Scaling Non-Regular Shared-Memory Codes by Reusing Custom Loop Schedules

Dimitrios S. Nikolopoulos[*]
Coordinated Science Laboratory
University of Illinois at Urbana-Champaign
1308 W. Main Street, Urbana, IL 61801
dsn@csrd.uiuc.edu

Ernest Artiaga, Eduard Ayguadé, Jesús Labarta
Department d' Arquitectura de Computadors
Universitat Politecnica de Catalunya
c/Jordi Girona 1–3, Modul D6
Barcelona, 08034 – Spain
{ernest, eduard, jesus}@ac.upc.es

Abstract

In this paper we explore the idea of customizing and reusing loop schedules to improve the scalability of non-regular numerical codes in shared–memory architectures with non–uniform memory access latency. The main objective is to implicitly setup affinity links between threads and data, by devising loop schedules that achieve balanced work distribution within irregular data spaces and reusing them as much as possible along the execution of the program for better memory access locality. This transformation provides a great deal of flexibility in optimizing locality, without compromising the simplicity of the shared-memory programming paradigm. In particular, the programmer does not need to explicitly distribute data between processors. The paper presents practical examples from real applications and experiments showing the efficiency of the approach.

Keywords OpenMP, shared–memory programming models, page placement, data and computation affinity

[*]Corresponding author, present address: Department of Computer Science, The College of William&Mary, McGlothlin Street Hall, Williamsburg, VA 23187–8795. Tel. 757–221–3455, fax. 757–221–1717.

1 INTRODUCTION

Programming models based on the abstraction of a shared address space became popular because they could potentially eliminate a number of tasks that make parallel programming difficult, such as the placement of data in memory, the assignment of computation to processors, and the management of communication. Parallelism can be expressed simply by pin-pointing loops and fragments of sequential code that can be safely executed in different threads, using compiler directives [16]. Unfortunately, these otherwise desirable features of shared-memory programming models are also the ones that make the use of these models problematic in scalable parallel architectures.

Scalable shared-memory multiprocessors use a LEGO architecture, in which off-the-shelf or proprietary computational nodes with processors and memory are interconnected via a fast switching network [10]. This setting is identical to that of distributed-memory architectures, with the exception that the nodes run a directory-based cache coherence protocol at their communication interfaces. The protocol allows processors to use their caches for coherent migration and replication of data, regardless of the location of data in memory. The programmer views the memory of the system as a flat, globally accessible address space and can exploit the caches to enable fast access to shared data. However, the cost of memory accesses upon cache misses varies, depending on whether the accesses are to locally or remotely located data. If the placement of data across nodes does not match the memory access pattern of the program, performance may suffer from the latency of remote memory accesses, which is several times higher than the latency of local memory accesses.

The parallel processing community has been addressing this problem by incorporating data and thread placement facilities in shared-memory programming models [3, 4, 17]. Albeit effective, this solution sacrifices the transparency of the shared-memory programming abstraction, by exposing architectural state to the programs. Shared-memory programming paradigms are fundamentally based on location transparency for both data and computation. Data distribution complicates the programming process and the underlying compilation and execution framework.

As an alternative to data distribution, we have proposed a dynamic optimization framework, for maximizing memory access locality in programs written with an architecture-agnostic shared-memory programming model, such as OpenMP [14]. The idea is to dynamically record the memory access pattern of the program while the program is running, and, if the complete memory access pattern is periodic, optimize data placement for the specific access pattern. Dynamic optimization of data placement is performed by migrating each page to the memory of the node that accesses the page more frequently during the execution of the program. This technique works extremely well, yielding performance as good as that of the best manual data distribution algorithms for a large number of parallel codes that have *strictly periodic* structure, i.e. they repeat the same parallel computation for a number of iterations [13].

The advantage of dynamic optimization is that it requires no modifications or extensions to the programming model. It is a purely runtime scheme that needs minimal compiler support for instrumenting the program to collect memory access traces and invoke a dynamic data distribution engine. It can also be used as a convenient tool for dynamic compilation and optimization of parallel programs, when the cost of runtime data distribution is prohibitive. Our dynamic optimization framework has been successful as a transparent optimizer of memory access locality in several OpenMP codes [12, 13].

1.1 Problem Statement

Although we have been able to use dynamic optimization of data placement in several OpenMP programs without modifications to the programming interface, this approach is limited by the fact that not all parallel codes are amenable to dynamic data placement optimizations. Our optimization framework relies on a periodic memory access pattern to optimize data placement for the program as a whole. Unfortunately, several parallel codes in use today do not have this property.

Some parallel codes have a dynamic memory access pattern, which changes with the evolution of the computation. The plight of our dynamic optimization scheme in these codes is the inability to speculate on the future memory accesses of the program based on a snapshot of the access pattern retrieved early during the execution. An optimization scheme based on the access rates to each page in memory is effective only if the memory access pattern has sufficient temporal locality (i.e. recent memory accesses are likely to provide a prediction for future memory accesses) and if the data distribution engine is able to identify phase changes in the memory access pattern. Although techniques for sampling and decaying memory access history to gauge dynamic memory access patterns have appeared in the literature [14, 18, 20], it is questionable if these techniques form a general solution.

A second problem is that dynamic optimization of data distribution is only one aspect of the performance tuning process for scalable shared-memory architectures. The balanced distribution of computation among processors is a second critical aspect, in which dynamic data distribution by itself can not be of much help. Dynamic optimization of data placement is always performed for a given work distribution scheme and is inherently orthogonal to load balancing. The penalty of load imbalance may well limit the scalability of the program, even if memory access locality within the program is optimized. Unfortunately, shared-memory programming standards like OpenMP lack the means to express flexible work distributions for load balancing purposes.

Load balancing, pretty much like data distribution can be dynamically optimized under the assumption that the computation in the program has some form of periodicity, so that the load imbalance can be exposed and resolved at runtime. Such an approach is outlined in [15]. The weakness of this solution, in addition to the inability to handle aperiodic computation patterns, is that it can not balance the load according to the physical properties of the problem modelled by the parallel computation. It can only alleviate the load imbalance incurred from an arbitrary static work distribution and up to the point where a measurable index of load balancing (e.g. floating point operations per processor) can not be further improved.

1.2 Contributions of the Paper

This paper presents an effective technique and the associated program transformations for implementing application-specific work distributions and simultaneously optimize memory access locality in shared-memory programming paradigms, without manual data distribution. In principle, we target array-based numerical codes, in which the bulk of the computation is executed in loop nests.

Our scheme relaxes the constraint of entirely transparent optimization, by allowing the programmer to encode application-specific work distribution schemes. The novelty of this scheme is that both load balancing and memory access locality are achieved by proper scheduling of loop iterations to processors. Effective loop schedules are identified and reused throughout the program, across executions of the same loop or across executions of different loops with overlapping access regions. The distribution of data is optimized implicitly, by exploiting the operating system's automatic page placement algorithm.

The proposed scheme provides the programmer with flexibility that can otherwise be provided only with data distribution statements and a data-centric model integrated with the shared-memory programming abstraction. Although the programming effort for proper thread and data distribution is not eliminated (the programmer must still express the desired correlation between computation and data), the scheme is more appropriate for shared-memory parallel programming, because it operates only within the scope of loop scheduling and implements coordinated rather than decoupled placement of computation and data, thus minimizing the associated overhead. Since directive-based shared-memory programming paradigms like OpenMP already allow some flexibility in the selection of loop schedules from predefined alternatives, adding our transformations as an option to the loop schedule clauses of parallelization directives seems to be a reasonable extension. Reusable loop schedules can be nicely expressed with *affinity clauses* in directives enclosing parallel loops and can be translated to parallel code with simple loop transformations.

We implemented and tested our technique in the familiar OpenMP framework. OpenMP is the de facto standard for parallel programming with the shared-memory abstraction and has been deployed widely in small-scale shared-memory architectures and more recently, in scalable NUMA multiprocessors and clusters. Currently, our technique handles effectively two types of OpenMP codes: First, codes where although the memory access pattern is aperiodic, processors can exploit memory access locality by reusing a significant amount of the data that they access during the course of the computation. Second, OpenMP codes that model irregular problem spaces, using irregularly shaped grids. In these codes, our technique enables the programmer to implement application-specific work distribution schemes, while optimizing transparently data distribution.

As far as performance is concerned, our scheme improves the performance of OpenMP code by more than 50%, compared to automatic data and work distribution algorithms implemented in the operating system and the runtime system respectively. Our technique outperforms slightly hybrid parallelization schemes using OpenMP and manual data distribution (by 5-10%), because it performs locality-conscious distribution of data and computation simultaneously. Data distribution is performed *lazily* and in parallel, whenever the processors experience page faults on unmapped pages during the execution of useful computation. In the case of manual data distribution, a higher cost is paid before the actual parallel computation, by either having one processor call the operating system to place data on the appropriate nodes, or inserting a dummy parallel loop that forces each processor to map locally the data assigned to it.

