

Scaling Irregular Parallel Codes with Minimal Programming Effort

Dimitrios S. Nikolopoulos,
Constantine D. Polychronopoulos

Coordinated Science Laboratory
University of Illinois at Urbana-Champaign
1308 West Main Street
Urbana, IL, 61801-228

dsn@csr.d.uiuc.edu, cdp@csr.d.uiuc.edu

Eduard Ayguadé

Dept. d' Arquitectura de Computadors
Universitat Politecnica de Catalunya
c/Jordi Girona 1-3
08034, Barcelona, Spain

eduard@ac.upc.es

ABSTRACT

The long foreseen goal of parallel programming models is to scale parallel code without significant programming effort. Irregular parallel applications are a particularly challenging application domain for parallel programming models, since they require domain specific data distribution and load balancing algorithms. From a performance perspective, shared-memory models still fall short of scaling as well as message-passing models in irregular applications, although they require less coding effort. We present a simple runtime methodology for scaling irregular applications parallelized with the standard OpenMP interface. We claim that our parallelization methodology requires the minimum amount of effort from the programmer and prove experimentally that it is able to scale two highly irregular codes as well as MPI, with an order of magnitude less programming effort. This is probably the first time such a result is obtained from OpenMP, more so, by keeping the OpenMP API intact.

1. INTRODUCTION

The convergence of parallel computer architectures provides a common ground for direct quantitative and qualitative comparisons between parallel programming models. Although contemporary high-end architectures have physically distributed memory for the sake of scalability, they integrate shared-memory and message-passing in a manner that enables efficient implementations of both programming paradigms. Programmers can use a single scalable platform, such as a ccNUMA multiprocessor or a cluster of SMPs, to reason about the advantages and disadvantages of different programming models in terms of performance, expressiveness, and portability. This capability is valuable, as it lets programmers seek the long pursued *sweet spot* of parallel programming, that is, minimize the effort required to obtain a scalable program.

Direct comparisons between parallel programming models in sev-

eral application domains indicate interesting trends and trade-off's [4, 5, 16, 19]. Message-passing tends to outperform shared-memory, by giving the programmer opportunities to algorithmically minimize the communication overhead. The performance margin between the two programming models is not prohibitive for using shared-memory though. On the contrary, it has been shown that with a handful of manual optimizations for improving memory access locality and load balancing, shared-memory programming models can approximate the performance of message passing with less coding effort [19]. Moving one step further, work from the authors has shown that it is possible to use flat directive-based shared memory parallelism without explicit interfaces for thread or data placement, and yet be able to sustain performance as good as that of message-passing or data-parallel models, using algorithms for *implicit* data distribution at runtime [13, 15]. This result, although interesting in itself, has been validated only with regular and embarrassingly parallel codes.

Irregular applications are probably the most challenging applications for parallel programming models. These applications have two undesirable properties that prevent scalability, namely irregular communication patterns and load imbalance. The existing experimental evidence suggests that message passing is the programming model of choice for irregular applications, yielding performance which is typically between 50% and an order of magnitude higher than the performance of shared-memory, data-parallel or hybrid implementations of the same programs [4, 16].

Scaling irregular applications with a shared-memory programming paradigm to perform in par with implementations of the same applications with message-passing remains an open problem. There are some important steps taken in this direction, encompassing techniques such as manual data placement, data reordering and dynamic subdivision of the problem space [6, 12, 19]. Unfortunately, most, if not all, of these techniques are non-portable and require complex code and data transformations, hence significant programming effort. The lack of a systematic methodology for applying these transformations is also a concern. It is a major research challenge to scale irregular applications using flat directive-based shared-memory parallelism, without explicit interfaces for data placement, thread placement, or load balancing.

This paper addresses the aforementioned problem and contributes a simple runtime methodology for scaling iterative irregular parallel

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage, and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

SC2001 November 2001, Denver

©2001 ACM 1-58113-293-X/01/0011 \$5.00

codes, using the standard, unmodified OpenMP API. Our methodology addresses simultaneously the problems of data locality and load balancing, using runtime performance monitoring. The idea is to have the runtime system *tape* information that reflects accurately the data access pattern of the program and potential load imbalance. This information can be used for both on-line and off-line optimization. We use two simple runtime techniques, memory reference tracing combined with user-level dynamic page migration [15], and dynamic redistribution of loop iterations. Both techniques bare similarities to the well-known inspector/executor model [18], although they are faster, since they use hardware counters instead of traces collected in software. As an alternative to the automatic runtime optimizations, we present a simple scheme for implementing arbitrary irregular data distributions through proper distribution of the iterations of OpenMP parallel loops. This scheme can be exploited when the programmer can contribute some domain-specific knowledge to improve data access locality or load balancing.

We ran experiments with three highly irregular codes from the Integrated Forecasts System of the European Center for Medium-Scale Weather Forecasting [20]. We parallelized the most time consuming loops of the codes with OpenMP, spending an effort of about a couple of hours per benchmark. Our experimental evidence show that in two of the three codes, our runtime techniques enable OpenMP to perform as well as MPI, while the programming effort to reach this level of performance with OpenMP is at least an order of magnitude less than the programming effort required by MPI. We consider this result as the main contribution of this paper. To the best of our knowledge, this is the first time that an implementation of OpenMP scales as well as MPI in irregular codes, without requiring any modifications to the OpenMP API.

The rest of this paper is organized as follows: Section 2 describes the irregular kernels used in our study. Section 3 analyzes our runtime optimization methodology. Section 4 reports experimental results and Section 5 concludes the paper and discusses some directions for future work.

2. IRREGULAR APPLICATION KERNELS

We used three computational kernels (LG, SL and TS) from the Integrated Forecasting System (IFS) of the European Center for Medium-Range Weather Forecasts (ECMWF) [20]. IFS uses a spectral forecast model for predicting weather for a period of up to 10 days ahead. The kernels perform transpositions of data between the three main computational phases of IFS, namely the 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 redistributions. 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 earth) per latitude 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.

An efficient message-passing implementation of these kernels requires the identification of data points that may be accessed remotely, compilation of message send and receive lists based on the owners of remotely accessed data, and pre-computing of com-

```

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

```

Figure 1: The HPF implementation of the LG kernel.

munication schedules. A data-parallel implementation can be obtained with less coding effort, by computing and reusing communication schedules at runtime [1]. Nevertheless, a significant amount of effort is still required to identify the best data distributions for the irregular grids. Previous research has shown that the grids require generalized *BLOCK* distributions (i.e. distributions of variable sized contiguous blocks of grid points along one dimension) and *INDIRECT* distributions (i.e. arbitrary distributions of grid points according to an indirection array that maps points to processors) [2].

