it should be as easy as assigning every component to one NUMA domain. The user-level annotations for the NUMA-aware scheduler assign each array and the related tasks to a socket following a round-robin approach, effectively assigning every array to a single NUMA node. The window size is $5N$ and the intersection is $2N$.

The *QR factorization* of a matrix A is a product $A = Q\,R$ where Q is orthogonal and R is upper triangular. We use a task-based implementation of the tiled algorithm, using LAPACK as described by Buttari et al. [6], which saves the R matrix and the Householder reflectors (to compute Q) in-place. The manual scheduling assigns the blocks in a round-robin fashion using the row identifier, while the subsequent tasks are assigned where most blocks reside (using the row identifier). The window size is equivalent to the total number of blocks the matrix is broken into and the intersection considered by RIP-MW corresponds to two rows of blocks.

Red-Black is the third algorithm for solving the stationary heat diffusion problem. The data decomposition is exactly the same as for Gauss-Seidel, but the task graph is more similar to Jacobi; the red sub-iterations are fully parallel (by tiles) and so are the black sub-iterations. The source-code-level annotations defining the manual scheduling divide the columns contiguously into as many groups as NUMA domains, like in Gauss-Seidel. Similarly, the window size is for the tasks of three iterations, with an intersection of a whole iteration for RIP-MW.

Symmetric matrix inversion (SMI) is used to compute the inverse of a symmetric matrix in a fast way by using a Cholesky factorization. We use the tiled task decomposition of the dense linear algebra version and the manual NUMA-aware scheduling as described by al Omairy et al. [1], using LAPACK. The window size corresponds to the tasks of the lower triangle of the matrix (it is symmetric), with an intersection of half a triangle.

6 EVALUATION

In this section we evaluate the performance of the proposed mechanisms considering the eight applications and two platforms described in Section 5. Our evaluation considers five different scheduling techniques:

- *Distributed First-In First-Out* (DFIFO), unaware of data locality. In this technique, each thread has its own ready queue and tasks are assigned to threads in a round-robin manner. When the queue of a thread is empty, it applies a work stealing mechanism to get tasks from other threads.
- *Socket Aware* (SA) scheduler, which is driven by a partition expressed in terms of annotations at the source code level done by an expert programmer. SA makes use of an API call that specifies the precise socket where tasks should run. The specific annotations of each benchmark are explained in Section 5.2.
- The *DEP* approach, which is described in Section 4 and represents the current state-of-the-art. All results reported in this section are normalized against DEP.
- Our two proposals based on graph partitioning algorithms: *RIP-DEP* and *RIP-MW*.

For every application, platform and method we repeat each experiment five times. In all speedup plots shown, values are averaged among the different repetitions and normalized to DEP (horizontal line at 1.0). Bar height represents the mean value, a horizontal thick line is the median, and error bars show the standard deviation. For each system configuration we include a plot of the geometric mean computed over the arithmetic means of the eight benchmarks. Our experiments are run with the following four configurations: On 24 cores of the UV100, using 8 cores per socket and 3 sockets (2 in the same blade, 1 in a different blade); on 32 cores of the bullion S16, using 8 cores per socket and 4 sockets (1 per module); on 32 cores of the bullion S16, using 4 cores per socket and 8 sockets (1 per module), and, finally, on all 288 cores of the bullion S16, using 18 cores per socket and 16 sockets (2 per module).

6.1 SGI Altix UV100

For the SGI Altix UV100 machine, we have done experiments using 3 sockets and 24 cores in total. All parallel runs use two sockets in the same blade (which communicate via QPI) and a third one from a different blade, although not a distant one. Results are shown in Figure 5. On average, RIP-DEP achieves speedups of 1.03× over the