The drawback of our scheme is that it is prone to false-sharing, whenever the blocks of data assigned to each processor are not page-size aligned. To circumvent this problem, it is necessary to use techniques such as array reshaping and index rewriting, thus placing more burden on the compiler. Fortunately, previous work on data parallel languages formalized the related techniques to a significant extent, therefore it is reasonable to have such an expectation from an advanced OpenMP compiler. We plan to address the relevant issues in future work. At the time being, we use manual transformations of arrays to cope with false sharing.

1.3 The rest of this paper

The rest of this paper is organized as follows: Section 2 illustrates two motivating examples for reusing custom loop schedules, a simple LU decomposition and an irregular data transposition kernel. Section 3 describes the most essential details of our transformations. Section 4 provides results from experiments with non-regular parallel codes, which compare our scheme against a shared-memory parallelization scheme which is oblivious to data distribution, a scheme which combines shared-memory parallelism and manual data distribution and implementations of the same codes with MPI. Section 5 reviews related work and Section 6 summarizes the paper.

2 Motivating Examples

This section provides two examples to highlight the issues that motivate the use of customizing and reusing loop schedules for optimizing memory access locality in shared-memory codes. Section 2.1 examines LU decomposition, a code in which although the memory access pattern of the program is aperiodic, there is a significant amount of data reuse that can be exploited by distributing data and scheduling appropriately the parallel loop. Section 2.2 presents a data transposition kernel from a weather forecasting system, which requires an irregular two-dimensional block distribution to balance the computational load among processors.

In the following discussion, we assume that the target architecture is a hardware cache-coherent, distributed shared-memory (DSM) multiprocessor, such as the SGI Origin2000 [9], the Sun Wildfire [7], and the Compaq GS320 AlphaServer [6]. The performance optimizations that we are seeking for in these architectures are of two kinds. First, we wish to distribute the data of each program among the nodes of the system, so that the processors on each node access local memory as frequently as possible and remote memory as infrequently as possible, whenever they miss in their caches. Second, we wish to distribute the work between processors, so that the work distribution balances the load according to the structure of the data space modelled by the application.

We take into account the fact that the operating system uses automatic page placement algorithms that distribute pages with the data of the program across the nodes of the system. The most popular of these algorithms is *first-touch* [11], which places each page on the same node with the processor that accesses the page first during the course of execution. First-touch is used in commercial operating systems such as IRIX and Solaris. Although first-touch is oblivious of the memory access pattern of the program, it is a policy able to attain satisfactory memory access locality in many practical cases.

2.1 LU

Consider the simple LU decomposition code shown in Figure 1(a). The code divides the element in column `k` of `a` with the pivot element and then updates the submatrix `a[k+1:n,k+1:n]`. We assume that the code is parallelized with a flat shared-memory model, by inserting a compiler directive that encloses the inner `j` loop and commands its parallel execution. The `m` loop can also be parallelized, but we omit this option here to simplify the discussion. We use OpenMP directives in the code. The example is taken from [3].

Conceptually, according to the memory access pattern of the parallelized loop, memory access locality will be better if the columns of `a` are distributed among processors. The problem is how to distribute the columns, so that processors can actually reuse data and avoid remote memory accesses. The default algorithm for distributing the iterations of a parallel loop among processors in OpenMP is the static algorithm, which assigns n/p consecutive iterations to each processor, where n is the number of iterations in the parallel loop and p the number of processors. In the case of LU, this algorithm implements implicitly a block distribution of the columns of `a` among processors, under two assumptions: First, that the operating system uses the first-touch page placement algorithm, so that each processor maps locally the columns of `a` that it updates first during the first iteration of the outer `k` loop; and second, that each column of `a` is page-aligned. The first requirement is usually met. Most popular commercial DSM multiprocessors use first-touch page placement in the operating system [6, 7, 9]. The requirement for page alignment can be met with additional compiler support or with programmer intervention, to pad and/or reshape `a` along the second dimension.

Figure 2(a) shows the layout of the elements of a 16×16 array, if the inner parallel loop of LU is parallelized and scheduled statically on four processors. The figure demonstrates the problem with the block distribution of the columns of `a`. With this layout, in every iteration of the `k` loop except the first one, at least one processor has to update one or more columns of `a` that reside in remote memories. This happens because for `k` ≥ 2, the work of the processors in the parallelized inner loop is redistributed so that each processor updates `(n-k)/p` consecutive columns of the submatrix `a[k+1:n,k+1:n]`. Figure 2(b) shows the partition of the array which is accessed during the 8th iteration of the outer loop (surrounded with boldface lines). Assuming that processors are numbered from 0 to 3 and from left to right, processors 0 and 1 will update four columns which are local to processor 2, processor 2 will update two columns which are local to processor 3 and processor 3 will be the only processor that will update local columns. 48 out of the 64 elements of the submatrix updated during the 8th iteration of the outer loop will be updated with remote memory accesses. It could be possible to handle this case by migrating the pages with the columns to the processors that access them in any given iteration on demand. This solution however may be inhibited by the cost of page migration. The amount of computation per column should be sufficient to balance the cost of migrating the pages that store the column, which may be as high as one ms. per page on state-of-the-art systems like the Origin. Therefore, this solution is viable only in codes with very large problem sizes.

The way to overcome this problem is to distribute data and/or computation, so that in every iteration of the outer loop, each processor updates only columns that reside in local memories. Since the computation works in one direction towards smaller submatrices of `a`, the way to achieve this is to have processors work on columns scattered across the array in the first iteration and update a subset of the same columns in subsequent iterations. One way to implement the desired data distribution and simultaneously balance the load, is to distribute the columns of `a` in a cyclic fashion as shown in Figure 2(c) and schedule the inner parallel loop so that processor `i` updates only columns `j` for which `j` mod (n/p) = `i`. As Figure 2(d) shows, with this data distribution in iteration 8, processor 0 will update columns 8 and 12 which are local, processor 1 will update columns 9 and 13 which are also local and so on.

Figure 1(b) shows how this is achieved with directives for manual data distribution and affinity scheduling. The functionality of the `!$distribute` directive is identical to that of the corresponding HPF directive [5]. The second dimension (i.e. the columns) of `a` are distributed in a cyclic manner across the processors that execute the program. The `!$affinity` directive is in analogy to the `!$onhome` clause of HPF and has been proposed in previous work as an extension to shared-memory programming paradigms that helps the programmer express mappings of computation that enforce memory access locality [3, 4].

What we try to circumvent with the work presented in this paper is the requirement to explicitly distribute data in codes like LU, whenever the desired collocation of computation and data can be achieved by letting the operating system place data in memory and in parallel, move the right pieces of computation close to the data they access. We show that in many cases, this can be done easily by carefully scheduling loop iterations to processors. We wish to avoid the implications of data distribution on the complexity of the programming model and the implementation of the compiler. We also wish to eliminate the overhead of manual data distribution and try to overlap automatic data distribution with computation, while achieving the same effect as the best data distribution algorithm for the program at hand.

We decide to tolerate an extension of the programming model that expresses affinity of computation to data for two reasons. First, such an extension can be expressed as part of the clauses that define the scheduling algorithm for parallel loops. Such flexible work distributions are already considered in shared-memory programming models and OpenMP in particular [16]. Second, there are several codes in which the affinity relation between computation and data depends on an application-specific distribution of computation, which can not be analyzed by the compiler or inferred at runtime. The next subsection presents one example.

2.2 The IFS LG Kernel

Figure 3 shows the HPF implementation of a snippet from the LG kernel, a data transposition routine which is part of the Integrated Forecasts System of the European Center for Medium-Range Weather Forecasts [19]. The LG kernel transposes a grid which models the earth's atmosphere, from the physical space to the Fourier space. Both the original and the transposed grids are irregular and use more points to model the parts of the atmosphere which are close the equatorial and less points to model the parts of the atmosphere which are close to the poles. The snippet shown in Figure 3 updates the elements of one of the most frequently accessed array in the code (`zgl`).

LG is an irregular parallel code, as far as the memory access pattern is concerned. The peculiar feature of this code is that the physical problem that it models has some form of structural irregularity, which makes certain regions of the modelled data space more densely populated with data points than others. Such a grid requires an application-specific load balancing algorithm. More specifically, the decomposition of the grid among processors has to be done with an unstructured block distribution, like the one shown in Figure 4(a).