We provide a few details on the HPF implementation of the kernels to give the reader an idea of the programming implications and the implementation effort required to parallelize the kernels efficiently.

The LG kernel handles the transpositions of data between the physical grid point space and the Fourier space. The core of the kernel in its HPF implementation is shown in Figure 1. *NGT0* is the number of fields to be transposed (representing different layers of the atmosphere) and *NGPTOTG* is the total number of grid points. The *GEN_BLOCK* distribution along the first dimension of *ZGL* copes with the quasi-regular structure of the grid by implicitly balancing the distribution of computation between processors, through the assignment of more grid points to processors working towards the poles. The *INDIRECT* distributions along the second dimension of *ZGL* and the first dimension of *ZGA* are used to reduce communication volume, by ensuring that grid points with the same vertical dimension are mapped to the same processor. The *REUSE* clause is used to compute and subsequently reuse the communication schedules imposed by the irregular data distributions.

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, which is called a *halo*. This data structure is updated at runtime according to the winds likely to be encountered in the trajectory. The core of the kernel in its HPF implementation is shown in Figure 2. The halo computation is modeled with an *INDIRECT* distribution and a special *UPDATE_HALO* clause, required to recompute the halo and the associated communication schedules at runtime. The

```

!HPF$ PROCESSORS PROCS(NPROC)
!HPF$ DISTRIBUTE(INDIRECT(MAPGP,
!HPF$& HALO=NSLHALO,*),
!HPF$& ONTO PROCS::ZSL1
REAL ZSL1(NGPTOTG,NFLDSL1)
!HPF$ DISTRIBUTE(INDIRECT(MAPGP,*),
!HPF$& ONTO PROCS::ZSL2
REAL ZSL2(NGPTOTG,NFLDSL1)
...
DO I=1,NSTOP
...
!HPF$ UPDATE_HALO (ZSL1)
...
DO JSL=1,ISL
!HPF$ INDEPENDENT,NEW(J),REDUCTION(ZSL2)
DO JLEV=1,NFLEVG
!HPF$ INDEPENDENT,NEW(Z1,Z2,Z3,Z4,Z5,Z6)
Z1=ZSL1(NSLIND(1,J,JSL),JLEV)
Z2=ZSL1(NSLIND(2,J,JSL),JLEV)
Z3=ZSL1(NSLIND(3,J,JSL),JLEV)
Z4=ZSL1(NSLIND(4,J,JSL),JLEV)
Z5=ZSL1(NSLIND(2,J,JSL),
+ MIN(NFLEVG,JLEV+1))
Z6=ZSL1(NSLIND(3,J,JSL),MAX(1,JLEV-1))+
+ (Z1+Z2+Z3+Z4+Z5+Z6)/(6.*ISL)
ENDDO
ENDDO
ENDDO

```

Figure 2: The HPF implementation of the SL kernel.

rest of the computation is a simple averaging of neighboring grid points.

The TS kernel uses Fourier and Legendre transforms to transpose data from the Fourier space to the spectral space and backwards. As shown in Figure 3, the HPF implementation uses *GEN_BLOCK* distributions to handle the computation imbalance that stems from the quasi-regular and the triangular grids in the Fourier and the spectral space respectively.

3. PARALLELIZING IRREGULAR KERNELS WITH OPENMP

Our goal is to parallelize the irregular kernels using the standard, unmodified OpenMP interface and the minimum of programming effort. In particular, we would like to parallelize the codes simply by enclosing the most time-consuming outermost parallel loops with `!$OMP PARALLEL DO` directives¹. We impose a hard constraint on the parallelization process, that is, the programmer should analyze the data access pattern at most to the extent that parallelization is determined. The programmer should spend no effort for analyzing the locality of memory accesses and the load distribution imposed by the data access pattern. Instead, the runtime system is engaged to extract accurate information about the data access pattern and load imbalance and use this information to optimize the program in a fully- or semi-automatic manner.

We begin with a straightforward loop-based parallelization, by identifying parallel loops and selecting the loops that are coarse enough to worth multithreaded execution. A profile of the sequential execution of the program is used to identify these loops. The parallelized loops are statically scheduled by dividing evenly their iterations among processors. We then instrument the program to

¹We assume that an optimized version of the sequential code is available.

```

!HPF$ PROCESSORS PROCS(NPROCS)
!HPF$ DISTRIBUTE(GEN_BLOCK(MAPGLA,*),
!HPF$& ONTO PROCS::ZREEL
REAL ZREEL(NRPOMAG,KFIELD)
!HPF$ DISTRIBUTE(*,GEN_BLOCK(MAPFL))
!HPF$& ONTO PROCS::ZBUFL
REAL ZBUFL(KFIELD,NFTOT2G)
!HPF$ DISTRIBUTE(*,GEN_BLOCK(MAPFM))
!HPF$ ONTO PROCS::ZBUFM
REAL ZBUFM(KFIELD,NFTOT2G)
...
!HPF$ INDEPENDENT,
!HPF$& ONHOME(ZBUFL(:,NPNT0(J))),
!HPF$& NEW(JP),REUSE(LREUSE)
DO J=1,NFTOTG
DO JF=1,KFIELD
ZBUFL(JF,NPNT0(J))=ZBUFM(JF,NPNT1(J))
ZBUFL(JF,NPNT0(J)+1)=ZBUFM(JF,NPNT1(J)+1)
ENDDO
ENDDO
...
!HPF$ INDEPENDENT,NEW(JF),
!HPF$& ONHOME(ZREEL(NPNTM(J,:),:)),
!HPF$& REUSE(LREUSE)
DO J=1,NFTOTG
DO JF=1,KFIELD
ZREEL(NPNTM(J),JF)=ZBUFL(JF,NPNT0(J))
ZREEL(NPNTM(J)+1,JF)=ZBUFL(JF,NPNT0(J)+1)
ENDDO
ENDDO
...
!HPF$ INDEPENDENT,NEW(JFLD),
!HPF$& ONHOME(PREEL(J,:)), REUSE(LREUSE)
DO J=1,NGPTOTG
DO JFLD=1,KFIELD
PREEL(J,JFLD)=ZREEL(NPNTL(J),JFLD)
ENDDO
ENDDO