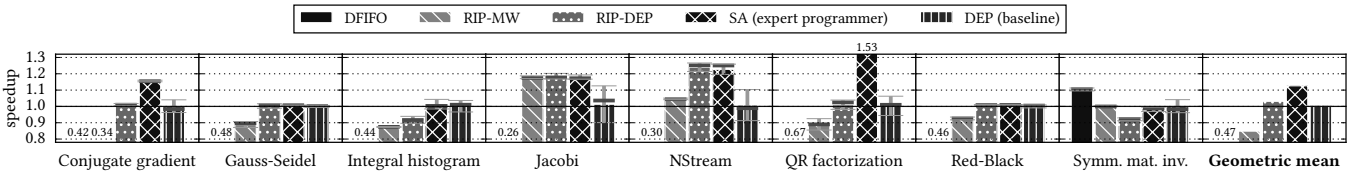
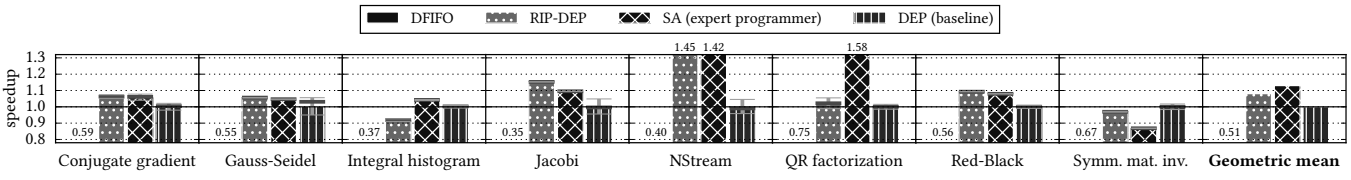
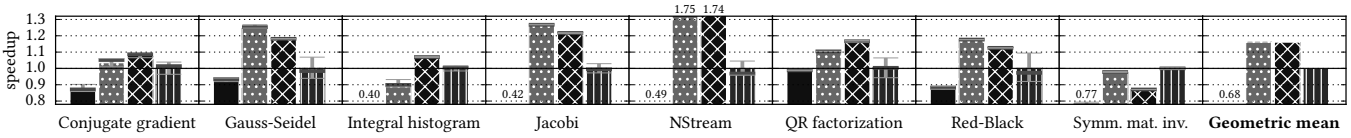


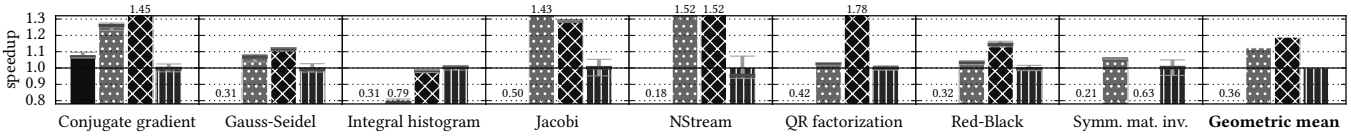
Figure 5: Speedup results in the SGI Altix UV100 using 3 sockets, 24 cores. DFIFO is locality-unaware, SA is manual, the rest are automatic methods.



(a) Speedup results in the bullion S16 using 4 sockets, 32 cores.



(b) Speedup results in the bullion S16 using 8 sockets, 32 cores.



(c) Speedup results in the bullion S16 using all 16 sockets, 288 cores.

Figure 6: Speedup results in the Atos Bull bullion S16. DFIFO is locality-unaware, SA is manual, the rest are automatic methods.

DEP baseline, RIP-MW only gets up to 0.84×, while the scheduling policies driven by an expert programmer (SA) provide a 1.13× speedup. On the other hand, the locality-unaware scheduler DFIFO has a general underperformance (0.45×) except in the Symmetric matrix inversion (1.10×). The relatively small benefits shown by RIP-DEP with respect to the DEP baseline are explained by the reduced number of NUMA domains, just three, considered in the experiments run in the UV100 machine.

However, even though the average benefits of RIP-DEP over DEP in the UV100 machine are small, there are some specific cases for which they are significant: for instance, in Jacobi the automatic partition using RIP-DEP achieves a 1.19× speedup over DEP and RIP-MW goes up to 1.18×. The benefit with RIP-DEP is more noticeable in larger machines such as the bullion S16, presented in Section 6.2, as the number of NUMA regions increases.

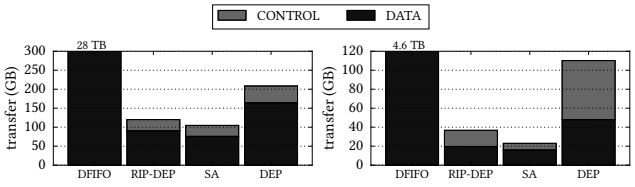
6.2 Atos Bull bullion S16

In the case of the Atos Bull bullion S16 machine we provide experiments considering 4 sockets (32 cores), 8 sockets (32 cores) and the full system (16 sockets and 288 cores). Overall, the results in the Atos Bull bullion S16 system show how RIP-DEP provides average performance improvements of 1.08× on 4 sockets, 1.16× on 8 sockets and 1.12× on 16 sockets with respect to the state-of-the-art. RIP-MW achieves very similar improvements on the experiments involving 4 and 8 sockets (i.e. 32 cores), as Section 6.1 shows. In the case of 288 cores, RIP-MW provides worse performance than RIP-DEP since the frequent graph partitions become a significant performance bottleneck. For readibility purposes, the RIP-MW technique does not appear on the experiments regarding the Atos Bull bullion S16 system.