For proper load balancing, the code requires a two-dimensional block distribution, where the size of the blocks along the vertical dimension is variable. Blocks assigned to processors that work on the north/south edges of the grid (i.e. close to the poles) are larger than blocks assigned to processors that work on other parts of the grid. As shown in Figure 3, the HPF solution to model such a grid is to define a *generalized block* distribution for the first dimension of the array, in which the size of each block is defined in a vector (`mapgla`) of size equal to the number of processors among which the array is distributed along the vertical dimension. The second dimension of the array is distributed with an indirect distribution, which actually collapses into a balanced block distribution of the second dimension, with an irregular ordering of accesses to columns of the array. Therefore, we do not treat it as a special case.

Likewise to LU, the problem with the LG kernel is that if the code is parallelized with a flat shared-memory model using compiler directives, static scheduling of parallel loops and automatic first-touch page placement by the operating system, the columns of the array will be distributed blockwise, which will force processors to access remotely located data most of the time. The same will happen if the loop is interchanged so that the array is distributed rowwise (see Figure 4(b)), and in addition, load balancing will be compromised, since the size of the blocks of rows assigned to each processor will be equal. This is undesirable, because the amount of work assigned to the topmost(bottommost) rows is less than the amount of work assigned to the other rows.

HPF handles this case by defining the irregular generalized block distribution and executing the loop that updates the elements of `zgl` as a triple-nested loop which iterates over the processor number. The bounds of the innermost two loops are the bounds of the blocks assigned to each processor according to the specified distribution of `zgl`. We wish to achieve the same effect in the shared-memory programming model using only extensions for flexible scheduling of loop iterations to processors and the automatic page placement algorithm of the operating system.

3 Reusing Customized Loop Schedules

The idea behind customizing and reusing loop schedules is to implement application-specific algorithms for work and data distribution in shared-memory codes, by scheduling appropriately the iterations of parallel loops. What makes this idea work, is the ability to coordinate the distribution of work with the distribution of data, which is performed automatically by the operating system. If the mapping of loop iterations to processors can match the data placement algorithm implemented by the OS, it is possible to implement arbitrary data distribution schemes, without having to extend the programming model with data distribution statements.

We exploit the fact that most operating systems of DSM multiprocessors use the first-touch page placement algorithm. First-touch provides an elegant way of mapping data to processors. If each processor *touches* the data that we wish to map to it first, the desired data distribution is performed implicitly and transparently to the programmer. In fact, in most cases, data distribution with first-touch can be performed on-the-fly, during the execution of the parallel computation. This is beneficial because it avoids the overhead of manually calling the operating system to place data before executing useful work.

Given the first-touch page placement algorithm, the only thing that needs to be done for implementing arbitrary data distributions is to restructure the parallel computation so that each processor accesses first the data that the desired distribution maps to it. In principle, this is possible by rewriting the loop so that it iterates over the processor number and the innermost iterations touch the data assigned to each processor. This is essentially the same approach used in earlier implementations of HPF [8].

The important limitation of this scheme is that the data assigned to each processor should be page-aligned. If not, it is likely that processors will map locally pages with significant amounts of data that “belong,, to other processors. As a consequence, false sharing will occur, the number of remote memory accesses will be increased and the program might suffer from high waiting times in the memory system. In many practical cases, this problem can be relatively easily circumvented by padding certain array dimensions, or by adding one dimension (the processor number dimension) to the array, a transformation known as array reshaping [1, 4]. Although these techniques have been formalized for automation in a restructuring compiler, they are not generally available. In this work, we apply these techniques to avoid false sharing via manual array transformations.

We demonstrate how this strategy works for the examples presented in Section 2. Consider LU. Figure 5 shows how the code can be restructured to implement the cyclic distribution of the columns of `a` and ensure that each processor updates only local columns in all iterations of the outer loop. Iterations of `a` are assigned to processors in a cyclic manner by executing the loop with a step equal to the number of processors. During the `k`-th iteration of the outer loop, each processor executes a subset of the iterations that the same processor executed during the `k-1`-th iteration of the outer loop. For example, assume that the program is executed with 4 processors. When `k=1`, processor 0 executes iterations 2,6,10,14,. . . , processor 1 executes iterations 3,7,11,15,. . . and so on. In the second iteration, processor 0 executes iterations 6,10,14 . . . , processor 1 executes iterations 7,11,15,. . . etc.

The initial cyclic assignment of iterations to processors is equivalent to a cyclic distribution of the columns of `a`. By reusing the initial schedule of the innermost parallel loop, we ensure that each processor updates a subset of the data that it updates during the first iteration of the outermost `k` loop. The appealing property of this scheme is that data is actually distributed while the processors execute useful computation, i.e. the first computational iteration of LU. There is no need to predistribute the data using manual data distribution and the overhead of data distribution is removed from initialization and overlapped with computation, so that it has a lesser impact on the execution time of the program.

This scheme can be extended to work with sequences of parallel loops that might have different bounds but update the same data. If the first loop of this sequence is restructured for localizing memory accesses, the schedule obtained for this loop can be applied to subsequent loops, so that the data access pattern of these loops matches the data distribution that the first loop implements.

In cases where the data access pattern needs to be changed (e.g. across loops that update or access different data), it is possible to discard any previously established distribution of data by unmapping the pages that contain elements of distributed arrays, using calls similar to the UNIX `mprotect()`. The side-effect of `mprotect()` is that pages with distributed data become invalid and will cause page faults whenever a processor access them after raising their protection bits. The first execution of the loop that changes the memory access pattern will force the pages to be remapped to processors on a first-touch basis and according to the new memory access pattern. This is an implicit mechanism for data redistribution, which extends the applicability of customized loop schedules to codes with dynamically changing memory access patterns.

Figure 6 shows a customized loop schedule that implements the irregular block distribution required by the IFS LG kernel. The idea is again to have processors touch data assigned to them first, so that the associated pages are placed in local memory modules. The generalized block distribution is handled by defining the bounds of the block assigned to each processor (first parallel loop in the code). `mapgla`, which stores the number of rows of `zgl` assigned to each processor, is used to define the lower and upper bound of the loop. Two-dimensional blocking is then applied to the parallel loop. Note that `nproca` and `nprocb` are the number of processors used for distribution along the horizontal and vertical direction respectively. Each block assigned to a processor along the vertical direction is `ngt0/nprocb` columns wide. Processor `p` accesses the block of columns `j`, where `p` mod `nprocb`

= `j`. The technique is no different than previously proposed techniques to access data distributed with multidimensional distributions [4] and is straightforward to extend for handling combinations of block and cyclic distributions, with potentially variable sizes for the blocks and the chunks of rows/columns assigned to each processor.

Figure 7 illustrates an example of how proper assignment of loop iterations to processors implements implicitly an indirect data distribution, using the first-touch page placement algorithm. The example shows another excerpt from the LG kernel. We assume that the number of rows assigned to each processor is such that the elements on the part of the column assigned to a processor are page-aligned. The indirect distribution is defined by an indirection map, which is obtained by accessing the values of vector `indl`. In order to implement the indirect block distribution by assigning iterations to processors, we identify the iterations that access the elements of the block assigned to each processor, as shown in the first code fragment in Figure 7. The array element `rindl(j)` stores the iteration of the loop that accesses the elements of row `indl(j)` of `zgl`. These elements must be mapped to the processor that *owns* `indl(j)`. This is implemented by constructing a map of iterations to processors, which is defined as a two-dimensional array `myiter(i,j), i=1, ...p, j=1, ...max(mapgla(i))`. The elements of this array are set with the second code fragment shown in Figure 7. Intuitively, if an element $\mathtt{i}_1$ is assigned to processor `p`, we first find the iteration $\mathtt{j}_1$ that accesses $\mathtt{i}_1$, by finding the value $\mathtt{j}_1$ that satisfies $\mathtt{indl}(\mathtt{j}_1) = \mathtt{i}_1$. We then set $\mathit{rindl}(i_1) = j_1$ and assign iteration $\mathtt{j}_1$ to processor `p` by setting $\mathtt{myiter(p,k)} = \mathtt{j}_1$ for some `k`. Finally, the original loop is transformed so that each processor executes its assigned set of iterations, as shown in the third code fragment in Figure 7.

In a practical implementation, the aforementioned loop scheduling transformations can be easily automated in an extension of the `SCHEDULE` clause of the OpenMP programming standard. In analogy to data-parallel directives implemented in variants of HPF, the `SCHEDULE` clause may include a `GEN_BLOCK(map(1:P))` parameter or an `INDIRECT(map(1:N))` parameter. In the first case, element `i` of `map` contains the size of a contiguous chunk of iterations assigned to processor `i`. In the second case, element `i` of `MAP` contains the mapping of an element of a shared array to a processor, along the dimension of the array indexed by the running index of the parallelized loop. The OpenMP compiler should interpret this as a mapping of the iteration that updates this element to the same processor. Similarly, in the case of LU, a clause of the type `cyclic,affinity(j)=data(a(i,j))` would instruct the compiler to schedule the iterations of the loop cyclically and reuse this schedule across invocations of the loop.

