Exploiting Memory Affinity in OpenMP through Schedule Reuse

Dimitrios S. Nikolopoulos
Coordinated Science Laboratory
University of Illinois at Urbana-Champaign
1308 W. Main Street, MC-228
Urbana, IL, 61801, U.S.A.

Ernest Artiaga, Eduard Ayguadé and Jesús Labarta
Department d' Arquitectura de Computadors
Universitat Politecnica de Catalunya
c/Jordi Girona 1–3, 08034
Barcelona, Spain

Abstract

In this paper we explore the possibility of reusing schedules to improve the scalability of numerical codes in shared–memory architectures with non–uniform memory access. The main objective is to implicitly construct affinity links between threads and data accesses and reuse them as much as possible along the execution of the application. These links are created thorugh the definition and reuse of iteration schedules statically defined by the user or dinamically created at run time. The paper does not include a formal proposal of OpenMP extensions but includes some experiments showing the usefulness of constructing affinity links in some irregular codes.

1 Introduction

Scaling OpenMP programs on shared-memory architectures with non-uniform memory access latency is a challenging problem, partly because the OpenMP programming paradigm is oblivious of the placement of data in memory and partly because extending OpenMP to run efficiently on NUMA architectures may have undesirable implications for the portability and the design philosophy of the programming model [1, 5].

We have undertaken a project to investigate whether scaling OpenMP on NUMA architectures requires extensions to the existing OpenMP API or not. More specifically, we investigated if the OpenMP API should be extended with interfaces for explicit placement of threads and data in the nodes of a NUMA system. Such an extension would compromise the user-friendly, incremental style of parallelization offered by OpenMP and trade it for higher performance on architectures where the placement of data in memory is critical for localizing memory accesses.

Our project was quite successful in relaxing the requirement of introducing data distribution directives in OpenMP, for a broad class of parallel codes. We were able to show that in iterative parallel codes with repeating memory access patterns and statically scheduled parallel loops, the memory accesses can be almost perfectly localized using a runtime data distribution technique based on dynamic page migration [4, 3]. The prerequisite for the effectiveness of this technique, is the ability of the OpenMP runtime system to obtain an accurate snapshot of the complete memory access pattern of the program. If this snapshot is available early at runtime, data can be relocated timely enough to minimize the latency of remote memory accesses of the program and sustain high performance and good scaling.

Unfortunately, there remain some important classes of parallel codes for which our runtime data distribution engine is still unable to optimize memory access locality, either because of the constraint that the memory access pattern is not iterative, or because runtime data distribution alone is insufficient if not combined with an application-specific load distribution scheme. In this paper we focus our interest in such applications. In particular, we concern ourselves with codes where although the memory access pattern is not repeatable, it is possible to schedule the computation so that each processor reuses the same data (or a subset of it per se) along the execution time of the application. We are also concerned with iterative irregular codes, for which explicit assignment of data to threads is the only option for load balancing.

The key idea to improve the scalability of OpenMP in these two classes of codes without reverting to thread or data distribution, is the construction of implicit affinity links between threads and data, through *reusable* loop schedules. As a starting point, we use custom loop schedules that strive for better load balancing. In cases in which the same loops are executed repeatedly in the program, we identify their custom schedules as reusable, meaning that in subsequent invocations, the same processors execute the same or a subset of the iterations that they executed during the first invocation. Combined with a first-touch page placement strategy, this simple technique sustains good memory access locality, even if the memory access pattern of the program is non-repeatable or irregular. In the rest of the paper, we present motivating examples and the rationale behind reusable loop schedules and support our arguments with measurements in three irregular kernels from a weather forecasting code and a simple hand-crafted LU decomposition, all written in unmodified OpenMP.