6.2.1 Using four sockets. Results using four sockets in the bullion S16 system are shown in Figure 6a. Under this configuration, the average speedup obtained using RIP-DEP is of 1.08× with respect to DEP. As in the Altix machine presented in Section 6.1, the execution times of an expert programmer-driven schedule (SA) attain a 1.13× speedup when compared with DEP. The naive DFIFO gets 0.51× performance degradation with respect to the state-of-the-art DEP approach.

RIP-DEP behaves better than the DEP baseline for the Conjugate gradient and Gauss-Seidel applications (1.05× improvement for both application) and much better for the Red-Black, Jacobi and NStream parallel codes (1.09×, 1.15× and 1.45×, respectively). These results are explained by the good structure of the task graphs of these codes, which benefit from partitioning the initial subgraph and, at the same time, the iterative access pattern to the blocks of data allows for a good locality-aware propagation. In particular, Jacobi shows very good performance under this system configuration for RIP-DEP, with a higher performance than the programmer-driven partition (SA, achieving a speedup of 1.09×).

6.2.2 Using eight sockets. Results with eight sockets, in Figure 6b, display larger speedups of the RIP-DEP approach with respect to DEP than previous scenarios. Here, RIP-DEP attains an average speedup of 1.16× over DEP, which is matched by the expert programmer-driven partition.



(a) Gauss-Seidel **(b) Integral histogram**

Figure 7: Coherence traffic to and from the BCS for selected applications using 32 cores in 8 sockets in the bullion S16.

Under this setting is where RIP-DEP obtains the highest benefit for Gauss-Seidel and Red-Black (1.26× and 1.18×, respectively). For the Integral histogram, the executions are somewhat slower using RIP-DEP than DEP (0.91×). The benchmark operates on a 2D domain and accumulates results in both dimensions, which creates horizontal and vertical data dependencies across tasks, which forces the partitioning algorithm to split the TDG in a way that the execution of some parallel tasks is serialized. However, as Section 6.3 shows, combining both graph partitioning and a locality-aware propagation (RIP-DEP) significantly reduces data movement with respect to DEP in the case of the Integral histogram application.

6.2.3 Full system. Results when using all the 288 cores of the bullion S16 system are shown in Figure 6c. The input set of some applications is increased to achieve good scalability on 288 cores (e.g., CG). When running on all cores of the bullion S16 system, the RIP-DEP approach achieves a remarkable average speedup of 1.12× with respect to the state-of-the-art DEP technique. For Jacobi, RIP-DEP achieves an outstanding 1.43× speedup over DEP, only surpassed by NStream, which achieves a speedup of 1.52× when using RIP-DEP due to its simple graph.

When using all cores of the bullion S16 system, the expert programmer-driven partition (SA) obtains an average speedup of 1.19× with respect to DEP. While in some cases (e.g., QR factorization) the non-automatic expert-driven SA partition achieves better performance than the automatic RIP-DEP method, in the case of the Symmetric matrix inversion code the policy driven by the expert programmer performs poorly. Symmetric matrix inversion's TDG is so complex that a proper partition needs to know where data are allocated, which is impossible to be statically determined unless very simple memory allocation policies are applied, which do not provide performance benefits either. For all settings, the SA technique applied to the Symmetric matrix inversion code performs below the DEP baseline, which shows the need for dynamic and automatic methodologies in the case of very complex TDGs.

6.3 Reduction of Coherence Traffic within the bullion S16 Machine

This section provides an evaluation of the coherence traffic triggered within the bullion S16 system by all the 5 approaches considered in this paper. This evaluation demonstrates how the RIP-DEP method we propose achieves remarkable reductions of coherence traffic. The bullion platform uses a sophisticated ccNUMA architecture composed of sets of 2 sockets grouped into entities called modules. The Bull Coherence Switch (BCS) [2], a proprietary ASIC, manages the inter-module interface and enables scaling up to a maximum of 8 modules (i.e., 16 sockets of Intel Xeon CPUs) in a single shared memory system. We use the measurement capabilities of the BCS to provide a precise analysis of the coherence traffic [7, 8]. We divide the coherence traffic in the system into two categories: *data messages*, which carry a single cache line payload, and *control messages*, which carry coherence protocol signaling activities without a data payload.

