Asynchronous Execution of OpenMP Code

Tien-hsiung Weng and Barbara Chapman

Computer Science Department
University of Houston
Houston, Texas 77024-3010
{thweng,chapman}@cs.uh.edu

Abstract. This paper presents the transformation of OpenMP source code to a Macro-Task Graph, an internal representation of the parallel program as a collection of tasks, which later can be asynchronously scheduled for out-of-order execution and optimized for locality reuse. The transformation is based on array region analysis. We also show the potential benefits of targeting OpenMP code to a macro-task graph, instead of directly generating a multi-threaded program. We show experimental results for a Jacobi kernel and part of the POP code in OpenMP and compiled traditionally versus macro-dataflow execution model using the SMARTS runtime system on SGI Origin 2000.

1 Introduction

OpenMP [1] is a popular parallel programming interface for medium scale high performance applications on shared memory platforms. Strong points are its ability to support incremental parallelization, portability, and ease of use. However, there are several obstacles to scaling an OpenMP code to hundreds or thousands of processors, in particular latency of remote memory access, poor cache memory reuse, large numbers of barriers and false sharing of data in cache. However, it is up to the user to ensure that performance does not suffer as a result. Good data locality is generally needed to overcome these performance problems. An additional potential cause of performance degradation on ccNUMA (cache coherent Non-Uniform Memory Access) systems is that a thread may need to access a remote memory location; despite substantial progress in interconnection systems, this still incurs significantly higher latency than accesses to local memory, and the cost may be exacerbated by network contention. If data and threads are unfavorably mapped, cache lines may ping-pong between locations. However, OpenMP provides few features for managing data locality. In particular, it does not enable the explicit binding of parallel loop iterations to a processor executing a given thread; such a binding might allow the distribution of work such that threads can reuse data.

The method provided for enforcing locality is to use private variables. However, systematically applied privatization may require a good deal of programming effort, akin to writing an MPI program. Several approaches have been proposed to provide performance with modest programming effort, including extending OpenMP by data distribution directives and directives to link parallel loop iterations with the node on which specified data is stored [2,3], first touch memory allocation, dynamic page migration by the operating system [3], user-level page migration [4], a combination of OpenMP and MPI [5], and compiler optimization [6]. Programming using data distribution directives has the potential to help the compiler ensure data locality; however, it means that programmers should be aware of the pattern in which threads will access data across large program regions. Hence the programming task remains more complex than with plain OpenMP. Moreover, there is no agreed standard for specifying data distributions in OpenMP; existing ccNUMA compilers have implemented their own data distribution directives. Thus, when they are used, performance may well be improved, but portability will be sacrificed. So OpenMP users may be forced to choose between simplicity and performance.

Our approach is to put work into compiler optimization and to use a runtime system that can better exploit data locality, provide higher degrees of parallelism, and reduce synchronization overheads. We target OpenMP to a data flow execution model realized using the SMARTS runtime system developed at Los Alamos National Laboratory.For this effort, we first evaluated SMARTS as a runtime system for OpenMP on SMPs and ccNUMA systems and then developed a Fortran API for SMARTS, which serves to pass information from compiler to runtime system. In the compiler analysis, we transform OpenMP to a macro-task graph. A macro-task graph is an intermediate representation (IR) of the OpenMP program where each node represents either a task or sub-iteration of a loop, called an *Iterate*, that may be assigned to a processor to be executed according to the precedence constraints represented by arcs of the graph. The assignment of a node to a processor can be permanent or hint. An arc represents a data dependence relation between a pair of nodes. The construction of this IR is based heavily on array region analysis. It describes or summarizes sub-regions of array that are accessed by a section of code and is commonly used for interprocedural side effect analysis, data dependence testing, array privatization, locality improvement, as well as communication optimization. It is applied to analyze the access patterns of the parallel program in order to make decisions on scheduling of the computation so that each processor may reuse data. However, it can be expensive to apply this testing between two call statements or between two different larger regions of code.