2 Reusable custom loop schedules

Consider the simple LU decomposition shown in the upper part of Figure 1. The memory access pattern of the code changes so that any appropriate distribution of data in one iteration becomes obsolete in the next iteration. Although the code is iterative, the amount of computation performed in each iteration is progressively reduced. If data is distributed with a regular BLOCK or CYCLIC distribution, each processor will be forced to access remotely located data from the second iteration of the k loop and beyond. Therefore, neither manual, nor runtime data distribution are expected to be effective. In order to achieve both balanced load and good memory access locality simultaneously, the code should be transformed so that each iteration of the j loop is executed on the node where the j-th column of a is stored.

```
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
```

```
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 1: A simple LU code implemented with OpenMP (top) and transformed to exploit data affinity with iteration schedule reuse (bottom).

An elegant solution for localizing memory accesses is to transform the code as shown in the lower part of Figure 1. Each processor computes locally its own set of iterations to execute. Iterations are assigned to processors in a cyclic manner and 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 n=1024 and 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*, which is likely to improve load balancing. However, the actual purpose of the cyclic assignment of iterations is to have each processor reuse the data that it touches during the first iteration of the outermost *k* loop. If the program is executed with a first-touch page placement algorithm, such a transformation achieves good localization of memory accesses.

A similar situation happens in a sequence of parallel loops with slightly different lower and/or upper bounds. In this case, the same *STATIC* schedule applied to each loop may provoke a different assignment of iterations to threads. In this case it would be necessary to restructure the loops in a similar way in order to ensure that each thread reuses data accessed or computed in previous loops.

We believe that this kind of transformations can be relatively easy to apply for a restructuring compiler, without requiring a new OpenMP directive. Even if this is not the case, the transformation requires merely an extension to the *SCHEDULE* clause of the OpenMP *PARALLEL DO* directive. This extension would dictate the compiler to compute the initial iteration schedule (cyclic in this case), provide it with a name and reuse it in subsequent invocations of the same loop or different loops in a sequence.

Although the previous examples have some interesting properties with respect to memory affinity, they are still fairly simple parallel codes, which can be handled by a regular distribution of data combined with loop schedule reuse. However, this is not the case for irregular parallel codes, where the notion of irregularity refers to the data access pattern. The peculiar feature of irregular parallel codes is that the physical problem they model has some form of structural irregularity, which makes certain regions of the modeled data space more densely populated with data points than others (modeling the earth towards the poles and close to the equatorial is a simple example of a irregular data space). Irregular codes necessitate the use of application-specific load balancers, which are hard to formalize for inclusion in a flat shared-memory programming model like OpenMP. At the same time, implementing irregular data distributions is undesirable for the sake of the simplicity and the portability of OpenMP.

A viable solution for establishing thread-to-data affinity relationships in irregular OpenMP codes stems from allowing more flexibility in the loop schedulers. More specifically, it is possible to construct loop schedules such that the assignment of iterations to processors implements implicitly irregular data distributions, customized to the semantics of the application. The idea is to construct explicit maps of data to processors (reflecting the irregular data distributions) and have the compiler schedule the iterations, so that each processor touches first and then reuses the data assigned to it by the map. What makes this technique effective, is an automatic first-touch data placement algorithm, which places data (more specifically the pages that cache the data) together with the processor that touches it first during the course of the program.

Proper collocation of threads and data in an OpenMP parallel loop can be implemented transparently in the runtime system with the following procedure. The compiler identifies the data accessed during the loop and injects *mprotect()* calls to invalidate the ranges of the virtual address space that contain this data [4]. This invalidation is required to discard the –possibly inopportune— placement of data before the first execution of the loop. During the first execution of the loop, data is placed in processor memories in the order they are accessed by processors, according to the first-touch algorithm. Having this observation in mind, the programmer can assign an arbitrarily sized and structured block of data to a processor, simply by assigning the loop iterations that access this block to the same processor in the OpenMP *PARALLEL* loop.

```
!HPF$ PROCESSORS PROCS(NPROC),
!HPF$& PROCSAB(NRPOCA,NPROCB)
!HPF$ DISTRIBUTE(GEN_BLOCK(MAPGLA),
!HPF$& INDIRECT(MAPFLD0)) ONTO PROCSAB::ZGL
REAL ZGL(NRPOMAG,NGT0)
!HPF$ INDEPENDENT,NEW(JFLD),
!HPF$& ONHOME(ZGL(INDL(J),:)), REUSE(LREUSE)
DO J=1,NGPTOTG
     DO JFLD=1,NGT0
          ZGL(INDL(J),JFLD)=ZGA(J,JFLD)
     ENDDO
ENDDO
```

(a)