Figure 7 shows the differences in data transfer to and from the BCS for Gauss-Seidel and Integral histogram. Results are obtained in the bullion S16 running with 8 sockets. We have data for the other 6 applications, though we do not display them since they are qualitatively equivalent to the ones we show. When compared with the DEP baseline, SA and RIP-DEP achieve significant reductions in total coherence traffic of 1.99× and, 1.74×, respectively, in the case of Gauss-Seidel. Similarly, SA and RIP-DEP transfer 4.79× and 3.00× less data, respectively, in the case of Integral histogram. On average, using the geometric mean, SA and RIP-DEP achieve reductions of 3.08× and 2.28× with respect to DEP. These results clearly show the superiority of RIP-DEP over DEP as it dramatically reduces DEP's coherence traffic to similar levels to the partitions done by an expert programmer.

6.4 Load Imbalance and Overhead

We measure the overhead and load imbalance incurred by the different methods. Results are calculated using

$$LB = \frac{\sum_{i \in threads} \text{useful time of thread } i}{\max_{i \in threads} \{\text{useful time of thread } i\} \cdot \#threads} \cdot 100 \quad (2)$$

for the load balance, where the useful time of a thread is the total time the thread is executing user-level tasks. The overhead (OH) is defined as the percentage of time running runtime system routines over the wall clock time and the graph partitioning overhead (GP), included in OH, is the same ratio restricted to graph partitioning procedures. Table 1 reports maximum, minimum and mean results computed over the eight applications described in Section 5 running on the UV100.

Although its lack of data locality awareness makes DFIFO worse than the other approaches in terms of performance, it achieves the best load balance and the smallest runtime overhead with an average of 96.1 % and 1.77 %, respectively. RIP-DEP achieves very well balanced partitions, with an average of 88.7 %. The cost of doing an initial partition and propagating is, on average, equivalent to 3.02 % of the total execution time. In the case of RIP-MW repartitioning can slightly improve load balancing (90.8 %). Overall, our proposals incur minimal overheads and do not produce unbalanced partitions in the considered applications.

6.5 Adding Page Migration Mechanisms

In Figure 8, we show experiments adding page migration mechanisms to the automatic locality-aware proposals (DEP, RIP-DEP and RIP-MW). These mechanisms take care of moving the physical memory pages that contain the output data of the tasks to the socket that hosts the core executing the task. As the figure shows, page migration does not give benefit in general and is detrimental in many cases. This is mainly the case when not much data is written, which makes the migration an unnecessary overhead.

In the particular case of the QR factorization benchmark, however, RIP-MW with page movement is the only automatic approach

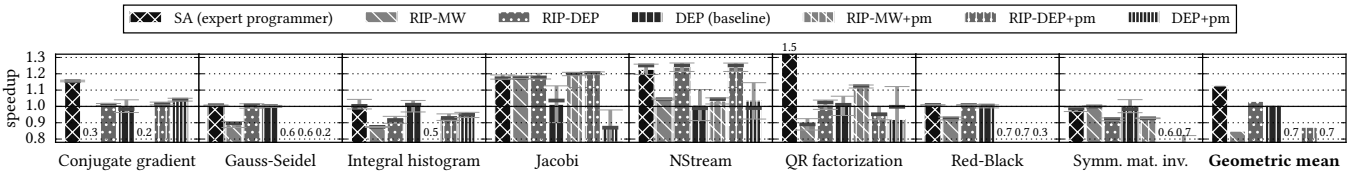


Figure 8: Speedup results in the SGI Altix UV100 using 3 sockets, 24 cores, with page migration mechanisms (marked as *pm*).