2 The SMARTS Runtime System and Our API

SMARTS [7](Shared-Memory Asynchronous Runtime System) was developed at Los Alamos National Laboratory to support both task and data parallelism. Written in Object-Oriented C++, it is originally designed to be the runtime system for POOMA (Parallel Object-Oriented Method and Application). The novel aspect is its macro-dataflow approach [8]. SMARTS supports the efficient execution of code, for which both the data and the work has been decomposed, on shared memory multiprocessors and ccNUMA systems by applying the data flow concept of exploiting temporal locality. It relies upon a prior partition of the original loop iteration space into so-called iterates, and a partition of the arrays in the program into data-objects. The array partitions defined and used in each iterate must be determined by the compiler. Based upon the data accesses, a task graph is constructed that captures the dependences between iterates. When an iterate requires data that is (partially) created by another iterate, the original execution order must be preserved; similarly, anti- and output dependences between iterates impose an ordering on their execution. So long as these dependences are respected, iterates that are not involved in such relationships can be scheduled concurrently for out-of-order execution. Note that the execution behavior depends on both the data and the work decomposition. The decomposition of an entire loop into many sets of iterations may result in having many independent iterates available for execution at any given time. To reduce the overheads associated with thread creation, SMARTS creates threads at the beginning of the program and carries out a join at the end of the program.

A pool of parallel tasks is implemented by multiple work queues to avoid contention on a shared queue and to promote cache locality of the task queues. When the workload is well distributed across work queues, contention is minimized. When a slave thread is idle, it first grabs work from its own work queue for execution; but if its queue is empty, it may steal work from another work queue. SMARTS uses the first touch policy by default. Under this policy, the process that first writes or reads a page of memory causes that page to be allocated in the node on which the process is running. Two variants of affinity scheduling have been employed: soft and hard affinity scheduling. With soft affinity or hint affinity scheduling, the user or compiler can specify where an iterate is to be bound, but the scheduling is up to SMARTS with work stealing when there is an imbalanced work load. Hard affinity scheduling means that an iterate is statically bound to a thread by the compiler without work stealing.

```
call OMP_SET_NUM_THREADS(4)
!$OMP PARALLEL
  do k=0, ITER
!$OMP DO
    do j = 1, size-2
       do i = 1, size-2
          A(i,j)=(B(i,j-1)+ B(i,j+1)+
                 B(i-1,j)+ B(i+1,j)) /4.0
       enddo
    enddo
!$OMP ENDDO
!$OMP DO
     do n = 0, size-1
        do m = 0, size-1
          B(m,n) = A(m,n)
        enddo
     enddo
!$OMP ENDDO
  enddo
!$OMP END PARALLEL
```

Fig. 1. Standard OpenMP version of Jacobi kernel

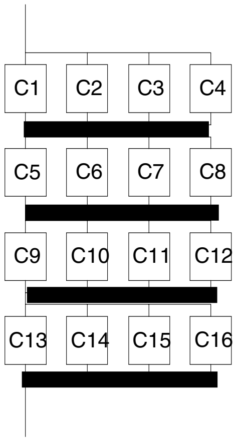

Fig. 2. Runtime execution model for Figure 1 (C stand for chunk or iterate and horizontal bold lines represent barrier)

Our strategy is to gather as much information as possible from the compiler to reduce runtime overhead. The information required includes iterate identification, read/write requests to data objects, mapping of specific iterates to thread and the type of mapping. The mapping decision is determined by generating the task graph (iterate dependence graph) and applying a heuristic scheduling algorithm at compile time.For hard affinity scheduling, each iterate is assigned a thread id and will be executed by that thread when its data becomes available.

To illustrate this, we show a simple OpenMP Jacobi kernel in Figure 1. Figures 3 and 4 show the corresponding code parallelized using calls to SMARTS via the API we have created. Figure 5 illustrates the iterate dependence graph constructed by the compiler (it may be compared with the execution model associated with the original OpenMP code in Figure 2). Based upon this graph, the compiler might assign the iterates to threads as follows: C1 to the master thread, C2 to thread 1, C3 to thread 2, and C4 to thread 3, then C5 to the master thread, C6 to thread 1, C7 to 2, C8 to 3 in order to reuse cache.