```

Figure 3: The HPF implementation of the TS kernel.

execute a few *probing* iterations. These iterations are used to identify two properties of the code: the exact memory access pattern and the load imbalance of parallel loops. The required instrumentation is trivial and can be automated in a compiler or preprocessor pass. The instrumented code includes calls to the runtime system for collecting memory access traces and a slightly expanded version of each parallel loop, used to collect workload information and pinpoint load imbalance. The information is collected directly from hardware counters attached to processors and memory, therefore no software bookkeeping overhead is involved. The proposed methodology is up to a certain extent similar the inspector/executor model [18], which was recently applied in the context of OpenMP for the parallelization of irregular reductions [10]. The most important difference is that our instrumentation code is much simpler and faster. It amounts to only a few calls to the runtime system for collecting information from the hardware counters using standardized interfaces.

Runtime inspection of the program is followed by a set of optimizations performed automatically by the runtime system. The runtime system applies implicit data distribution using dynamic page migration and loop iteration redistribution for the unbalanced loops. These optimizations are effectively combined for maximizing the performance gain. The optimizations are also self-evaluated at runtime and rolled back if they do not appear to improve performance.

```

...
CALL UPMLIB_INIT()
CALL UPMLIB_MEMREFCNT(U, SIZE)
CALL UPMLIB_MEMREFCNT(RHS, SIZE)
CALL UPMLIB_MEMREFCNT(FORCING, SIZE)
...
DO STEP=1, NITER
  CALL COMPUTE_RHS
  CALL X_SOLVE
  CALL Y_SOLVE
  CALL Z_SOLVE
  CALL ADD
  IF ((STEP.EQ. 1) .OR. (NUM_MIGRATIONS .GT. 0)) THEN
    CALL UPMLIB_MIGRATE_MEMORY()
  ENDIF
ENDDO

```

(a)

```

...
CALL UPMLIB_INIT()
CALL UPMLIB_MEMREFCNT(U, SIZE)
CALL UPMLIB_MEMREFCNT(RHS, SIZE)
CALL UPMLIB_MEMREFCNT(FORCING, SIZE)
...
DO STEP=1, NITER
  CALL COMPUTE_RHS
  CALL X_SOLVE
  CALL Y_SOLVE
  IF (STEP.EQ. 2) THEN
    CALL UPMLIB_RECORD()
  ELSE IF (STEP.GT. 2) THEN
    CALL UPMLIB_REPLAY()
  ENDIF
  CALL Z_SOLVE
  IF (STEP.EQ. 1) THEN
    CALL UPMLIB_MIGRATE_MEMORY()
  ELSE IF (STEP.EQ. 2) THEN
    CALL UPMLIB_RECORD()
    CALL UPMLIB_COMPARE_COUNTERS()
  ELSE
    CALL UPMLIB_UNDO()
  ENDIF
ENDDO

```

(b)

Figure 4: Using page migration for data distribution (left) and redistribution (right) in NAS BT.

3.1 Data distribution with memory reference tracing and dynamic page migration

The idea of implementing arbitrary data distributions by tracing memory references and applying an intelligent dynamic page migration algorithm was recently explored by the authors [13, 15]. The key concept is to take snapshots of the memory access trace of the program at an execution point where the trace corresponds to the exact memory reference pattern of the enclosed parallel computation. In iterative parallel codes, this snapshot can be retrieved at the end of one iteration². Using this snapshot, the runtime system can relocate pages so that each processor finds locally the data that it accesses more frequently, no matter how data is initially placed in memory by the operating system.

Runtime data distribution is applied using instrumentation of the native OpenMP code. The code inserted to the program invokes the runtime system to retrieve snapshots of the page reference counters and apply page migration algorithms using these snapshots as an indication of the memory access pattern. The snapshots are retrieved at specific points of execution, either for the entire address space, or for regions of the address space which are likely to incur frequent remote memory accesses.

The runtime data distribution algorithm ensures accuracy, timeliness and good amortization of the overhead of data movement. These three requirements are met because the runtime system retrieves snapshots of the hardware counters that reflect the memory access pattern with high accuracy, shortly after the beginning of execution. Due to the iterative structure of the codes, these snap-

²We have taken several steps to relax the constraint of optimizing only iterative codes, but the related issues are out of the scope of this paper.

shots indicate accurately which processor accesses each page more frequently throughout the execution of the program. Using these snapshots, a competitive page migration algorithm is able to move each page to the node that minimizes the number of remote memory accesses to the page. The page migration algorithm of our runtime system uses a criterion that considers the number of accesses per node, an estimation of latency per access and potential contention at memory modules. This criterion migrates pages so that the latency, rather than the number, of remote memory accesses to each page is minimized. The criterion is self-evaluated, by measuring the number and latency of remote memory accesses on a per-iteration basis. Page migrations that do not improve either of these two performance factors are rolled back by the runtime system.

Figure 4 shows how an iterative program is instrumented with calls to our runtime system (prefixed with *upplib_*) to implement runtime data distribution and redistribution algorithms. Earlier publications [14, 15] provide more details on this methodology.

Memory reference tracing and dynamic page migration is the first transparent optimization that we apply to irregular codes. The technique is expected to be effective, because the runtime system is the only entity that can infer precisely the access pattern to data blocks in memory. No regular distribution of data is appropriate for irregular codes, while sophisticated combinations of static irregular distributions —such as variable-sized block or indirect distributions— may be less effective than expected, because data is actually distributed at page rather than element-level granularity. Instead of having the programmer analyze the data access pattern and implement a possibly complex data distribution scheme, we have the runtime system infer the access pattern and implement data distribution implicitly, according to an accurate cost-effectiveness criterion. A similar approach was taken in [11], to implement adap-

```

!$OMP PARALLEL
IF (INSTRUMENT .EQ. 1) THEN
    IAM=OMP_GET_THREAD_NUM()
    CALL START_FLOP_COUNTER(IAM)
ENDIF
!$OMP DO
DO M=1,N
    LOOP BODY
END DO
IF (INSTRUMENT .EQ. 1) THEN
    CALL STOP_FLOP_COUNTER(IAM)
    INSTRUMENT = 0
ENDIF
!$OMP END PARALLEL

```

(a)

```

NPROC=OMP_GET_NUM_THREADS()
!$OMP PARALLEL
!$OMP DO
DO I=1,NPROC
    DO J=1,NPROC
        DO K=1,NC(I,J)
            M = C(I,J,K)
            LOOP BODY
        ENDDO
    ENDDO
ENDDO
!$OMP END PARALLEL

```

(b)

Figure 5: Instrumentation of OpenMP loops for detecting load imbalance and load balancing transformation.

tive data placement in software distributed shared memory systems, for applications in which compiler analysis or predetermined data distribution schemes are inadequate for optimizing memory access locality.