4 RESULTS

4.1 Experimental Setting

We present experimental results that demonstrate the potential of reusing customized loop schedules for achieving good memory access locality. We experimented on a 128-processor SGI Origin2000 located at NCSA. This system has MIPS R10000 processors running at 250 MHz, with 32 Kilobytes of split L1 cache, 4 Megabytes of unified L2 cache per processor, and 64 Gigabytes of uniformly distributed DRAM memory. The operating system used is IRIX version 6.5.11. The page size for data pages on the Origin2000 is 16 Kilobytes. All experiments were submitted to benchmarking queues and they were executed on dedicated processors.

We experimented with four codes. LU and the LG kernel were already presented as examples in the previous sections. We also performed experiments with SL and TS, two irregular data transposition kernels taken from the IFS weather forecast code. The data kernels perform transpositions of data between the three main computational phases of IFS, namely the physical grid-point space computation, the Fourier space computation, and the spectral space computation. These transpositions are performed to ensure that the computational parts of IFS are executed in parallel without interprocessor communication. Data transpositions in the IFS code can be implemented with appropriate data redistribution. Unfortunately, the grids of the main computational phases of IFS cannot be represented with regular (e.g. *BLOCK* or *CYCLIC*) data distributions. The physical space grid and the Fourier space grid are quasi-regular, because the number of grid points (used to model the atmosphere) is progressively reduced when moving from the equatorial to the poles. The spectral space grid, which is produced from a Legendre transform of the Fourier space grid, has a triangular shape.

The LG kernel handles the transpositions of data between the physical grid point space and the

Fourier space. The SL kernel computes a trajectory from a grid point backwards in time and interpolates some quantities at the departure and the mid point of the trajectory, using the semi-Lagrangian method. The main computational challenge in a parallel implementation of SL is that computing the trajectory requires that each processor collects a set of global grid point indices from neighboring processors. These grid points are represented by a compact read-only data structure, called a *halo*. The halo is updated at runtime according to the winds which are likely to be encountered in the trajectory. The TS kernel uses Fourier and Legendre transforms to transpose data from the Fourier space to the spectral space and backwards.

The original implementation of the codes uses MPI. The codes are parallelized by decomposing the grids between processors for balanced load, according to the shape and the population of different parts of the grids. Communication follows nearest-neighbor patterns and is manually optimized.

For LU, we compare the three OpenMP implementations (loop-parallel version, version with manual data distribution and version with loop schedule reuse). For the three irregular codes, we compare the performance of four versions of each code. The first version is the original MPI implementation. The second version is an OpenMP implementation derived from the HPF implementation of the codes [2], by parallelizing the loops denoted as `independent` in the HPF implementation, and applying reordering of loop nests, so that inner loops work along columns of the arrays for better spatial and temporal cache locality. The third version uses OpenMP and the customized loop schedules, via manual transformations that we applied to the codes. We note that these transformations are straightforward to implement in an OpenMP compiler, assuming that a user gives a description of the thread-to-data affinity relationship in the loop's `SCHEDULE` clause.

The fourth version is a hybrid data-parallel/OpenMP version which uses manual data distribution. While developing this version, we had the option of using the native SGI compiler, which implements multidimensional block and cyclic data distributions, in conjunction with affinity mapping of threads to data. These options are enabled with compiler directives similar to the ones used in HPF. Unfortunately, we had to disqualify this option for two reasons. First, the SGI implementation has been performing poorly in several experiments we did with the affinity scheduling clause of the SGI compiler. Second, it is impossible to implement the appropriate data distributions for the irregular kernels using the SGI directives. We reverted to a brute-force solution and placed manually the pages with the elements of the distributed arrays across processors. We applied array reshaping and padding, together with rewriting of array access indices, as needed for the accurate implementation of the irregular two-dimensional distributions. We purposely didn't apply reshaping or padding in the versions that use loop schedule reuse, to evaluate the impact of false sharing in the performance of the codes.

4.2 Results

Figure 8 illustrates the execution times of LU decomposition performed on a dense 4096×4096 matrix and the three irregular kernels operating on a 63×63 grid respectively. Execution times are plotted from 1 to 128 processors for LU and from to 1 to 100 processors for the three irregular kernels. The latter require a square number of processors for the grid decomposition. Note that execution time is plotted in logarithmic scale and the lower/upper bounds are adjusted according to the execution times of the benchmarks. Note also that for the irregular codes, we report the execution time per iteration, averaged over 100 iterations.

There is a highly consistent performance trend in all four benchmarks. The loop schedule reuse transformation improves the performance of the unmodified OpenMP implementation at least as much and in most cases slightly more than manual data distribution. This verifies the common belief that some form of guided data distribution is necessary for shared-memory programs running on NUMA architectures, but also shows that data distribution can be implemented implicitly and in parallel with the execution of useful computation. The advantage of our implicit data distribution mechanism compared to manual data distribution is attributed to the reduced overhead of our scheme, which distributes data during rather than before the parallel computation.

The OpenMP versions that use customized loop schedules perform within 5% off MPI in the irregular kernels. The same versions outperform the versions that use manual data distribution by up to 13% and the plain OpenMP versions by a margin that ranges between 23% and 55%. The notable exception is SL, where the performance of loop schedule reuse suffers from false sharing.

The message from the presented results is that it is possible to obtain the full benefit of memory access locality without introducing data distribution extensions to OpenMP. The comparison with MPI is of particular interest, first because it is among the first to contradict the existing experimental evidence that position OpenMP behind MPI in terms of performance and scalability, and second because the programming effort required to reach this level of performance with OpenMP is one order of magnitude less than the programming effort required to reach the same level of performance with MPI [15].

To quantify the improvement in memory access locality from customizing and reusing loop schedules, we traced the memory accesses in the programs and calculated the amount of local and remote memory accesses issued to each node, during the executions of the programs on 64 processors. Figures 9 through 12 show these results. The processors on the Origin2000 are attached to nodes with two processors per node. The processors in a node share the memory of the node. The histograms show the accumulated number of memory accesses per node, divided into local accesses (i.e. accesses from the processors on the node, gray part of the bars) and remote accesses (i.e. accesses from processors outside the node, black part of the bars).

Aside from reducing radically memory latency by reducing the number of remote memory accesses per node, the schedule reuse transformation helps in alleviating contention at memory modules. Contention is alleviated by balancing the remote memory accesses across the nodes of the system. This is crucial for distributing evenly the traffic of messages in the interconnection network. Memory access balancing is almost excellent in LU and LG, when iteration schedule reuse is applied. TS has a somewhat more unbalanced memory access pattern, but the overall number of remote memory accesses is reduced significantly.

SL has severe false sharing in pages that are accessed by neighboring processors. To circumvent this problem, we applied array padding in the version that uses loop schedule reuse. We transformed the primary array (`zsl1`) as shown in Figure 13. The array is distributed with an implicit indirect distribution along the first dimension, likewise to the example in Figure 7. The result of this optimization is shown in Figure 14 and validates the argument about false sharing and the effectiveness of the solution. Figure 15 shows how the simple padding transformation reduces and balances remote memory accesses in SL.

5 Related Work

Data distribution was explored in depth in various projects that investigated data-parallel programming languages and in particular, High Performance Fortran. The works of Benkner et.al. [2] and Hirandani et.al. [8] are probably the most relevant to our work, since they also explored the option of reusing schedules for improving the runtime performance of message-passing code obtained from translating HPF directives. Instead of reusing loop schedules though, these works propose to reuse communication schedules, once these schedules are obtained in the first iteration of the computation. Our work proposes to reuse iteration schedules, so that computation is moved to data dynamically, while the computation is in progress and the operating system places data in memory according to its local algorithm.

Anderson et.al. [1] and Hirandani et.al. [8] proposed loop transformations for mapping loop iterations to data, according to multidimensional data distributions. We use similar transformations to implement loop schedules that implicitly set up arbitrary distributions of computation and data. We note that these transformations are beneficial not only to memory access locality but also to cache access locality, particularly if the transformed loops are executed multiple times within the same program.

The authors were the first to propose runtime tracing of memory accesses as a method for dynamic data distribution [13, 14] and extended this work to handle codes where dynamic data distribution should coordinate with a dynamic load balancing algorithm [15]. This work extends this framework in applications where dynamic optimization is difficult, due to the lack of periodicity in the memory access pattern, or due to the inability of automatic scheduling algorithms to implement application-specific load distributions.