Figure 3 also illustrates the structure of the generated SMARTS code and our API. The main program is first executed by the master thread which creates additional threads, performs initializations, and passes read/write access information for iterates to the runtime system, along with iterate affinity, and iterate identification, etc. The call to SM_concurrency(n) creates an additional n-1 slave threads in which all iterates will be run. Information such as array partition information is passed via the API function SM_define_array; SM_startgen tells the scheduler that the master thread is starting a new data parallel statement. Information on read/write access of iterates to data objects is then passed to the runtime system via SM_def_readwrite.

```
call SM_concurrency(n)
call SM_define_array(info)
schedtype = hintaffinity
call SM_startgen()
do k = 1, iter
   do i1 = 1, sqit
      do j1 = 1, sqit
         call SM_def_readwrite(loop0)
         call SM_getinfo(loop0, affin, datinfo)
         call SM_handoff(0, schedtype, affin, datinfo)
      end do
   end do
   do i1 = 1, sqit
      do j1 = 1, sqit
         call SM_def_readwrite(loop1)
         ! handOff for loop 1
         call SM_handoff(1, schedtype, affin, datinfo)
      end do
   end do
end do
call SM_run()
call SM_end()
```

Fig. 3. Main program using SMARTS

Next, the API routine SM_handoff takes arguments such as iterate, type of affinity scheduling, and a thread ID that specifies the iterate mapping to slave threads and passes it to the runtime scheduler. SM_run is called to tell the scheduler that no more iterates will be handed off and the scheduler starts to work. SM_getinfo obtains information such as iterate affinity and the bounds of each iterate from the compiler.

```
subroutine fortran_loop0(datinfo)
  use shared_DATA
  call SM_compute_bound(datinfo,lb1,ub1,lb2,ub2)
  do j = lb1, ub1
     do i = lb2, ub2
        A(i,j) = (B(i,j-1)+ B(i,j+1)+
                  B(i-1,j)+ B(i+1,j)) / 4.0
     enddo
  enddo
end subroutine fortran_loop0

subroutine fortran_loop1(datinfo)
  use DATA_jacobi
  call SM_compute_bound(datinfo,lb1,ub1,lb2,ub2)
  do n = lb1,ub1
     do m = lb2, ub2
        B(m,n) = A(m,n)
     enddo
  enddo
end subroutine fortran_loop1
```

Fig. 4. Parallel loops using SMARTS

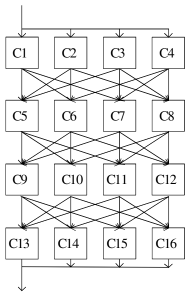

Fig. 5. Dataflow execution model associated with Figure 1

As soon as iterates have been handed off, the SMARTS runtime scheduler checks if their data are available. If so, the scheduler immediately schedules the iterate to the specified slave queue for execution. When the slave thread is idle, it grabs the iterate and the parallel loops routine associate with the iterate as Figure 4 will be invoked. In Figure 5, C1, C2, C3, and C4 correspond to subroutine fortran_loop0, and C5, C6, C7, and C8 correspond to subroutine fortran_loop1. At the end of the parallel region, SM_end is invoked to perform a join, which destroys the slave threads.

3 Transformation of OpenMP to Macro-Task Graph

Given an OpenMP source code, our aim is the compiler generation of semantically equivalent parallel code that is represented by a task graph for Macro-Dataflow computation and scheduling of the graph nodes for cc-NUMA systems. First, tasks are generated by following the semantics of OpenMP worksharing constructs. The grain size of an iterate is important. When it is too coarse, parallelism may be limited and load imbalance may be a problem; while too fine granularity will cause excessive overheads. After a suitable grain size is obtained, the compiler must determine any data dependences between pairs of nodes by array section analysis to generate the macro task graph. Where dependences exist, the compiler must further determine the reuse between the two nodes by applying array region analysis [9,10,11]. Thus a macro task graph ***G(N,E)*** consists of a set of nodes ***N*** = $\{n_1, n_2, .., n_m\}$ connected by a set of edges ***E***, each of which is denoted by e_{ij}. Each node represents a task. A task can be a region of code such as basic block, sub-iteration of loop (parallel loop), procedure, etc. The weight of node n_i, $w(n_i)$ is its execution time. Each edge e_{ij} represents a precedence constraint that must hold between node i and node j. The weight $w(e_{ij})$ is the level of reuse of data between the two end nodes in a percentage.

The task graph is generated by data dependence testing using regular section analysis. We use the example in Fig. 7 (a): A1, A2, A3, and A4 to illustrate our approach. Suppose that task 1 accesses areas A1 and A2; if we apply an immediate union, we get A'=A1∪A2, that is, a summary which also covers un-accessed regions shown as dark areas in (b). Now suppose task 2 accesses A3. The dependence relation between task 1 and task 2 is obtained by intersecting A3 and A', A''=A3∩(A1∪A2) as in (c); A'' is non-empty which causes a *false dependence*. But it is also equal to A''= (A3∪A1)∩(A3∪A2); this requires three operations, but the solution is exact as in (d). Therefore, we test whether the union is exact; if it is, we perform it immediately; otherwise, we delay the union and the two sub-regions are kept. But, we must make sure that the number of elements in the list does not exceed a threshold value.