```
DO J=1,NGPTOTG
     RINDL(INDL(J))=J
ENDDO
```

(b)

```
!$OMP PARALLEL DO PRIVATE(IAM)
DO IAM=1,OMP_GET_NUM_THREADS()
     DO J=1,MAPGLA(IAM)
          MYITER(IAM,J)=RINDL(J)
     ENDDO
ENDDO
```

(c)

```
!$OMP PARALLEL DO PRIVATE(IAM)
DO IAM=1,OMP_GET_NUM_THREADS()
     DO J=1,MAPGLA(IAM)
          ZGL(MYITER(IAM,J),JFLD)=ZGA(J,JFLD)
     ENDDO
ENDDO
```

(d)

Figure 2: Implementing a generalized block distribution implicitly, by proper assignment of loop iterations to processors.

Figure 2 illustrates an example of how proper assignment of loop iterations to processors implements implicit irregular data distributions, using the first-touch page placement algorithm. The example shows an excerpt from the data transposition in the LG kernel, taken from the Integrated Forecasts System of the European Center for Medium Range Weather Forecasting [6]. The HPF implementation of the kernel distributes array *ZGL* using a generalized block distribution along its first dimension (Figure 2(a)). Generalized block distributions are used for load balancing in irregular grids. They assign variable-sized blocks to processors, to cope with structural irregularities that make certain regions of the grids more densely populated than other regions of the grids. The size of the block assigned to each processor in a generalized block distribution is defined by the elements of an array (*MAPGLA* in our example). *MAPGLA(i)* contains the size of the block assigned to processor *i*.

In order to implement the generalized block distribution by assigning iterations to processors, we identify the iterations that access the elements of the block assigned to each processor by the *GEN_BLOCK* distribution, as shown in Figure 2(b). 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)* according to the *ONHOME* clause. 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 code fragment shown in Figure 2(c). Intuitively, if an element i_1 is assigned to processor p, we first find the iteration j_1 that accesses i_1, by finding the value j_1 that satisfies $INDL(j_1) = i_1$. We then set $RINDL(i_1) = j_1$ and assign iteration j_1 to processor p by setting $MYITER(p, k) = j_1$ for some $k, 1 \leq k \leq MAPGLA(p)$. Finally, the original loop is transformed so that each processor executes its assigned set of iterations, as shown in Figure 2(d).