6 CONCLUSIONS

On scalable multiprocessor architectures, shared-memory parallelization suffers often from poor performance. This happens particularly in codes where computation and data must be aligned in an application-specific manner, due to structural irregularities of the modelled physical problem and/or lack of periodicity in the data access pattern. In this paper we have proposed the technique of reusing customized loop schedules, as a simple transformation for improving memory access locality in such programs, without manual data distribution. We have shown how customized loop schedules can be used in OpenMP to implement irregular data distributions simultaneously with the distribution of computation, using the first-touch page placement algorithm. The results of this work corroborate the belief that OpenMP and shared-memory programming models in general can scale well on tightly-coupled NUMA architectures without requiring significant extensions or mixtures of shared-memory with other forms of parallelism, such as data-parallel, SPMD, message-passing and so on. Further research is required to investigate if a similar argument is valid on distributed memory architectures, such as clusters and constellations, where the abstraction of shared memory is supported by computationally costly software extensions to the operating system and complex memory coherence protocols.

ACKNOWLEDGEMENTS

We are grateful to the ECMWF and Siegfried Benkner for providing us with the irregular kernels. This work was supported by the EU TMR grant No. ERBFMGECT-950062, NSF grant No. EIA-99-75019, and the Spanish Ministry of Education grant No. TIC-98-511.

References

[1] J. Anderson, S. Amarasinghe, and M. Lam. Data and Computation Transformations for Multiprocessors. In *Proc. of the 5th ACM Symposium on Principles and Practice of Parallel Programming (PPoPP'95)*, pages 166–178, Santa Barbara, California, July 1995.

[2] S. Benkner, P. Mehrotra, J. V. Rosendale, and H. Zima. High-Level Management of Communication Schedules in HPF-like Languages. In *Proc. of the 12th ACM International Conference on Supercomputing (ICS'98)*, pages 109–116, Melbourne, Australia, July 1998.

[3] J. Bircsak, P. Craig, R. Crowell, Z. Cvetanovic, J. Harris, C. Nelson, and C. Offner. Extending OpenMP for NUMA Machines. In *Proc. of the IEEE/ACM Supercomputing'2000: High Performance Networking and Computing Conference (SC'2000)*, Dallas, Texas, Nov. 2000.

[4] R. Chandra, D. Chen, R. Cox, D. Maydan, N. Nedelijkovic, and J. Anderson. Data Distribution Support on Distributed Shared Memory Multiprocessors. In *Proc. of the 1997 ACM Conference on Programming Languages Design and Implementation (PLDI'97)*, pages 334–345, Las Vegas, Nevada, June 1997.

[5] H. P. F. Forum. High Performance FORTRAN Language Specification, Version 2.0. Technical Report CRPCTR-92225, Center for Research on Parallel Computation, Rice University, Jan. 1997.

[6] K. Gharachorloo, M. Sharma, S. Steely, and S. V. Doren. Architecture and Design of AlphaServer GS320. In *Proc. of the 9th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'IX)*, pages 13–24, Cambridge, Massachusetts, Nov. 2000.

[7] E. Hagersten and M. Koster. WildFire: A Scalable Path for SMPs. In *Proc. of the 5th International Symposium on High Performance Computer Architecture (HPCA-5)*, pages 171–181, Orlando, Florida, Jan. 1999.

[8] S. Hirandani, K. Kennedy, J. Mellor-Crummey, and A. Sethi. Advanced Compilation Techniques for Fortran D. Technical Report CRPC-TR93338, Center for Research on Parallel Computation, Rice University, Oct. 1993.

[9] J. Laudon and D. Lenoski. The SGI Origin: A ccNUMA Highly Scalable Server. In *Proceedings of the 24th International Symposium on Computer Architecture (ISCA'97)*, pages 241–251, Denver, Colorado, June 1997.

[10] D. Lenoski and W. Weber. *Scalable Shared-Memory Multiprocessing*. Morgan Kaufmann, 1995.

[11] M. Marchetti, L. Kontothanassis, R. Bianchini, and M. Scott. Using Simple Page Placement Schemes to Reduce the Cost of Cache Fills in Coherent Shared-Memory Systems. In *Proc. of the 9th IEEE International Parallel Processing Symposium (IPPS'95)*, pages 380–385, Santa Barbara, California, Apr. 1995.

[12] D. Nikolopoulos and E. Ayguadé. A Study of Transparent Implicit Data Distribution Mechanisms for OpenMP using the SPEC benchmarks. In *Proc. of the 2nd Workshop on OpenMP Applications and Tools (WOMPAT'2001), LNCS Vol. 2104*, pages 115–129, West Lafayette, Indiana, July 2001.

[13] D. Nikolopoulos, E. Ayguadé, J. Labarta, T. Papatheodorou, and C. Polychronopoulos. The Trade-Off between Implicit and Explicit Data Distribution in Shared-Memory Programming Paradigms. In *Proc. of the 15th ACM International Conference on Supercomputing (ICS'2001)*, Sorrento, Italy, June 2001.

[14] D. Nikolopoulos, T. Papatheodorou, C. Polychronopoulos, J. Labarta, and E. Ayguadé. Is Data Distribution Necessary in OpenMP ? In *Proc. of the IEEE/ACM Supercomputing'2000: High Performance Networking and Computing Conference (SC'2000)*, Dallas, Texas, Nov. 2000.

[15] D. Nikolopoulos, C. Polychronopoulos, and E. Ayguadé. Scaling Irregular Parallel Codes with Minimal Programming Effort. In *Proc. of ACM/IEEE Supercomputing'2001: High Performance Networking and Computing Conference (SC'2001)*, Denver, Colorado, Nov. 2001.

[16] OpenMP Architecture Review Board. OpenMP Fortran Application Programming Interface. Version 1.1, Nov. 1999.

[17] V. Schuster and D. Miles. Distributed OpenMP, Extensions to OpenMP for SMP Clusters. In *Proc. of the Workshop on OpenMP Applications and Tools (WOMPAT'2000)*, San Diego, California, July 2000.

[18] B. Verghese, S. Devine, A. Gupta, and M. Rosenblum. Operating System Support for Improving Data Locality on CC-NUMA Compute Servers. In *Proc. of the 7th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VII)*, pages 279–289, Cambridge, Massachusetts, Oct. 1996.

[19] P. White. IFS Documentation: Part VI, Technical and Computational Procedures. Technical Report CY21R4, European Centre for Medium-Range Forecasts, Feb. 2000.

[20] K. Wilson and B. Aglietti. Dynamic Page Placement to Improve Locality in CC-NUMA Multiprocessors for TPC-C. In *Proc. of the ACM/IEEE Supercomputing'2001: High Performance Networking and Computing Conference (SC'2001)*, Denbver, Colorado, Nov. 2001.

Figure 1: The LU code implemented with OpenMP (left) and extended with data distribution and an affinity clause (right) to optimize memory access locality in the parallel loop.

Figure 2: Block and cyclic data distribution and implications on memory access locality in LU.

Figure 3: A snippet of the LG kernel implemented in HPF.

Figure 4: Unstructured two-dimensional block distribution in LG (left) and load imbalance caused by a regular block distribution (right).

Figure 5: Data and computation distribution in LU using loop schedule reuse.

Figure 6: Implementing a two-dimensional irregular block distribution using a customized loop schedule in the IFS LG kernel.

Figure 7: Implementation of an indirect irregular distribution.

Figure 8: Execution times of LU and LG, SL and TS.

Figure 9: Histograms of memory accesses in LU.

Figure 10: Histograms of memory accesses in LG.

Figure 11: Histograms of memory accesses in SL.

Figure 12: Histograms of memory accesses in TS.

Figure 13: Padding of *zsl1* in SL to cope with false sharing.

Figure 14: Performance of loop schedule reuse in SL, after applying array reshaping and padding to alleviate false sharing.

Figure 15: Histograms of memory accesses in SL with loop schedule reuse, before and after applying padding.

```
program LU
integer n
parameter (n=problem_size)
double precision a(n,n)
do k=1,n
   do m=k+1,n
      a(m,k)=a(m,k)/a(k,k)
   end do
!$omp parallel do private(i,j)
   do j=k+1, n
      do i=k+1,n
         a(i,j)=a(i,j)-a(i,k)*a(k,j)
      enddo
   enddo
enddo
```

(a)

```
program LU
integer n
parameter (n=problem_size)
double precision a(n,n)
!$distribute (*, cyclic) ::  a
do k=1,n
   do m=k+1,n
      a(m,k)=a(m,k)/a(k,k)
   end do
!$omp parallel do private(i,j)
!$affinity(j)=(a(i,j))
   do j=k+1, n
      do i=k+1,n
         a(i,j)=a(i,j)-a(i,k)*a(k,j)
      enddo
   enddo
enddo
```

(b)

Figure 1: The LU code implemented with OpenMP (left) and extended with data distribution and an affinity clause (right) to optimize memory access locality in the parallel loop.