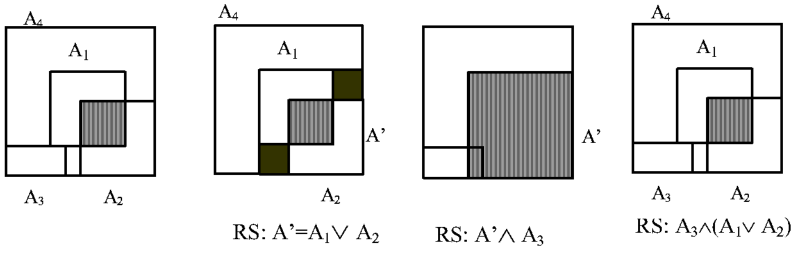

Fig. 6. Union and intersection operations

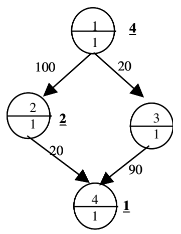

Fig. 7. Task graph

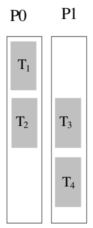

Fig. 8. Static scheduling, two processors

Fig. 8 shows the schedule reuse for the task graph in Fig. 7. We assume that first touch policy is applied. Task 2 is scheduled to p0 for data locality since its level of reuse is the highest of task 1's immediate successors. Task 3 can be scheduled on p1. Finally, task 4 is scheduled to p1 since the incoming edge from its parents task 3 indicates maximum reuse. The first touch policy can be used to obtain good memory access locality [14] even with a program that has irregular memory access patterns. By analyzing the memory access patterns in the program during graph generation, we can schedule tasks for reuse.

4 The Benefits of a Dataflow Execution Model for OpenMP

Although there is little potential for speedup with our Jacobi code, the above example already hints at the performance improvement possible if SMARTS is used as a runtime system for OpenMP. Specific benefits of targeting OpenMP to the dataflow execution model include cross-loop out-of-order execution, temporal data locality, and reduction of synchronization overheads.

4.1 Reduction of Synchronization Overheads

OpenMP employs a fork-join programming model, in which a master thread executes the sequential regions of the program. When a parallel region is encountered, it forks additional worker threads to share the work. The model is both flexible and simple, but it may incur significant synchronization overheads. In Figure 1, the first parallel DO construct requires a barrier to maintain program correctness since there is a true dependence between the first loop and the second loop. The corresponding runtime execution schema is shown in Figure 2, where the iterations performed by each thread are assumed to be a chunk. In this case, each chunk of the second parallel DO construct (C5, C6, C7 and C8) must wait until C1, C2, C3, and C4 have completed. On the other hand, in the dataflow execution model as shown in Fig. 5, each of the parallel loops in the second loop nest only wait for three iterates of the first parallel DO construct to finish. Thus this approach can reduce the synchronization overhead.

4.2 Data Locality

It is up to the programmer to write OpenMP code that makes good use of cache and avoids false sharing of cache lines by appropriately privatizing data, possibly padding shared data and blocking loops. Unfortunately, this requires a nontrivial effort. The SMARTS runtime system uses the first touch policy and affinity scheduling with or without work stealing to support locality. Work stealing is clearly useful for load balancing. But if it is too eagerly applied, it may lead to performance degradation, due to the potential conflict with cache reuse. So we assign cache reuse a higher priority than load balancing. We schedule iterates that use the same data to the same processors.

4.3 Out-of-Order Execution

Traditional OpenMP compilation will lead to execution of one parallel construct (DO or SECTIONS) in parallel. SMARTS allows an out-of-order execution within and between parallel constructs if and only if there are no data dependencies between the parallel constructs. Therefore it is necessary to compute the data dependencies between parallel constructs by applying the appropriate array section analysis and array data flow analysis. With the standard OpenMP execution model in Fig. 8, C17 can only be executed after C13 finishes and C21 can only be executed when C17, C18, C19, and C20 have completed. A translation from OpenMP to SMARTS provides an opportunity for out-of-order execution. As illustrated in Fig. 9, under this approach C17 no longer needs to wait for C13. Further, the chunks can be reordered to permit cache reuse.