In our framework, the runtime system captures the irregularities of the access pattern and place pages strictly on a thread-to-data affinity basis, unless the memory access pattern is so unbalanced (in terms of memory accesses per node), that the locality-based distribution of pages introduces memory pressure and excessive contention. Although we have not quantified this problem, we plan to address it, given that it occurs frequently in irregular codes.

3.2 Loop iteration redistribution

Our runtime data distribution method is effective in localizing memory accesses but is not sufficient for balancing the workload assigned to each processor. Load imbalance is a characteristic of irregular parallel codes, because the grids used to model the problem in these codes have always some sort of physical irregularity, which makes certain regions of them densely populated by data points and other regions of them sparsely populated by data points.

The idea behind our runtime loop iteration redistribution technique is to dynamically balance an unbalanced parallel loop by normalizing the chunks of iterations assigned to each processor according to a metric of load imbalance. A similar idea was explored in [7, 17] to balance the load on networks of workstations running software distributed shared memory middleware. The main difference with our work is that the authors of [7, 17] used compiler analysis to infer the memory access pattern of the program and drive the load balancing and data locality transformations applied by the runtime system, while our framework relies solely on information collected at runtime for both the memory access pattern and the load distribution of the program.

The constraint that we pose in our automatic load balancing method is that the schedule obtained for a loop after rebalancing its load must be reusable, meaning that each processor should execute the same set of iterations every time the loop is executed. Iteration schedule reuse implies data reuse, which is critical for exploiting locality at all levels of the memory hierarchy. It is possible to use a single reusable schedule across multiple loops with the same load imbalance characteristics. Note that the requirement for a reusable iteration schedule makes it impractical to use a dynamic

work-queue based scheduling algorithm (such as forms of self-scheduling), since the non-deterministic order of synchronization events in these algorithms cannot guarantee a repeatable schedule. A workaround for this problem is to use loop schedule memoization, that is, record a dynamic loop schedule that achieves good load balance at runtime and reuse it by constructing a static schedule that assigns the same chunks of iterations and in the same order as in the recorded dynamic schedule.

We measure load imbalance on a per-loop basis as shown in Figure 5. During one probing iteration, we instrument parallel loops to measure the number of floating point operations executed by each processor. Each thread maintains a private flop counter and the difference in the number of flops executed by different processors is used as the metric of load imbalance (Figure 5(a)).

Load balancing can be an extraordinarily complex problem, even for the symmetric case of two processors. We adopt a simple algorithm. For each processor i , $i = 1 \dots P$, we compute the flop equivalent of one iteration, denoted as $FLPI(i)$. Let $FLOP(i)$ be the number of flops executed by processor i in the loop. Since the loop is initially scheduled statically, $FLPI(i) = \frac{P \cdot FLOP(i)}{N}$. The number of floating point operations executed by the processor is measured by reading the processor's hardware performance counter that tallies the number of graduated floating point instructions. The algorithm transfers the flop equivalent of one iteration from the most loaded to the least loaded processor repeatedly, until $\max FLOP(i)$ can not be further reduced. For each processor i , we record every iteration transferred to i and construct $P-1$ chunks C_{ij} , $j = 1 \dots P$, $j \neq i$, each chunk containing the iterations transferred from processor j to processor i . Chunk C_{ii} contains the iterations initially assigned to processor i by the static schedule (i.e. iterations $(i-1) \frac{N}{P} \dots i \frac{N}{P}$), minus the iterations transferred to other processors (i.e. $\bigcup C_{ji}$, $j = 1 \dots P$, $j \neq i$). The initially unbalanced loop is transformed so that each processor executes the P chunks of iterations C_{ij} , $j = 1 \dots P$, as shown in Figure 5(b). The chunks are stored as simple linear vectors of size equal to the population of each chunk (NC_{ij}) for convenience.

The instrumentation pass emits both the statically scheduled and the rebalanced version of the loop. In the rebalanced version, the vectors C_{ij} , $i \neq j$ are initially empty, while the vectors C_{ii} contain the iterations assigned to processor i by the static schedule. The load balancing algorithm runs between the first and the second

iteration of the parallel program. The selection between the two versions of the loop in subsequent iterations is done with a conditional, which is set to true if the first iteration detects that the loop is unbalanced. A boolean variable (*INSTRUMENT*) is used to deactivate the flop counters after the first iteration, in case the loop is not rebalanced by the runtime system. The drawback of this simple implementation is that it might cause code explosion, particularly if most of the code is enclosed within OpenMP *PARALLEL* regions.

3.3 Combining runtime data distribution with load balancing

The two optimizations presented previously are evaluated by the runtime system at execution time. The runtime system compares the effectiveness of loop load balancing and implicit data distribution by measuring the actual execution time of loops in two iterations and rolls back any transformation that does not improve performance.

The runtime system is also coping with the fact that the two optimizations may work in an antagonistic manner, if they are not combined effectively. This happens because the redistribution of loop iterations changes the memory access pattern of the program, thus making any previously established distribution of data obsolete. The solution we adopt is to run the probing iterations for load balancing, using different optimized data distributions for statically scheduled and dynamically rebalanced loops, and select the best performing combination based on the absolute metric of execution time.

More specifically, the runtime system optimizes memory accesses and treats load imbalance in the first three pairs of iterations of the parallel computation. In the first iteration, all parallel loops are statically scheduled and at the end of the iteration the runtime system records a snapshot of the memory access trace of the program, by reading the page reference counters. Using this snapshot, the runtime system identifies the best location for each page belonging to a distributed array, in terms of remote memory access latency. The runtime system migrates all pages that are not located in the nodes identified by the migration criterion and proceeds to the execution of the second iteration, where it checks if page migration reduces the iteration execution time. If it doesn't, the page migrations are rolled back.