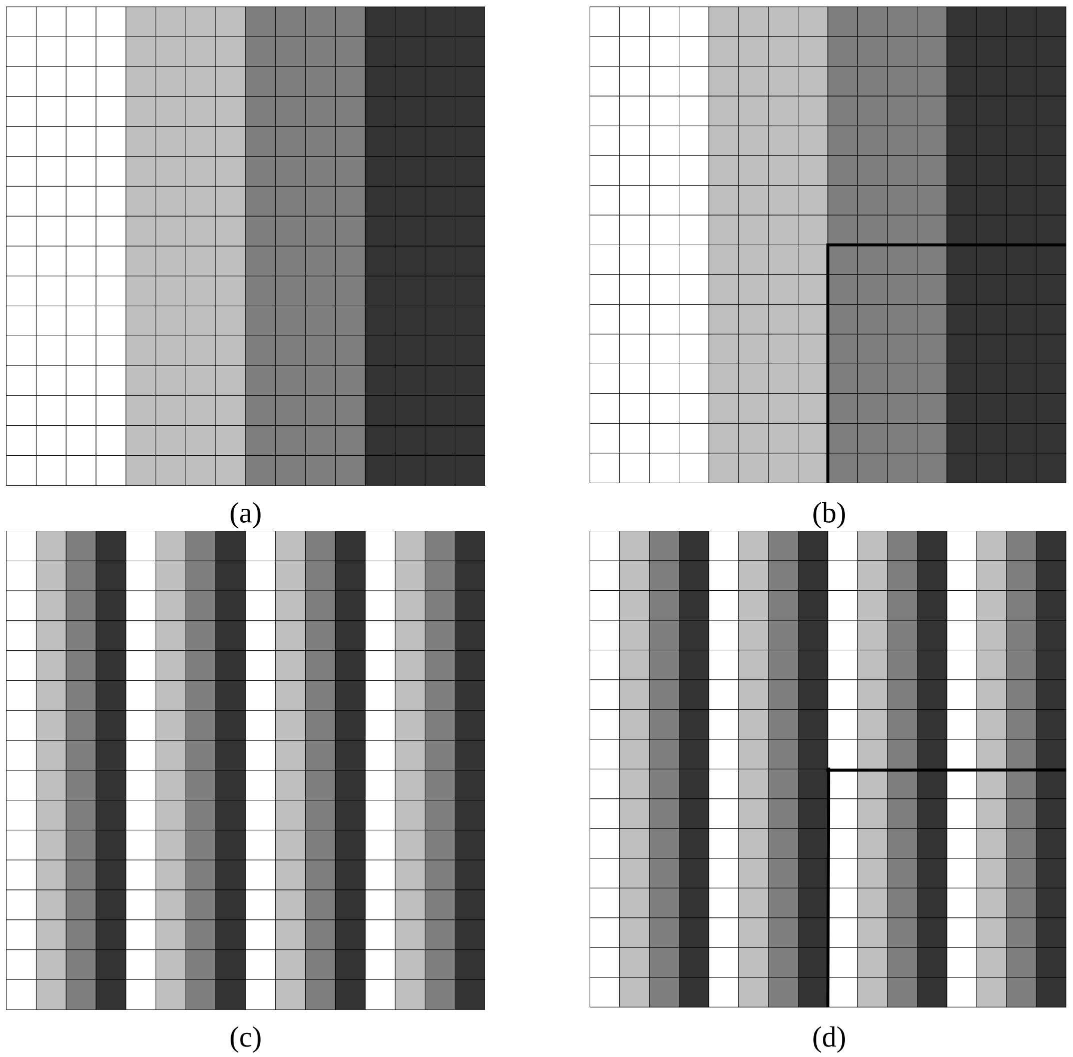
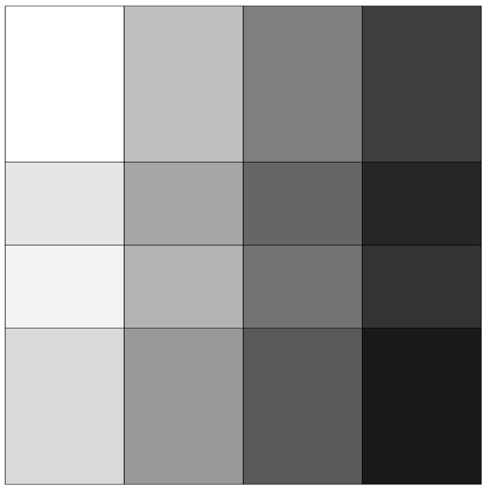

(a) (b) (c) (d)

Figure 2: Block and cyclic data distribution and implications on memory access locality in LU.

```
!hpf$ processors procs(nproc),procsab(nproca,nprocb)
!hpf$ distribute(gen_block(mapgla),indirect(mapfld0)) onto procsab::zgl
real zgl(npromag,ngt0)
!hpf$ independent,new(jfld),onhome(zgl(indl(j),:)), reuse(lreuse)
do j=1,ngptotg
   do jfld=1,ngt0
      zgl(indl(j),jfld)=zga(j,jfld)
   enddo
enddo
```

Figure 3: A snippet of the LG kernel implemented in HPF.


(a)

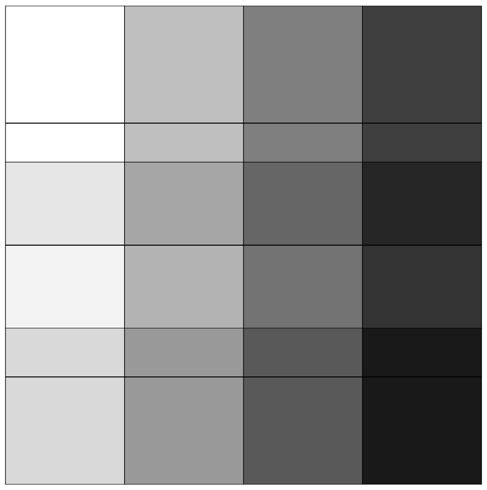

(b)

Figure 4: Unstructured two-dimensional block distribution in LG (left) and load imbalance caused by a regular block distribution (right).

```
program LU
integer n
parameter (n=problem_size)
double precision a(n,n)
integer num_procs
num_procs = omp_get_max_threads()
do k=1,n
   do m=k+1,n
      a(m,k)=a(m,k)/a(k,k)
   enddo
!$omp parallel do private(i,j,myp,jlow)
!$omp& shared(a,k)
   do myp = 0, num_procs-1
      jlow = ((k / num_procs) * num_procs) + 1 + myp
      if (myp .lt.  mod(k, num_procs))
         jlow = jlow + num_procs
      do j=jlow, n, num_procs
         do i=k+1, n
            a(i,j) = a(i,j) - a(i,k)*a(k,j)
         enddo
      enddo
   enddo
enddo
```

Figure 5: Data and computation distribution in LU using loop schedule reuse.

```
nprocs=omp_get_num_threads()
myblock_start(1) = 1
myblock_end(1) = mapgla(1)
!$omp parallel do private(p,pp)
do p=2,nprocs
   do pp=1,p-1
      myblock_start(p)=1+myblock_start+mapgla(pp)
   enddo
   myblock_end(p)=myblock_start(p)+ mapgla(p) - 1
enddo
!$omp parallel do private(iam)
do iam = 1, omp_get_num_threads()
   do j = myblock_start(iam), myblock_end(iam)
      do jfld=1+mod(iam-1,nprocb)*(ngt0/nprocb), &
      & (mod(iam-1,nprocb)+1)*(ngt0/nprocb)
         zgl(j,jfld)=zga(j,jfld)
      enddo
   enddo
enddo
```

Figure 6: Implementing a two-dimensional irregular block distribution using a customized loop schedule in the IFS LG kernel.

```
   do j=1,ngptotg
      rindl(indl(j))=j
   enddo

   !$omp parallel private(iam)
   iam=omp_get_thread_num()
   do j=1,mapgla(iam)
      myiter(iam,j)=rindl(j)
   enddo
   !$omp end parallel

   !$omp parallel private(iam)
   iam=omp_get_thread_num()
   do j=1,mapgla(iam)
      zgl(myiter(iam,j),jfld)=zga(j,jfld)
   enddo
   !$omp end parallel
```

Figure 7: Implementation of an indirect irregular distribution.