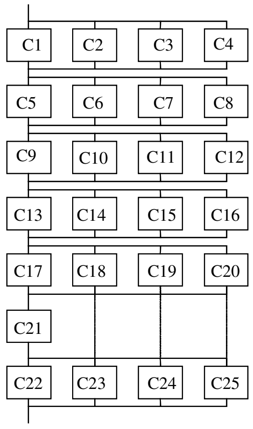

Fig. 9. Runtime execution model for OpenMP

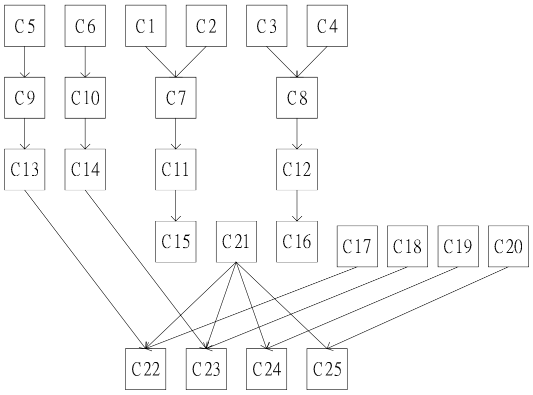

Fig. 10. Dataflow execution model associated with translated code of Figure 9

5 Experimental Results

We show results of comparisons between OpenMP and SMARTS versions of a simple application based upon the Jacobi kernel as well as one in which we translated part of a Fortran90 OpenMP version of the POP code to a corresponding parallel code based on SMARTS. POP (the Parallel Ocean Program) [13] is an explicit finite difference model developed by the Los Alamos ocean modeling research group. It makes extensive use of modules, array syntax, and WHERE constructs. Our target platform was an Origin 2000 at the National Center for Supercomputing Applications (NCSA). They were compiled with SGI's MIPSpro Fortran 90 compiler under the option –64 –Ofast –IPA and run on MIPS R10000 processors at 195 MHz with 32 Kbytes of split L1 cache, 4 Mbytes of unified L2 cache per processors and 4 Gbytes of DRAM memory. We made use of the strategies provided by SGI to obtain good performance under OpenMP. First touch allocation, where a datum is stored on the node where it is first accessed, was used in each case. We also set _DSM_MIGRATION (page migration) environment variable to OFF.

Different implementations of OpenMP on the available hardware platforms incur different overheads for OpenMP directives. Synchronization and loop scheduling overheads have been measured by [12]. The PARALLEL DO directive with REDUCTION clause has one of the worst overheads under SGI's MIPSpro f90 compiler on the SGI Origin 2000. Even though the overhead of the barrier directive itself is minor, the time spent waiting for all threads to finish could be significant when large numbers of processors are involved.

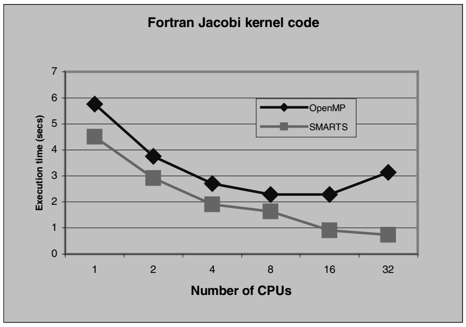

Fig. 11. Execution time for OpenMP Jacobi and translated Jacobi using SMARTS with array size of 2048x2048

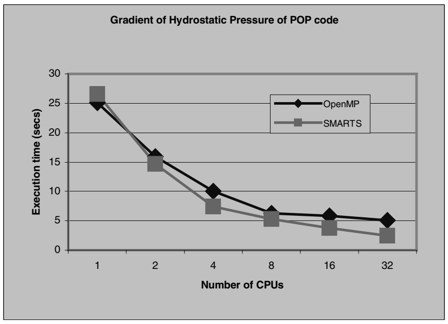

Fig. 12. Execution time for POP routine computing Gradient of Hydrostatic Pressure and translated code using SMARTS with array size of 2048x2048

We show results for a Jacobi computation based upon runs with a 2048 by 2048 double precision matrix in Fig. 10. With more processors, the barrier clearly increasingly affects the performance; from 8 processors on, it does not give good speed up. On the other hand, under data flow execution, the Jacobi program obtains better speedup up to 32 processors. When both programs are compiled with –O2, the difference between the two is much larger.