Table 1: Load balance (LB), runtime overheads (OH) and graph partitioning overheads (GP) in the SGI Altix UV100 using three sockets (as percentages, %).

		mean	min.	max.
DFIFO	**LB**	96.1	78.7	99.5
	OH	1.77	0.02	8.07
RIP-MW	**LB**	90.8	85.5	97.7
	OH	3.02	0.05	6.33
	GP	1.18	0.01	3.44
RIP-DEP	**LB**	88.7	79.7	94.9
	OH	3.02	0.03	13.44
	GP	0.030	0.000	0.089
SA	**LB**	92.9	86.9	99.2
	OH	3.84	0.03	23.28
DEP	**LB**	86.5	69.9	98.3
	OH	3.12	0.04	12.71

with positive results (1.12×). In order to understand this, consider the shape of the graph, which is a triangle pointing downwards similar to the Cholesky graph from Listing 1, and the way RIP-MW advances, shown in Figure 1. The partitioning algorithms generally aim at clustering connected tasks, so the first window is partitioned as three blocks (as many as sockets). At the same time, the QR algorithm does each step by working on an element of the diagonal, applying it to the rest of the matrix and discarding the whole row and column of that element afterwards. This means that load balancing mechanisms used by the graph partitioning algorithms schedule tasks created in consecutive windows in such a way that inter-socket data movement is sometimes unavoidable. For this reason, migrating the pages helps overcoming the remote accesses and makes sure that, when future partitions use the intersection, the sockets where the tasks are executed are the ones containing the data.

7 RELATED WORK AND EXISTING RESULTS

Techniques that take advantage of shared memory systems which integrate different memory devices have been studied for long time. For instance, Yan et al. [44] present the hierachical place trees (HPT), in which the programmer describes the memory hierarchy as a tree and the tasks are distributed on the tree leaves (where the workers reside) programatically on the source code of the application. Similarly, Chatterjee et al. [14] show a domain-specific language that allows the programmer to include the locality information using affinity groups for the tasks in a file separated from the application source code, making the approach more portable.

Graph partitioning has been used to statically assign tasks to processors in parallel machines [32]. This has been done mostly in two ways: i) dividing a graph where each vertex corresponds to a block of data and the edges represent simultaneous use of data by several processes, and ii) considering a process graph, mainly related to message-passing programming models, where each vertex corresponds to one of the processes and the edges represent communications between them. Our work is the first to dynamically apply graph partitioning to reduce NUMA effects on shared-memory systems, whereas prior proposals partition the graph statically.

One of the most recent developments to guide load balancing techniques via graph partitioning techniques is SPAWN by Papin et al. [31]. This approach assigns the tasks to the processing elements by a Voronoi tessellation. As the execution goes, the processing elements get an electrical charge value depending on the amount of work they have and the tessellation is thus updated.

There have been previous results in partitioning directed acyclic graphs using standard partitioners: Tanaka and Tatebe [37] used the multiple-constraint capabilities of METIS —that partition in a multidimensional space— to schedule workflows, which are typically more coarse-grained than shared-memory codes. Our results show a high overhead when using a similar approach in our context.

8 CONCLUSIONS AND FUTURE WORK

This work shows how graph partitioning methods can be leveraged to improve performance of parallel shared memory codes as well as to reduce data transfers across the system. The benefits of automatic approaches based on graph partitioning overcome the state-of-the-art without requiring expert programmer hints to drive the scheduling decisions.

Future work will go in the direction of taking even more advantage of the structure of the graph. The partitioner will be extended to get better performance with RIP-MW, which has the potential for achieving further performance improvements in applications that drastically change the structure of their TDG on runtime.

ACKNOWLEDGMENTS

This work has been supported by the RoMoL ERC Advanced Grant (GA 321253), by the European HiPEAC Network of Excellence, by the Spanish Ministry of Economy and Competitiveness (contract TIN2015-65316-P), by the Generalitat de Catalunya (contracts 2014-SGR-1051 and 2014-SGR-1272) and by the European Union's Horizon 2020 research and innovation programme (grant agreements 671697 and 779877). I. Sánchez Barrera has been partially supported by the Spanish Ministry of Education, Culture and Sport under Formación del Profesorado Universitario fellowship number FPU15/03612. M. Moretó has been partially supported by the Spanish Ministry of Economy, Industry and Competitiveness under Ramón y Cajal fellowship number RYC-2016-21104.

REFERENCES

[1] Rabab al Omairy, Guillermo Miranda, Hatem Ltaief, Rosa M. Badia, Xavier Martorell, Jesús Labarta, and David Keyes. 2015. Dense Matrix Computations on NUMA Architectures with Distance-Aware Work Stealing. *Supercomput. Front. Innov.* 2, 1 (Jan. 2015), 49–72. https://doi.org/10.14529/jsfi150103

[2] Atos [n. d.]. Bull bullion S16 Technical Specifications. Technical specifications. https://bull.com/wp-content/uploads/2016/08/f-bullion_s16_e7v3-en2_web.pdf