In the second pair of iterations, the runtime system detects load imbalance, assuming that the runtime data distribution algorithm has already optimized page placement for maximum memory access locality. After running the load rebalancing algorithm, a fourth iteration is executed to evaluate whether load balancing improves performance. In this iteration, the runtime system measures execution time on a per-loop basis. If the load balancing transformation slows down a loop, the transformation is rolled back and the loop is scheduled statically in subsequent iterations. A similar scheme was explored in the SUIF compiler [3], to determine the number of processors that must be assigned to each parallel loop for maximizing its speedup and sequentialize loops that are too fine-grain to worth parallelization. After the loop transformations for rebalancing the load are committed and if there is at least one loop the iterations of which are redistributed, the runtime system executes a third pair of probing iterations, to optimize the placement of data according to the new loop schedules. This step ensures that the placement of data matches the memory access pattern of the rebalanced computation and combines effectively load balancing with memory access locality.

3.4 A semi-automatic approach

So far, we have described a fully automated procedure that uses a combination of transparent optimizations for data locality and load balancing. Although the memory access locality optimization technique is highly accurate, the careful reader will argue that the automatic loop redistribution scheme presented previously might suffer from inaccuracies, primarily because it does not take into account the physical properties of the data space in the programs. This problem can be circumvented if the programmer contributes some domain-specific knowledge to the parallelization procedure.

Indeed, in the irregular codes used in this study, the programmer can utilize an application-specific load balancing strategy that takes into account the irregular structure of the grids. This load balancing scheme ensures that processors working on grid partitions close to the poles receive more iterations than processors working on grid partitions close to the equatorial. The key for exploiting this domain-specific knowledge without forcing the programmer to revert to manual data distribution, is the ability to implement the application-specific load balancing scheme and the associated data mapping scheme simultaneously, using only standard OpenMP parallelization structures.

In an OpenMP *PARALLEL* loop, collocation of threads and data can be easily established by using the first-touch page placement algorithm. The first-touch algorithm places each page together with the processor that reads or writes data in the page first during the course of execution. To ensure proper collocation, the pages that contain shared data accessed during the loop must be invalidated before the execution of the loop, so that the previous location of them is discarded. This can be done transparently in the runtime system, by having the compiler identify the data accessed during the loop and using the *mprotect()* system call to invalidate ranges of the virtual address space that contain this data [15]. With this simple modification, the data accessed during the loop is distributed along with the distribution of loop iterations. 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. In the case of the irregular kernels used in this study, the loop iteration assignment emulates as accurately as possible *GEN_BLOCK* and *INDIRECT* data distributions.