This procedure can be easily automated in an extension of the *SCHEDULE* clause of the OpenMP *DO* directive. 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 the *MAP* array contains the size of a contiguous chunk of iterations assigned to processor i. In the second case, element i of the *MAP* array contains the mapping of an element of a shared array to a processor, along the dimension of the array indexed by the 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.

3 Results

We present some results to demonstrate the potential of loop schedule reuse for exploiting memory affinity and substituting irregular data distributions. The results are taken from experiments on a 64-processor SGI Origin2000. The system on which we experimented has MIPS R10000 processors running at 250 MHz, with 32 Kilobytes of split L1 cache and 4 Megabytes of unified L2 cache per processor, and 12 Gigabytes of DRAM memory. The operating system is IRIX version 6.5.5. The page size for data pages is 16 Kilobytes. All experiments were conducted on a dedicated, idle system.

Figures 3 and 4 illustrate the execution times of our simple LU decomposition (performed on a 1400×1400) and the three irregular kernels from the Integrated Forecasts System (IFS) of the European Center for Medium Range Fore-

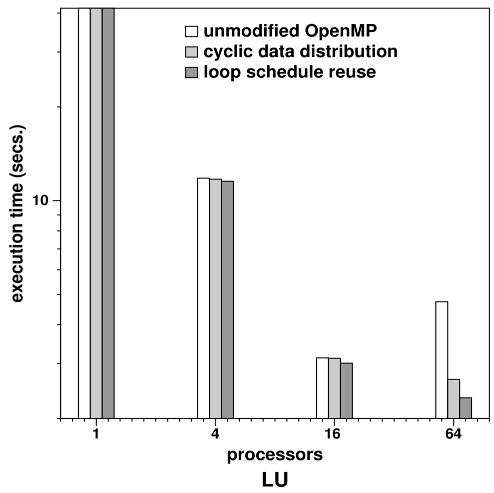

Figure 3: Execution times of LU.

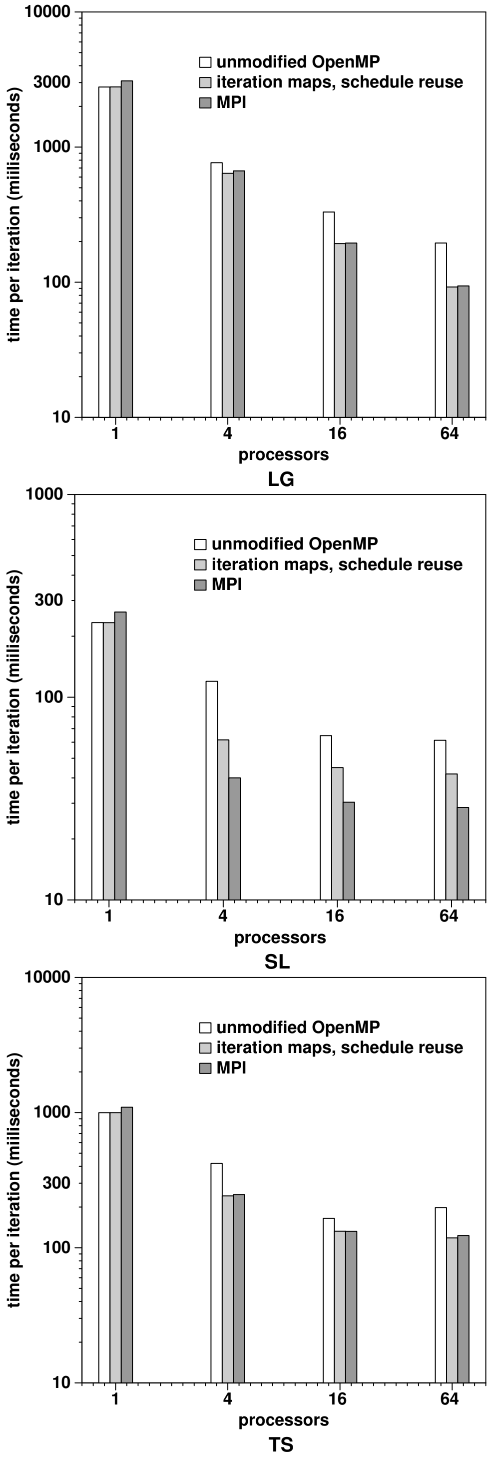

Figure 4: Execution times of the irregular kernels.

casting (ECMWF) [6], respectively. The irregular kernels perform data transpositions between the main computations phases of the IFS code. LG transforms data from the physical grid space to the Fourier grid space, TS transforms data from the Fourier space to the spectral space and vice versa, while SL computes trajectories of grid points according to the encountered winds. LG and SL use quasi-regular grids that model the atmosphere, using more points towards the equatorial and less points towards the poles. TS uses a triangular grid, which is produced from applying Legendre transforms to the Fourier space grid.