[3] Jairo Balart, Alejandro Duran, Marc Gonzàlez, Xavier Martorell, Eduard Ayguadé, and Jesús Labarta. 2004. Nanos Mercurium: A Research Compiler for OpenMP. In *6th European Workshop on OpenMP (EWOMP 2004)*. 103–109. http://people.ac.upc.edu/eduard/papers/paper_a31.pdf.gz

[4] Erik Boman, Karen Devine, Lee Ann Fisk, Robert Heaphy, Bruce Hendrickson, Vitus Leung, Courtenay Vaughan, Ümit V. Çatalyürek, Doruk Bozdag, and William Mitchell. 1999. Zoltan. Sandia National Laboratories. http://www.cs.sandia.gov/Zoltan

[5] Aydın Buluç, Henning Meyerhenke, Ilya Safro, Peter Sanders, and Christian Schulz. 2016. Recent Advances in Graph Partitioning. In *Algorithm Engineering: Selected Results and Surveys*, Lasse Kliemann and Peter Sanders (Eds.). Lecture Notes in Computer Science, Vol. 9220. Springer International Publishing, Cham, 117–158. https://doi.org/10.1007/978-3-319-49487-6_4 arXiv:1311.3144

[6] Alfredo Buttari, Julien Langou, Jakub Kurzak, and Jack Dongarra. 2008. Parallel Tiled QR Factorization for Multicore Architectures. *Concurr. Comput. Pract. Exp.* 20, 13 (July 2008), 1573–1590. https://doi.org/10.1002/cpe.1301

[7] Paul Caheny, Lluc Alvarez, Said Derradji, Mateo Valero, Miquel Moretó, and Marc Casas. 2018. Reducing Cache Coherence Traffic with a NUMA-Aware Runtime Approach. *IEEE Trans. Parallel Distrib. Syst.* 29, 5 (May 2018), 1174–1187. https://doi.org/10.1109/TPDS.2017.2787123

[8] Paul Caheny, Marc Casas, Miquel Moretó, Hervé Gloaguen, Maxime Saintes, Eduard Ayguadé, Jesús Labarta, and Mateo Valero. 2016. Reducing Cache Coherence Traffic with Hierarchical Directory Cache and NUMA-Aware Runtime Scheduling. In *International Conference on Parallel Architectures and Compilation (PACT '16)*. ACM, New York, NY, USA, 275–286. https://doi.org/10.1145/2967938.2967962

[9] Marc Casas, Rosa M. Badia, and Jesús Labarta. 2010. Automatic Phase Detection and Structure Extraction of MPI Applications. *Int. J. High Perform. Comput. Appl.* 24, 3 (Aug. 2010), 335–360. https://doi.org/10.1177/1094342009360039

[10] Marc Casas, Miquel Moretó, Lluc Alvarez, Emilio Castillo, Dimitrios Chasapis, Timothy Hayes, Luc Jaulmes, Oscar Palomar, Osman Unsal, Adrián Cristal, Eduard Ayguadé, Jesús Labarta, and Mateo Valero. 2015. Runtime-Aware Architectures. In *Euro-Par 2015: Parallel Processing*. Springer, Berlin, Heidelberg, 16–27. https://doi.org/10.1007/978-3-662-48096-0_2

[11] Ümit V. Çatalyürek. 2011. PaToH Graph Partitioner. http://www.cc.gatech.edu/~umit/software.html#patoh

[12] Ümit V. Çatalyürek and Cevdet Aykanat. 1999. Hypergraph-Partitioning-Based Decomposition for Parallel Sparse-Matrix Vector Multiplication. *IEEE Trans. Parallel Distrib. Syst.* 10, 7 (July 1999), 673–693. https://doi.org/10.1109/71.780863

[13] Charm 2016. Charm++ Programming Model. http://charmplusplus.org/

[14] Sanjay Chatterjee, Nick Vrvilo, Zoran Budimlic, Kathleen Knobe, and Vivek Sarkar. 2016. Declarative Tuning for Locality in Parallel Programs. In *45th International Conference on Parallel Processing (ICPP 2016)*. IEEE, 452–457. https://doi.org/10.1109/ICPP.2016.58

[15] Mohammad Dashti, Alexandra Fedorova, Justin Funston, Fabien Gaud, Renaud Lachaize, Baptiste Lepers, Vivien Quema, and Mark Roth. 2013. Traffic Management: A Holistic Approach to Memory Placement on NUMA Systems. In *18th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '13)*. ACM, New York, NY, USA, 381–394. https://doi.org/10.1145/2451116.2451157