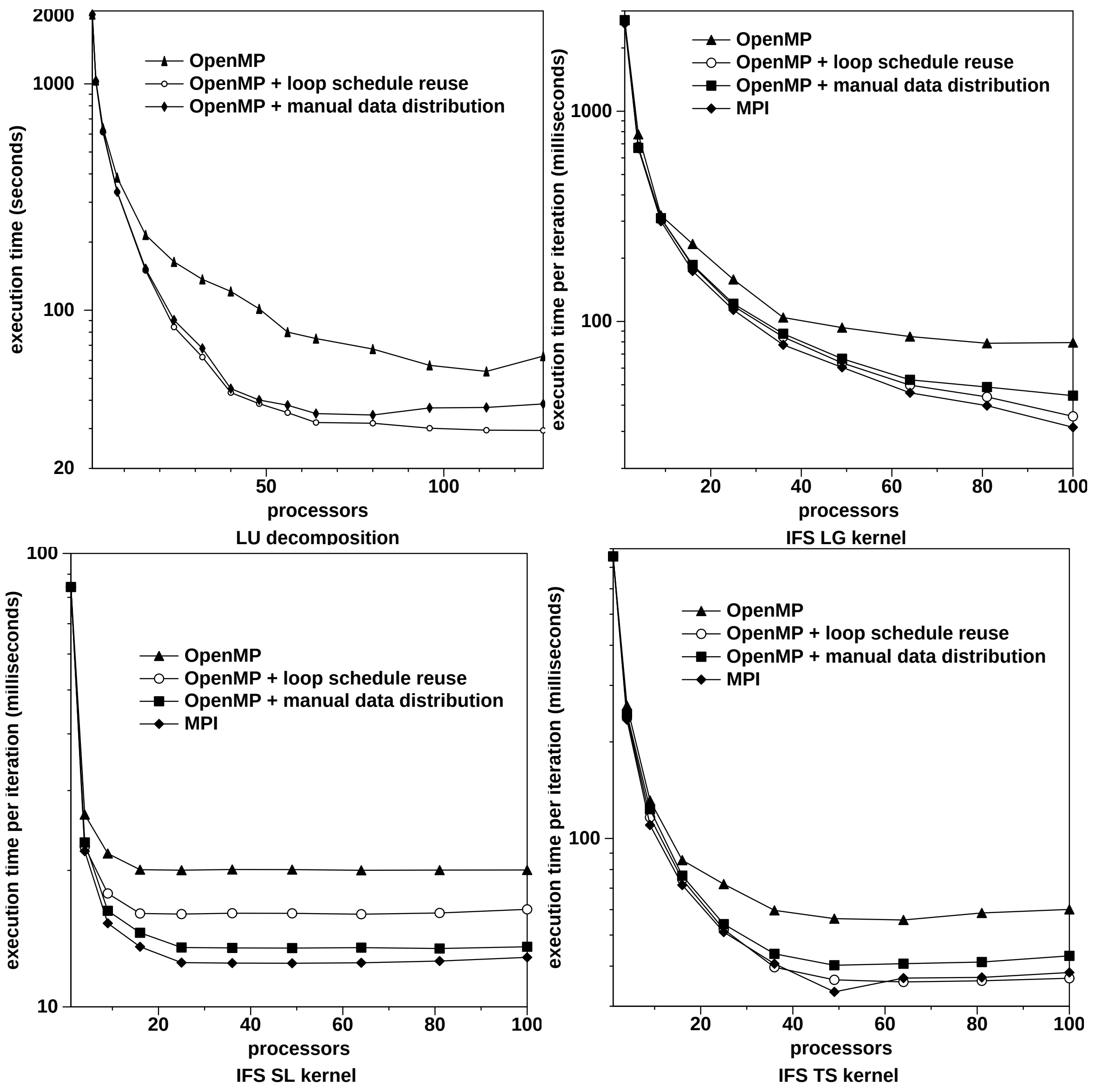


Figure 8: Execution times of LU and LG, SL and TS.

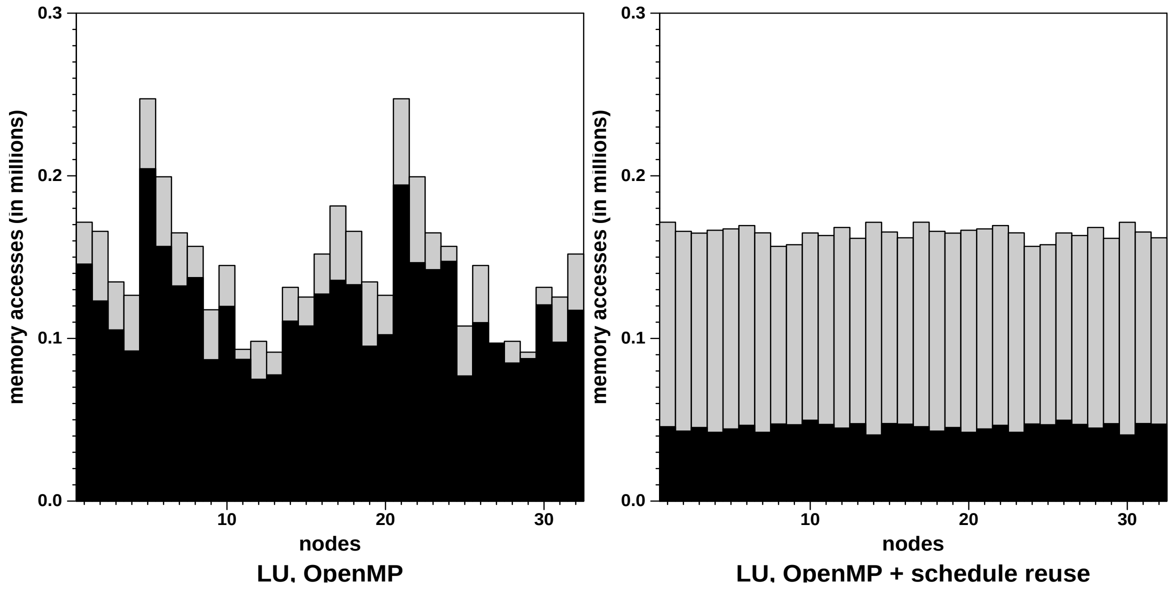


Figure 9: Histograms of memory accesses in LU.

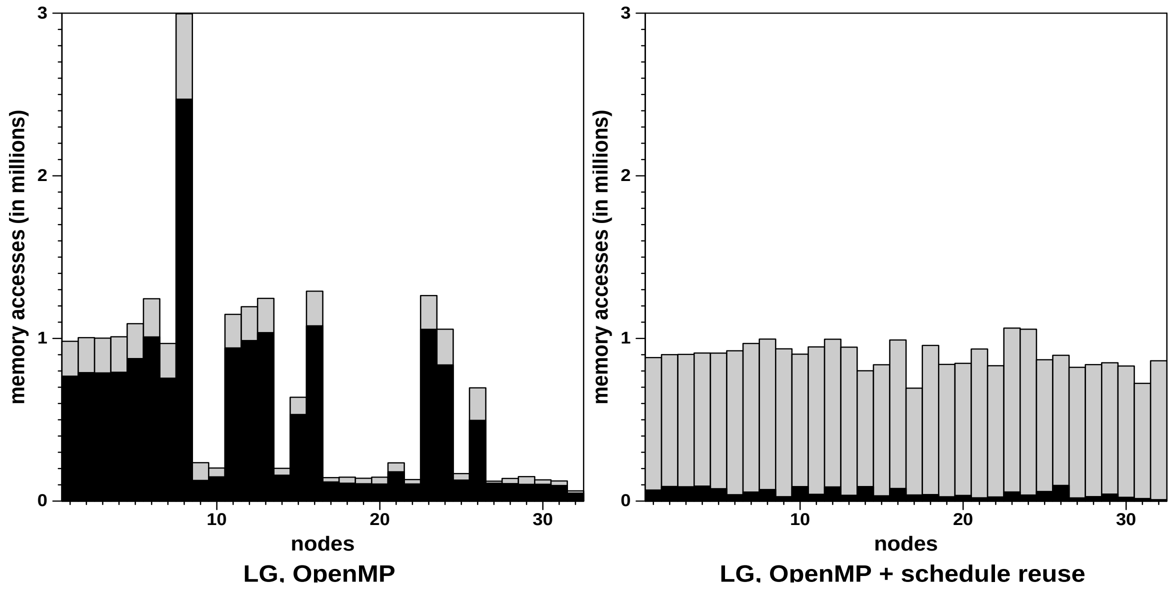


Figure 10: Histograms of memory accesses in LG.