Figure 6 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. In this case, *ZGL* is distributed using a *GEN_BLOCK* distribution along its first dimension. The size of each block in this distribution is defined by the elements of an array *MAPGLA*. In other words, *MAPGLA(i)* contains the number of data elements assigned to processor *i* by the distribution. In order to implement this 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 6(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)*, $i=1, \dots, P, j=1, \dots, \max(\text{MAPGLA}(i))$. The elements of this array are set with the code fragment shown in Fig-

```

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

ure 6(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 6(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. We are undergoing an effort to formalize these extensions.

4. RESULTS

We provide experimental results to illustrate the bottom line of our methodology, i.e. that our runtime techniques can scale an irregular OpenMP code to perform as well as a well-tuned message-passing counterpart, using at least an order of magnitude less programming effort.

We conducted experiments on a 64-processor SGI Origin2000, with MIPS R10000 processors running at 250 MHz. Each processor in this system has 32 Kilobytes of split L1 cache and 4 Megabytes of unified L2 cache. The system contains 12 Gigabytes of DRAM memory, distributed uniformly between 32 nodes. The operating system of the system is IRIX 6.5.5. The page size used for data pages is 16 Kilobytes. The performance metric used for comparisons is the average execution time per iteration. For the benchmarks that use our runtime system, the average is computed over

all the iterations of the program, including probing iterations.

It was our intention to compare OpenMP with both MPI and HPF. Unfortunately, this was not possible because we did not have an HPF compiler available for the Origin2000 by the time this paper was in print. As a rough indication for indirect comparisons, we report that on a Quadrics QSW CS2, the HPF+ versions of LG, TS and SL were 26%, 55% and 12 times slower than the MPI versions respectively [2].

Figure 7 shows the execution time per-iteration in the three irregular kernels. The codes require a number of processors equal to a power of two, therefore we executed them on 1, 4, 16 and 64 processors. We ran 100 iterations for each kernel, using the T63 problem size, which fits the scale of the system on which we experimented. We ran five versions of each code: a well tuned manually parallelized MPI version, originally developed by the HPF+ project consortium [2] (label MPI); a hand-parallelized OpenMP version, obtained by enclosing the outermost loops of the data transposition code with *!\$OMP PARALLEL DO* directives (label OpenMP); the OpenMP version instrumented to use implicit data distribution via dynamic page migration (label *OpenMP+mig*); the OpenMP version instrumented to use implicit data distribution together with our automatic load balancing transformation (label *OpenMP+mig+lb*); and an OpenMP version that uses implicit data distribution and load balancing through proper assignment of loops iterations to processors, as described in Section 3.4 (label *OpenMP+semiauto*). Note that the results are drawn in logarithmic scale for the sake of readability.

The scalability of LG and SL is satisfactory. LG yields a speedup of more than 32 on 64 processors. SL's speedup is to some extent limited by poor locality in the nearest neighbor computations along the halo data structure. TS does not scale beyond 16 processors, because of the granularity of its parallel loops.

The outcome of the experiments is that although straightforward parallelization with OpenMP directives in the sequential code pro-

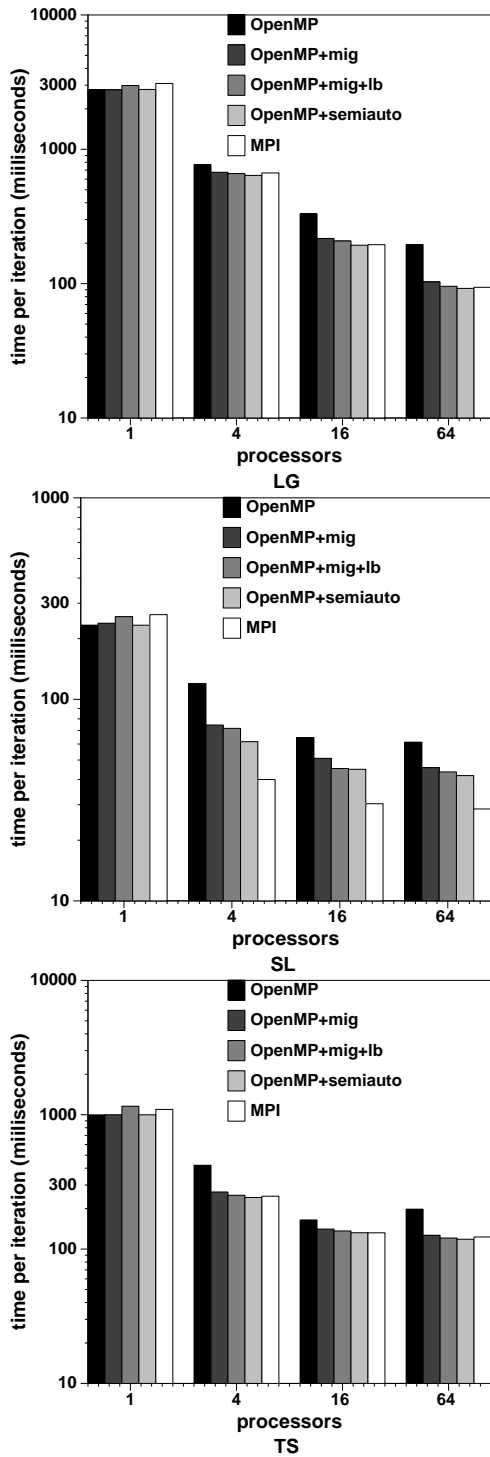


Figure 7: Execution time per iteration in LG, SL and TS with alternative OpenMP-based parallelization methods and MPI.

duces parallel code which runs almost two times slower than the corresponding MPI code, our runtime techniques are able to make the difference between the two programming models imperceptible in LG and TS, while in SL, the slowdown is reduced by almost 50%. The mediocre performance in SL is attributed to the inability of the runtime system to reduce communication traffic for the ele-

ments of the halo data structure. In the MPI version of the benchmark, the halo is constructed explicitly and each processor receives the exact set of elements required to perform its assigned nearest neighbor computation before the beginning of the parallel phase. Precomputing the communication schedule for the halo minimizes communication volume, while in the parallel phase, each processor has the required elements from its neighbors locally available. On the contrary, in the OpenMP implementation, the elements of the halo are shared between processors and their coherence is maintained by the hardware cache coherence protocol of the Origin2000. This implies that the nearest neighbor computations of each processor incur protocol-related communication traffic for maintaining coherent values of the halo elements in the caches during the parallel computation. In other words, the OpenMP implementation does not exploit an optimized precomputed communication schedule for the halo elements, but relies on cache coherence for communicating implicitly the correct values of these elements to each processor.