Execution times are plotted on processor scales ranging from 1 to 64 processors in even powers of two. The OpenMP implementation of LU with loop schedule reuse is compared against the unmodified OpenMP implementation and an implementation that encompasses explicit data distribution directives, provided as extensions to OpenMP by the SGI compiler [2]. The OpenMP implementation of the irregular kernels that uses iteration maps and schedule reuse is compared against the unmodified OpenMP implementation and a well-tuned MPI implementation of the same programs. The MPI implementation implements irregular data distributions, including generalized block distributions (and indirect distributions (i.e. distributions based on an indirection map between array indices and processors).

The message from the presented results is that it is possible to obtain the full benefit of thread-to-memory affinity without introducing data distribution extensions to OpenMP. This is ac-

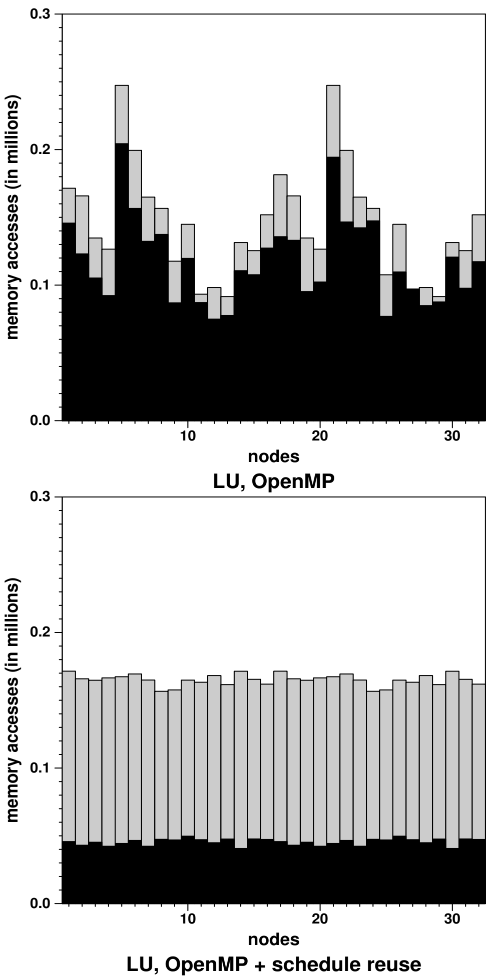

Figure 5: Histograms of memory accesses in LU.

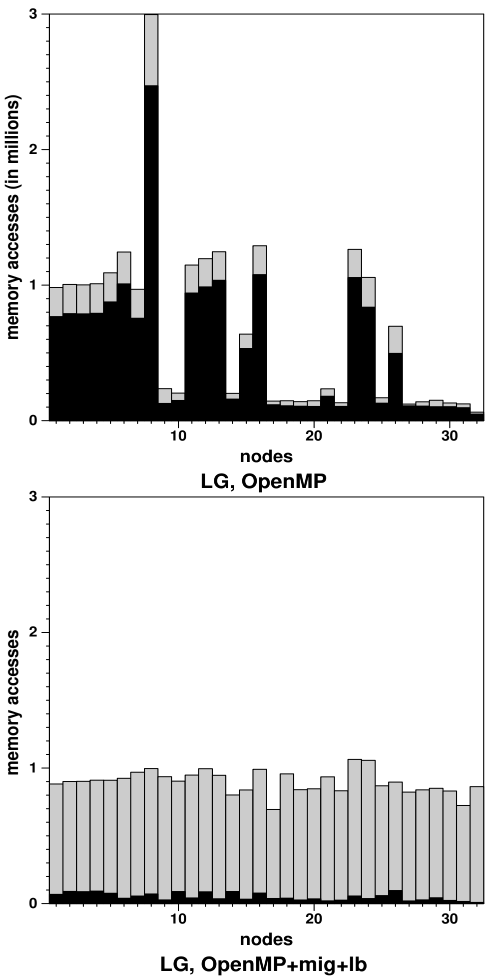

Figure 6: Histograms of memory accesses in LG.

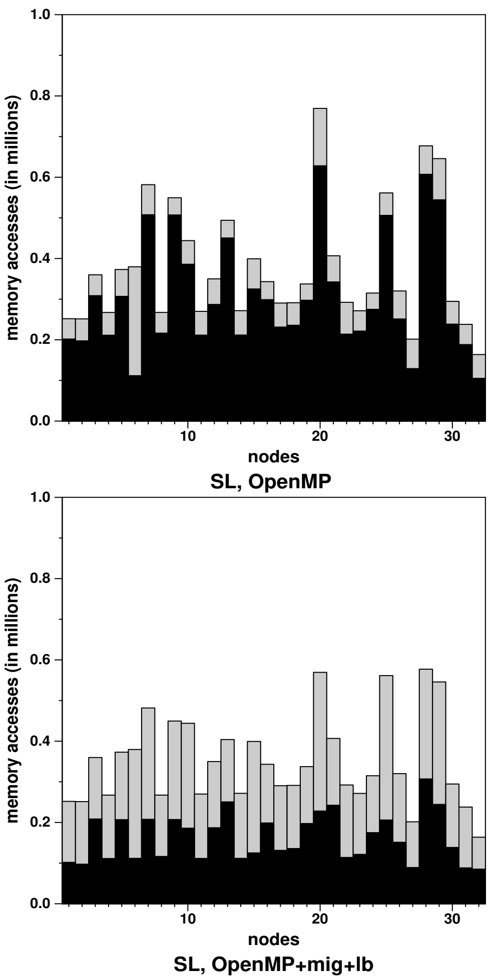

Figure 7: Histograms of memory accesses in SL.

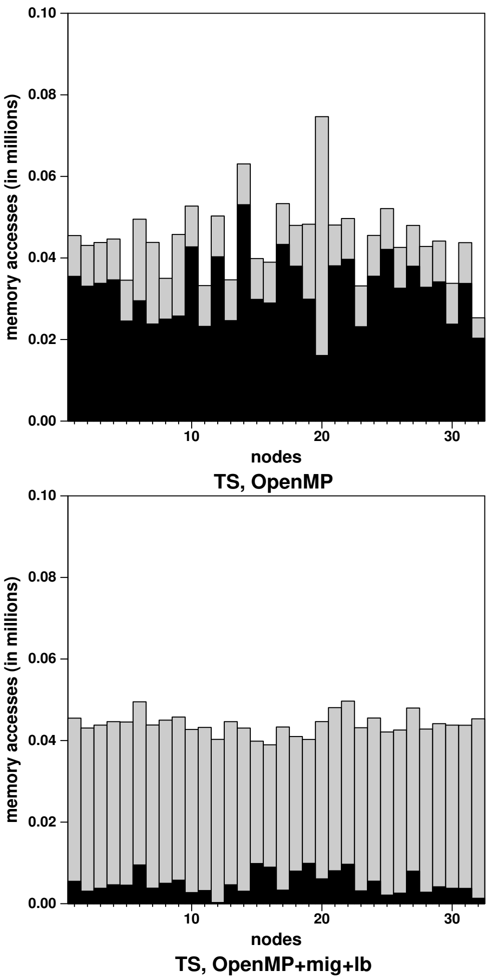

Figure 8: Histograms of memory accesses in TS.

complished by providing additional flexibility in scheduling the work-sharing constructs. Our transformations improve the scalability of the unmodified OpenMP implementations approximately by a factor of 2, while the performance of the irregular OpenMP kernels is competitive to that of MPI. The latter result 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 with MPI.

Figures 5 through 8 show histograms of memory accesses, taken from the execution of the benchmarks on 64 processors. The processors on the Origin2000 are attached to nodes with two processors per node. The processors in a node shared the memory modules of the node. The histograms show the accumulated 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). The histograms demonstrate the impact of using loop schedule reuse on memory access locality. 