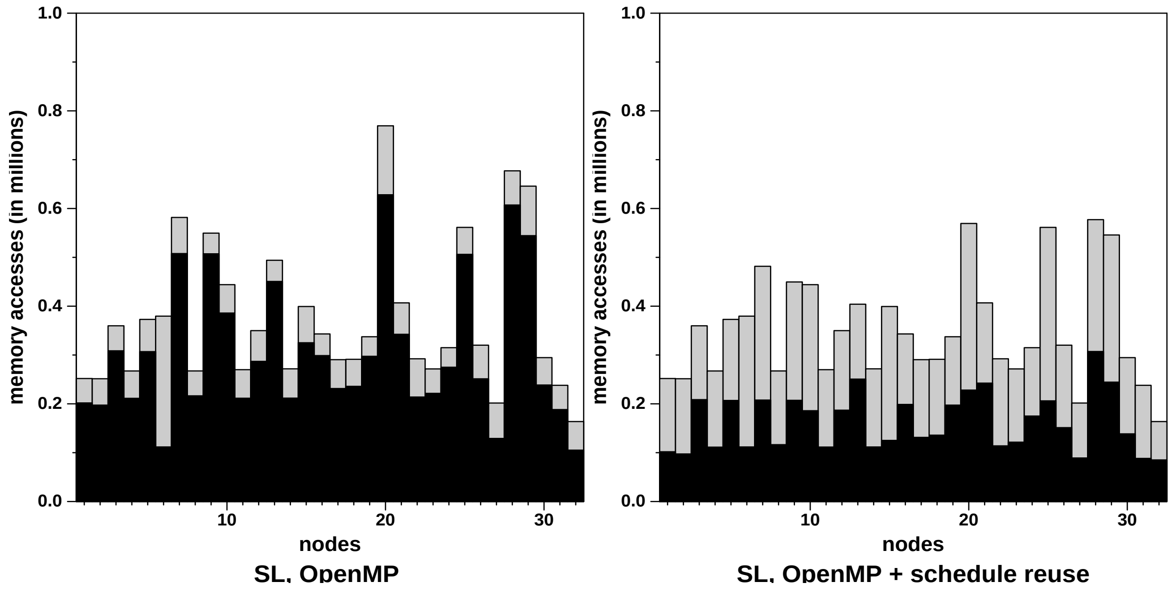


Figure 11: Histograms of memory accesses in SL.

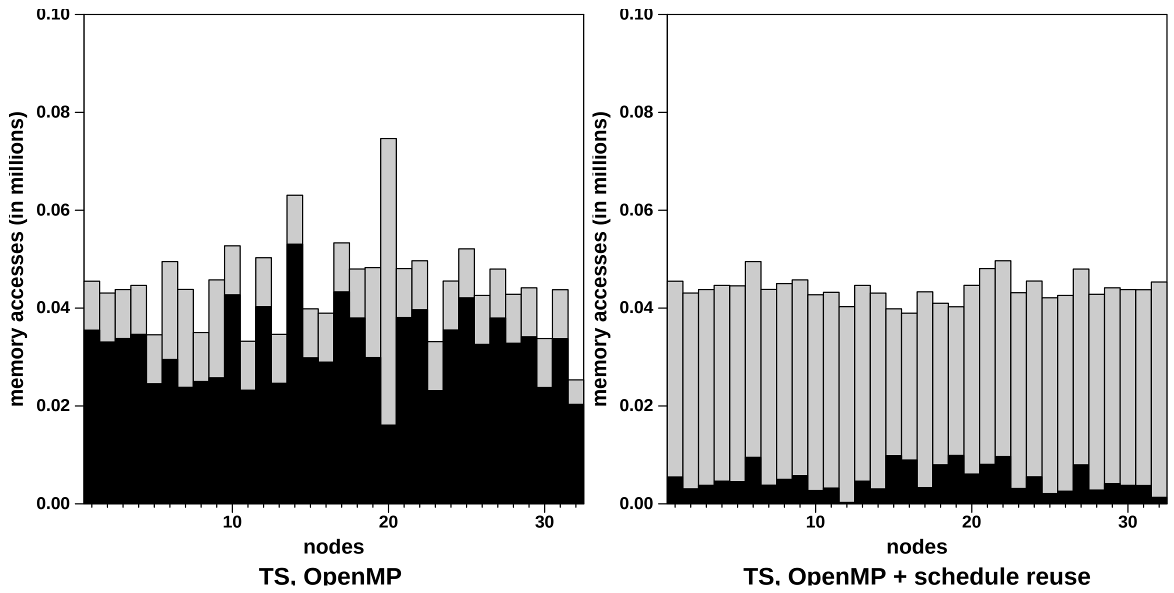


Figure 12: Histograms of memory accesses in TS.

```
Original :
real zsl1 (ngptotg, nfldslb1)

Transformed:
padded_nfldslb1= ((nfldslb1*sizeof(real)) & (~ (page_size-1)))/sizeof(real)
real zsl1 (ngptotg, padded_nfldslb1)
```

Figure 13: Padding of *zsl1* in SL to cope with false sharing.

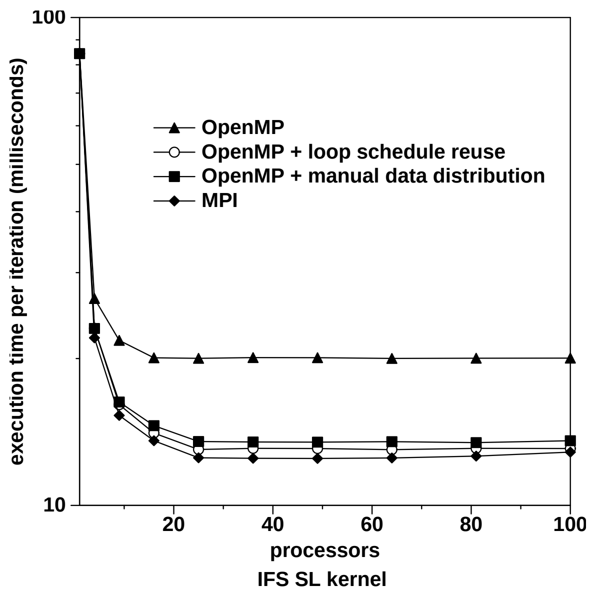
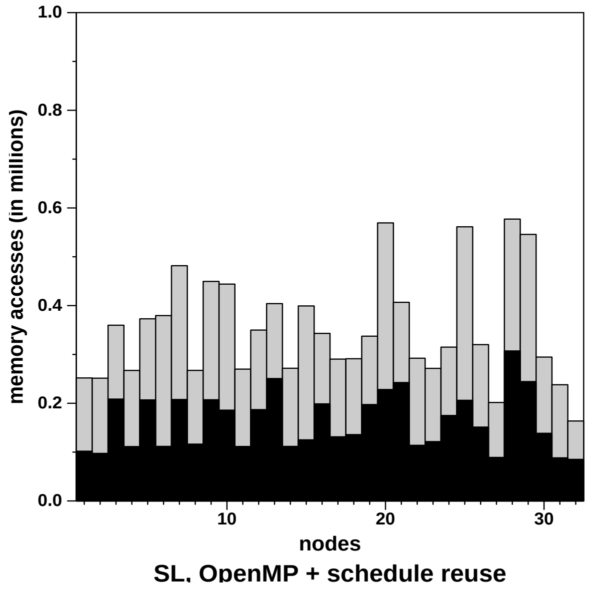


Figure 14: Performance of loop schedule reuse in SL, after applying array reshaping and padding to alleviate false sharing.



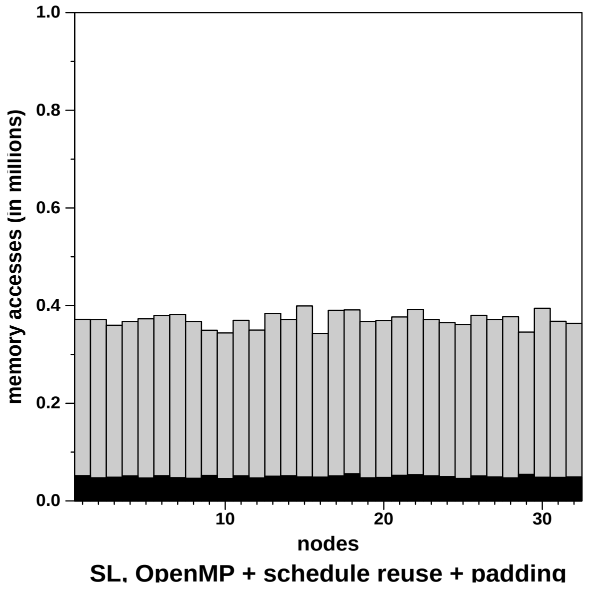


Figure 15: Histograms of memory accesses in SL with loop schedule reuse, before and after applying padding.