[16] Matthias Diener, Eduardo H.M. Cruz, Philippe O.A. Navaux, Anselm Busse, and Hans-Ulrich Heiß. 2014. kMAF: Automatic Kernel-Level Management of Thread and Data Affinity. In *23rd International Conference on Parallel Architectures and Compilation (PACT '14)*. ACM, New York, NY, USA, 277–288. https://doi.org/10.1145/2628071.2628085

[17] Andi Drebes, Karine Heydemann, Nathalie Drach, Antoniu Pop, and Albert Cohen. 2014. Topology-Aware and Dependence-Aware Scheduling and Memory Allocation for Task-Parallel Languages. *ACM Trans. Archit. Code Optim.* 11, 3, Article 30 (Aug. 2014), 25 pages. https://doi.org/10.1145/2641764

[18] Andi Drebes, Antoniu Pop, Karine Heydemann, Albert Cohen, and Nathalie Drach. 2016. Scalable Task Parallelism for NUMA: A Uniform Abstraction for Coordinated Scheduling and Memory Management. In *International Conference on Parallel Architectures and Compilation (PACT '16)*. ACM, New York, NY, USA, 125–137. https://doi.org/10.1145/2967938.2967946

[19] Luc Jaulmes, Marc Casas, Miquel Moretó, Eduard Ayguadé, Jesús Labarta, and Mateo Valero. 2015. Exploiting Asynchrony from Exact Forward Recovery for DUE in Iterative Solvers. In *International Conference on High Performance Computing, Networking, Storage and Analysis (SC '15)*. ACM, New York, NY, USA, 53:1–53:12. https://doi.org/10.1145/2807591.2807599

[20] Laxmikant V. Kalé and Sanjeev Krishnan. 1996. Parallel Programming with Message-Driven Objects. In *Parallel Programming Using C++*, Gregory V. Wilson and Paul Lu (Eds.). MIT Press, Cambridge, MA, USA, 175–213.

[21] George Karypis and Vipin Kumar. 1997. Metis Graph Partitioner. http://glaros.dtc.umn.edu/gkhome/metis/metis/overview

[22] George Karypis and Vipin Kumar. 1998. A Fast and High Quality Multilevel Scheme for Partitioning Irregular Graphs. *SIAM J. Sci. Comput.* 20, 1 (Jan. 1998), 359–392. https://doi.org/10.1137/S1064827595287997

[23] George Karypis and Vipin Kumar. 1999. Multilevel *k*-Way Hypergraph Partitioning. In *36th Annual ACM/IEEE Design Automation Conference (DAC '99)*. ACM, New York, NY, USA, 343–348. https://doi.org/10.1145/309847.309954

[24] George Karypis and Vipin Kumar. 2007. hMetis Partitioning Software. http://glaros.dtc.umn.edu/gkhome/metis/hmetis/overview

[25] Madhavan Manivannan and Per Stenström. 2014. Runtime-Guided Cache Coherence Optimizations in Multi-Core Architectures. In *28th International Parallel and Distributed Processing Symposium (IPDPS 2014)*. IEEE, 625–636. https://doi.org/10.1109/IPDPS.2014.71

[26] Larry McAvoy and Carl Staelin. 1996. Lmbench: Portable Tools for Performance Analysis. In *USENIX 1996 Annual Technical Conference*. USENIX, 279–294. https://www.usenix.org/legacy/publications/library/proceedings/sd96/mcvoy.html

[27] John D. McCalpin. 1995. Memory Bandwidth and Machine Balance in Current High Performance Computers. *IEEE Comput. Soc. Tech. Comm. Comput. Archit. TCCA Newsl.* (1995), 19–25. http://www.cs.virginia.edu/stream/

[28] OpenBLAS 2016. OpenBLAS Library. http://www.openblas.net/

[29] OpenMP Committee. 2013. *OpenMP 4.0 Complete Specifications*. OpenMP Committee Technical Report. OpenMP Architecture Review Board. http://www.openmp.org/mp-documents/OpenMP4.0.0.pdf

[30] Vassilis Papaefstathiou, Manolis G.H. Katevenis, Dimitrios S. Nikolopoulos, and Dionisios Pnevmatikatos. 2013. Prefetching and Cache Management Using Task Lifetimes. In *27th ACM International Conference on Supercomputing (ICS '13)*. ACM, New York, NY, USA, 325–334. https://doi.org/10.1145/2464996.2465443