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. Balancing remote memory accesses 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 somewhat more unbalanced memory access pattern, but the overall number of remote memory accesses is reduced drastically. The only program in which schedule reuse has limited effectiveness in reducing and balancing remote memory accesses is SL. We suspect that false sharing is the reason for this behaviour, but more experiments are needed to track the problem to its source.

4 Conclusion

In this paper we have presented loop schedule reuse, a simple methodology for improving memory access locality in OpenMP programs. We have also shown that it is possible to use customizable loop schedules in OpenMP, to implement arbitrary data distributions using the first-touch page placement algorithm. The results of this work corroborate the belief that OpenMP can scale well on tightly-coupled NUMA architectures without requiring extensions or modifications to the programming model. Further research is required to investigate if OpenMP can scale well on loosely-coupled NUMA architectures such as clusters and constellations, using automatic data placement algorithms and appropriate program transformations . This is the primary target of our future work.

Acknowledgments

We are grateful to the ECMWF and Siegfried Benkner for providing us with the irregular kernels. Constantine Polychronopoulos and Theodore Papatheodorou contributed valuable insight in earlier stages of this research. This work was supported by the EU TMR grant No. ERBFMGECT-950062, the NSF grant No. EIA-99-75019, the Office of Naval Research grant No. N00014-96-1-0234, a research grant from the National Security Agency, a research grant from Intel Corporation and the Spanish Ministry of Education grant No. TIC-98-511.

References

[1] 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, November 2000.

[2] 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.

[3] 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.

[4] 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, November 2000.

[5] 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.

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