The difference between MPI and OpenMP appears to be mainly an issue of memory access locality. This conclusion is drawn from the fact that most of the reduction in execution time per iteration is obtained from using memory access tracing and dynamic page migration for implicit data distribution (i.e. from the *OpenMP+mig* version). Load balancing plays a less critical role, although it appears to be necessary to match the performance of MPI. The semi-automatically parallelized OpenMP versions that use application-specific load balancing perform better than the versions that use our automatic load balancing heuristic. This is also attributed to data locality, which is better if the load balancing procedure assigns contiguous rather than scattered blocks of iterations to processors.

Figures 8 through 10 explain why our page migration engine improves radically the performance of OpenMP. The charts show histograms of the number of memory accesses per-node, divided into local and remote accesses. The page migration algorithm has two important effects, which show up in the memory access traces of LG and TS. First, it nearly eliminates remote memory accesses. Second, it balances the memory accesses among nodes, thus alleviating contention. The latter effect is of particular importance for ccNUMA systems and clustered architectures, because nodes that concentrate more remote accesses are likely to suffer from contention at the memory modules and network interfaces. Unfortunately, in SL, our runtime optimization methods reduce remote memory accesses only by 50% and do not alleviate contention. One reason for this behavior is that the halo data structure introduces false sharing of data at the page level (i.e. data that reside in the same page are actively communicated between processors residing in different nodes). False-sharing tends to incur ping-pong of pages i.e. pages bounce between two or more nodes that issue approximately the same number of remote accesses to the pages. Our runtime system freezes any page likely to bounce between two nodes more than once, to compensate for the unnecessary overhead. However, the remote memory access rates to falsely shared pages are not reduced. This effect is exacerbated by the fact that the Origin2000 uses a relatively large page size (16 Kilobytes). Other reasons that may explain the performance of OpenMP in SL are under investigation.

Considering programming effort, Figure 11 shows the coding overhead (in lines) of OpenMP and MPI, compared to the length of the sequential code. We also present the overhead of the semi-automatically parallelized *OpenMP* version, since this is the one that performs closest to MPI. As expected, the additional code re-

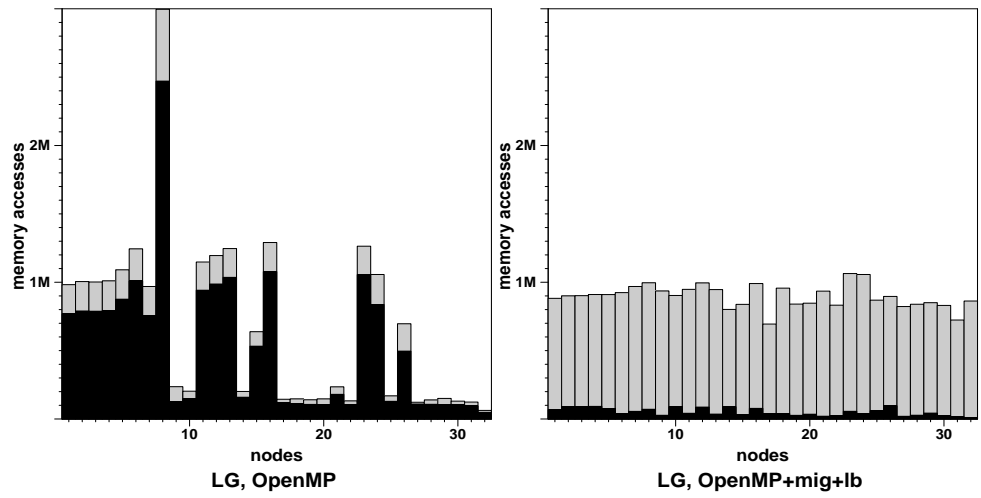


Figure 8: Per-node memory accesses of one iteration in LG, divided into local (gray part) and remote (black part) references.

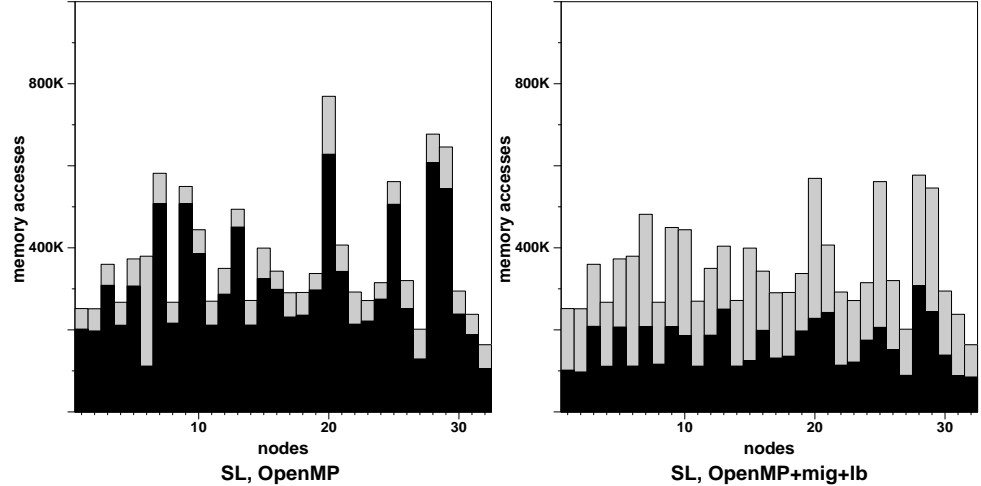


Figure 9: Per-node memory accesses of one iteration in SL, divided into local (gray part) and remote (black part) references.

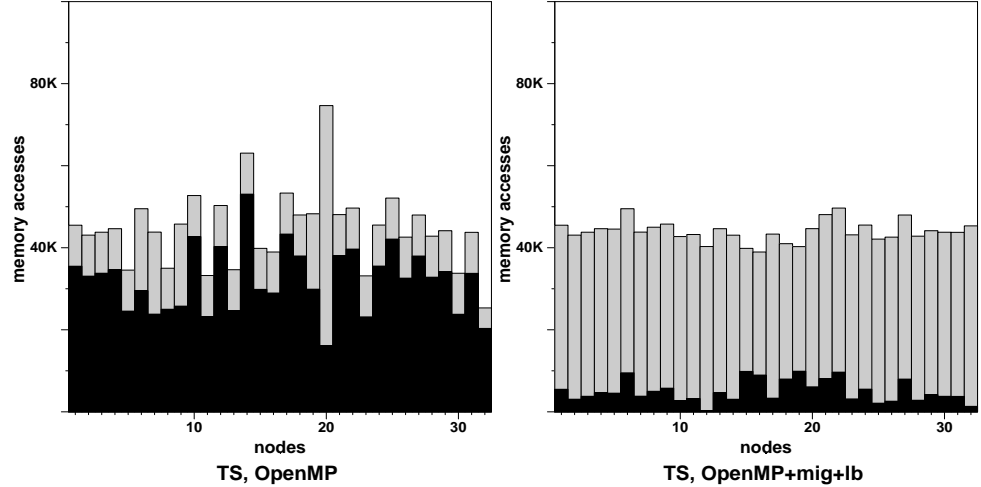


Figure 10: Per-node memory accesses of one iteration in TS, divided into local (gray part) and remote (black part) references.

	effort (lines of code)		speedup (on 64 procs.)		effort/speedup	
	MPI	OpenMP	MPI	OpenMP	MPI	OpenMP
LG	3920	262	32.92	30.2	119.1	8.7
SL	2525	227	9.20	5.57	274.5	40.8
TS	3862	155	8.89	8.44	434.42	18.36

Table 1: Effort/speedup ratio of OpenMP and MPI.

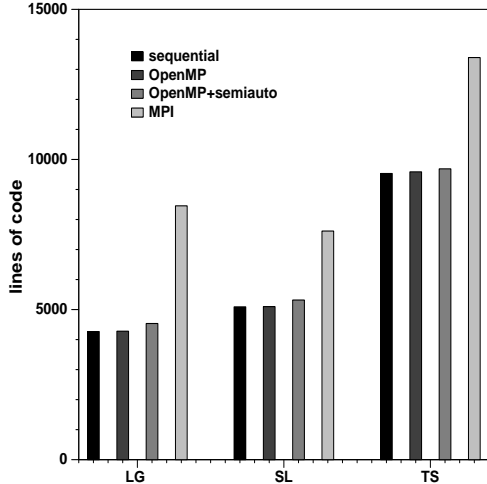


Figure 11: Coding overhead of OpenMP and MPI.

quired by OpenMP is negligible. In the worst case, it amounts to a few directives and slightly expanded versions of redistributed parallel loops. On the contrary, MPI requires about 50% more lines of code. We are aware of the fact that additional lines of code may not be a representative metric for programming effort, because it cannot capture the complexity of programming the parallel constructs. However, this limitation actually favors MPI, because the message-passing versions require a lot more sophisticated programming for communication preprocessing.

As another indication of programming effort, Table 1 reports the effort/speedup ratio for the MPI and the *OpenMP+semiauto* version. The effort/speedup metric indicates that OpenMP requires one to two orders of magnitude less effort to obtain 92–95% of the MPI speedup in LG and TS and 61% of the MPI speedup in SL.

5. CONCLUSIONS AND FUTURE WORK

We have presented a simple runtime methodology for scaling irregular OpenMP codes with minimal programming effort. The presented methodology addresses simultaneously the problems of load balancing and data locality, using inspection of the runtime behavior of the program. The runtime system collects memory reference traces and load indices per-processor and uses this information to optimize the program on-the-fly, with transparent mechanisms and transformations. Our results show that in two highly irregular codes, our runtime optimizations enable OpenMP to perform as well as MPI. This is probably the first time such a result is obtained with the standard OpenMP interface, moreover, without significant involvement from the programmer.

Porting our runtime framework to clusters of SMPs is the focus of future work. Although our general framework for runtime optimization of memory access locality and load balancing is concep-