Fig. 11 illustrates our results on a routine from the POP code that computes the gradient of hydrostatic pressure at level k. For our measurement, we executed the code with an array of size 2048 by 2048. The number of data objects and read/write requests are greater than that of the Jacobi kernel. Hence, the initialization overheads incurred when passing information to the runtime system lead to a longer execution time for the translated code on one processor. This overhead is amortized by the function of the problem size and the number of iterations.

6 Conclusions

In this paper, we show that by using the SMARTS runtime system as an execution target for OpenMP, we can increase program performance by improving data locality, reducing synchronization overhead as well as exploiting out of order execution to maximize parallelism. An API has been designed to interface between Fortran and the SMARTS C++ library. We compare OpenMP with a translated code using SMARTS on two codes, clearly showing that the translated version outperforms the OpenMP code and also scales well. We can do even better by further developing the compiler to improve the analysis required for the translation. Our approach avoids the use of data distribution directives, which add complexity to the programming model and potentially destroy portability. Our experimental results consistently show a benefit from this approach.

References

1. OpenMP Architecture Review Board, Fortran 2.0 and C/C++ 1.0 Specifications. At www.openmp.org.
2. B.Chapman, P.Mehrotra, and H.Zima, Enhancing OpenMP with Features for Locality Control. Proc. ECMWF Workshop Towards Teracomputing {The Use of Parallel Processors in Meteorology". Reading, England, November 1998.
3. J. Bircsak, P. Craig, R. Crowell, J. Harris, C. A. Nelson, C. D. Offner, Extending OpenMP For NUMA Machines: The Language, WOMPAT 2000, Workshop on OpenMP Applications and Tools, San Diago, July 2000.
4. D. S. Nikolopoulos, T. S. Papatheodorou, C. D. Polychronopoulos, J. Labarta and E. Ayguade, Is Data Distribution Necessary in OpenMP ? Proc. of the IEEE/ACM Supercomputing'2000: High Performance Networking and Computing Conference, Dallas, Texas, November 2000.
5. L. A. Smith and P. Kent, Development and Performance of a Mixed OpenMP/MPI Quantum Monte Carlo Code,Proceedings of the First European Workshop on OpenMP, Lund, Sweden, Sept. 1999, pp. 6–9.
6. S. Satoh, K. Kusano, and M. Sato, Compiler Optimization Techniques for OpenMP Programs, Second European Workshop on OpenMP (EWOMP 2000), Edinburgh, September 14–15, 2000.
7. S. Vajracharya, S. Karmesin, P. Beckman, J. Crotinger, A. Malony, S. Shende, R. Oldehoeft, and Stephen Smith, Exploting Temporal Locatlity and parallelism through Vertical Execution. ICS '99.
8. S. Vajracharya, P. Beckman, S. Karmesin, K. Keahey, R. Oldehoeft, and C. Rasmussen, Programming Model for Cluster of SMP. PDPTA '99.
9. P. Havlak and K. Kennedy. An implementation of interprocedural bounded regular section analysis. IEEE Transactions on Parallel and Distributed Systems, 2(3):350–360, July 1991.
10. V. Balasundaram, K. Kennedy, A Techniques for Summarizing Data Access and its Use in Parallelism Enhancing Transformations, Proceedings of ACM SIGPLAN'89 Conference on Programming Language Design and Implementation, Jun. 1989.
11. R. Triolet, F. Irigoin, P. Feautrier, Direct Parallelization of CALL Statements, Proceedings of ACM SIGPLAN '86 Symposium on Compiler Construction, Palo Alto, CA, July 1986, pp. 176–185.
12. J.M. Bull, Measuring Synchronisation and Scheduling Overheads in OpenMP, Proceedings of the First European Workshop on OpenMP, Lund, Sweden, Sept. 1999, pp. 199–105.
13. Parallel Ocean Program, http://www.acl.lanl.gov/climate/models/pop
14. D. S. Nikolopoulos and E. Artiaga and E. Ayguadé and J. Labarta. Exploiting memory affinity in OpenMP through schedule reuse. ACM SIGARCH Computer Architecture News, 29(5):49–55, 2001.