[31] Jean-Charles Papin, Christophe Denoual, Laurent Colombet, and Raymond Namyst. 2015. SPAWN: An Iterative, Potentials-Based, Dynamic Scheduling and Partitioning Tool. In *SC '15 - RESPA Workshop*. https://hal.inria.fr/hal-01223897

[32] François Pellegrini. 1994. Static Mapping by Dual Recursive Bipartitioning of Process Architecture Graphs. In *Scalable High Performance Computing Conference (SHPCC 1994)*. IEEE, 486–493. https://doi.org/10.1109/SHPCC.1994.296682

[33] François Pellegrini. 2012. SCOTCH. https://www.labri.fr/perso/pelegrin/scotch/

[34] François Pellegrini. 2014. Scotch and libScotch 6.0 User's Guide. http://gforge.inria.fr/docman/view.php/248/8260/scotch_user6.0.pdf

[35] Fatih Porikli. 2005. Integral Histogram: A Fast Way to Extract Histograms in Cartesian Spaces. In *IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR 2005)*, Vol. 1. IEEE, 829–836. https://doi.org/10.1109/CVPR.2005.188

[36] Maria Predari and Aurélien Esnard. 2016. A *k*-Way Greedy Graph Partitioning with Initial Fixed Vertices for Parallel Applications. In *24th Euromicro International Conference on Parallel, Distributed, and Network-Based Processing (PDP 2016)*. IEEE, 280–287. https://doi.org/10.1109/PDP.2016.109

[37] Masahiro Tanaka and Osamu Tatebe. 2012. Workflow Scheduling to Minimize Data Movement Using Multi-Constraint Graph Partitioning. In *12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid 2012)*. IEEE, 65–72. https://doi.org/10.1109/CCGrid.2012.134

[38] Xavier Teruel, Xavier Martorell, Alejandro Duran, Roger Ferrer, and Eduard Ayguadé. 2007. Support for OpenMP Tasks in Nanos V4. In *Conference of the Center for Advanced Studies on Collaborative Research (CASCON '07)*. IBM Corp., Riverton, NJ, USA, 256–259. https://doi.org/10.1145/1321211.1321241

[39] Mustafa M. Tikir and Jeffrey K. Hollingsworth. 2008. Hardware Monitors for Dynamic Page Migration. *J. Parallel Distrib. Comput.* 68, 9 (Sept. 2008), 1186–1200. https://doi.org/10.1016/j.jpdc.2008.05.006

[40] Mateo Valero, Miquel Moretó, Marc Casas, Eduard Ayguadé, and Jesús Labarta. 2014. Runtime-Aware Architectures: A First Approach. *Supercomput. Front. Innov.* 1, 1 (Sept. 2014), 28–43. https://doi.org/10.14529/jsfi140102

[41] Raul Vidal, Marc Casas, Miquel Moretó, Dimitrios Chasapis, Roger Ferrer, Xavier Martorell, Eduard Ayguadé, Jesús Labarta, and Mateo Valero. 2015. Evaluating the Impact of OpenMP 4.0 Extensions on Relevant Parallel Workloads. In *OpenMP: Heterogenous Execution and Data Movements. International Workshop on OpenMP (Lecture Notes in Computer Science)*. Springer, Cham, 60–72. https://doi.org/10.1007/978-3-319-24595-9_5

[42] Philippe Virouleau, François Broquedis, Thierry Gautier, and Fabrice Rastello. 2016. Using Data Dependencies to Improve Task-Based Scheduling Strategies on NUMA Architectures. In *Euro-Par 2016: Parallel Processing*. Springer, Cham, 531–544. https://doi.org/10.1007/978-3-319-43659-3_39

[43] Qian Wang, Xianyi Zhang, Yunquan Zhang, and Qing Yi. 2013. AUGEM: Automatically Generate High Performance Dense Linear Algebra Kernels on X86 CPUs. In *International Conference on High Performance Computing, Networking, Storage and Analysis (SC '13)*. ACM, New York, NY, USA, 25:1–25:12. https://doi.org/10.1145/2503210.2503219

[44] Yonghong Yan, Jisheng Zhao, Yi Guo, and Vivek Sarkar. 2009. Hierarchical Place Trees: A Portable Abstraction for Task Parallelism and Data Movement. In *Languages and Compilers for Parallel Computing*. Springer, Berlin, Heidelberg, 172–187. https://doi.org/10.1007/978-3-642-13374-9_12