tually applicable to clusters running software distributed shared-memory, there is a significant amount of effort that needs to be spent on engineering the runtime mechanisms required to implement memory access tracing. Since hardware page reference counters are not available in clustered architectures built up from commodity components, dynamic memory reference tracing has to be ported to software. Although remote memory accesses can be easily intercepted with relatively low overhead in the communication runtime system, local memory accesses cannot be tracked on a per-page basis without introducing unacceptable operating system overhead. We investigate workarounds for this problem. The most appealing solution appears to be the use of fine-grain multithreading and thread migration based on remote access traces, as a symmetric alternative to page migration [8].

We also explore the possibility of selecting different placement and replication/migration methods for different data in the program, according to their access pattern. More specifically, we attempt to use runtime data access information, in order to classify data into data that concentrate an insignificant amount of remote accesses and can therefore be distributed across the nodes of the cluster, data that are mostly read-shared and can be replicated across the nodes of the cluster, and irregularly accessed data that need to be kept coherent using a sophisticated migration/replication mechanism derived from lazy release consistency or other similar software shared-memory protocols [9].

Acknowledgments

We are grateful to the ECMWF and Siegfried Benkner for providing us with the irregular kernels. Jesús Labarta and Theodore Papatheodorou contributed valuable insight in earlier stages of this research. This work was supported by the E.C. TMR grant No. ERBFMGECT-950062, the National Science Foundation 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.

6. REFERENCES

- [1] S. Benkner, P. Mehrotra, J. Van 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.
- [2] HPF+ Project Consortium. HPF+: Optimizing HPF for Advanced Applications. Deliverable 1.2c, Final Evaluation Report. <http://www.par.univie.ac.at/project/hpf+>, April 1998.
- [3] M. Hall and M. Martonosi. Adaptive Parallelism in Compiler-Parallelized Code. In *Proc. of the Second SUIF Compiler Workshop*, Stanford, California, August 1997.

- [4] D. Henty. Performance of Hybrid Message-Passing and Shared-Memory Parallelism for Discrete Element Modeling. In *Proc. of the IEEE/ACM Supercomputing'2000: High Performance Networking and Computing Conference (SC'2000)*, Dallas, Texas, November 2000.
- [5] J. Hoeflinger, P. Alavilli, T. Jackson, and B. Kuhn. Producing Scalable Performance with OpenMP: Experiments with two CFD Applications. *Parallel Computing*, 27(4):391–413, April 2001.
- [6] Y. Charlie Hu, A. Cox, and W. Zwaenepoel. Improving Fine-Grain Irregular Shared-Memory Benchmarks by Data Reordering. In *Proc. of the IEEE/ACM Supercomputing'2000: High Performance Networking and Computing Conference (SC'2000)*, Dallas, Texas, November 2000.
- [7] S. Ioannidis and S. Dwarkadas. Compiler and Runtime Support for Adaptive Load Balancing in Software Distributed Shared Memory Systems. In *Proc. of the 4th ACM SIGPLAN Workshop on Languages, Compilers and Runtime Systems for Parallel Computers (LCR'98)*, pages 107–122, Pittsburgh, Philadelphia, May 1998.
- [8] A. Itskovitz, A. Schuster, and L. Shalev. Thread Scheduling in Distributed Shared Memory Systems. *The Journal of Systems and Software*, 42(1):71–87, January 1998.
- [9] P. Keleher, A. Cox, and W. Zwaenepoel. Lazy Release Consistency for Software Distributed Shared Memory. In *Proc. of the 19th International Symposium on Computer Architecture (ISCA'92)*, pages 13–21, Queensland, Australia, May 1992.
- [10] J. Labarta, E. Ayguadé, J. Oliver, and D. Henty. New OpenMP Directives for Irregular Data Access Loops. In *Proc. of the Second European Workshop on OpenMP*, Edinburgh, Scotland, September 2000.
- [11] D. Lowenthal and G. Andrews. An Adaptive Approach to Data Placement. In *Proc. of the 10th International Parallel Processing Symposium (IPPS'96)*, Honolulu, Hawaii, April 1996.
- [12] J. Mellor-Crummey, D. Whalley, and K. Kennedy. Improving Memory Hierarchy Performance for Irregular Applications. In *Proc. of the 13th ACM International Conference on Supercomputing (ICS'99)*, pages 425–433, Rhodes, Greece, 1999.
- [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)*, pages 23–37, Sorrento, Italy, June 2001.
- [14] D. Nikolopoulos, T. Papatheodorou, C. Polychronopoulos, J. Labarta, and E. Ayguadé. A Case for User-Level Dynamic Page Migration. In *Proc. of the 14th ACM International Conference on Supercomputing (ICS'2000)*, pages 119–130, Santa Fe, New Mexico, May 2000.
- [15] 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.
- [16] L. Oliker and R. Biswas. Parallelization of a Dynamic Unstructured Application using Three Leading Paradigms. In *Proc. of the IEEE/ACM Supercomputing'99: High Performance Networking and Computing Conference (SC'99)*, Portland, Oregon, November 1999.
- [17] U. Rencuzogullari and S. Dwarkadas. Dynamic Adaptation to Available Resource for Parallel Computing on an Autonomous Network of Workstations. In *Proc. of the 8th ACM Symposium on Principles and Practice of Parallel Programming (PPoPP'01)*, Snowbird, Utah, June 2001.
- [18] J. Saltz, R. Mirchandaney, and D. Baxter. Runtime Parallelization and Scheduling of Loops. In *Proc. of the 1st ACM Symposium on Parallel Algorithms and Architectures*, pages 303–312, Santa Fe, New Mexico, June 1989.
- [19] H. Shan, J. P. Singh, R. Biswas, and L. Oliker. A Comparison of Three Programming Models for Adaptive Applications on the Origin2000. In *Proc. of the IEEE/ACM Supercomputing'2000: High Performance Networking and Computing Conference (SC'2000)*, Dallas, Texas, November 2000.
- [20] P. White. IFS Documentation: Part VI, Technical and Computational Procedures. Technical Report CY21R4, European Center for Medium-Range Forecasts, February 2000.