# Exploiting Locality in the Run-Time Parallelization of Irregular Loops *

María J. Martín‡, David E. Singh†, Juan Touriño‡, and Francisco F. Rivera†
‡Dept. of Electronics and Systems, University of A Coruña, Spain
†Dept. of Electronics and Computer Science, University of Santiago de Compostela, Spain
mariam@udc.es, david@dec.usc.es, juan@udc.es, fran@dec.usc.es

## Abstract

*The goal of this work is the efficient parallel execution of loops with indirect array accesses, in order to be embedded in a parallelizing compiler framework. In this kind of loop pattern, dependences can not always be determined at compile-time as, in many cases, they involve input data that are only known at run-time and/or the access pattern is too complex to be analyzed. In this paper we propose run-time strategies for the parallelization of these loops. Our approaches focus not only on extracting parallelism among iterations of the loop, but also on exploiting data access locality to improve memory hierarchy behavior and, thus, the overall program speedup. Two strategies are proposed: one based on graph partitioning techniques and other based on a block-cyclic distribution. Experimental results show that both strategies are complementary and the choice of the best alternative depends on some features of the loop pattern.*

## 1. Introduction

The aim of this work is the parallelization of loops with indirect array accesses, which commonly appear in scientific and engineering applications: sparse matrix programs, fluid flow and molecular dynamics simulations, finite element codes... A generic case of such loop is shown in Figure 1, where arrays $I1$ and $I2$ can take any integer value. Any dependence can appear inside the loop: RAW (true data dependence), WAW (output dependence) or WAR (anti-dependence). On the one hand, if there are no loop-carried dependences, we have a DOALL loop, which can be easily parallelized to achieve the desired speedup. On the other hand, DOACROSS loops have dependences across iterations and they have to be executed in succession or, if we want to extract some parallelism, synchronization primitives should be inserted at appropriate points in the code to guarantee the order of accesses imposed by dependences.

```
DO i = 1,N
     ...
  ... = A(I1(i)) op ...
     ...
  A(I2(i)) = ...
     ...
ENDDO
```

**Figure 1. Loop with indirect array accesses**

There are, in the literature, a number of run-time approaches for the parallelization of this kind of loops. All of them focus on extracting the maximum degree of parallelism from the loop and reducing the overhead of the run-time analysis. Our approach not only maximizes parallelism, but also increases data locality to better exploit memory hierarchy in order to improve code performance. The target computer assumed throughout this paper is a CC-NUMA shared memory machine. We intend to increase cache line reuse in each processor and to reduce false sharing of cache lines, which is an important factor of performance degradation in CC-NUMA architectures.

Although for illustrative purposes the loop of Figure 1 (with one read and one write per iteration) will be used as case study, our method is a generic approach that can also be applied to loops with more than one indirect read access per iteration. One particular case of our scope is the irregular reduction pattern. However, as it frequently appears in real codes and it presents special characteristics, it is better to use efficient specific strategies for the parallel execution of this kind of pattern, such as the ones described in [6][19].

The work is organized as follows: in Section 2 we review other approaches for the parallelization of loops with indirect array accesses, stressing similarities and differences between those strategies and our proposals. Next, we describe in Section 3 our methods, based on an inspector-executor strategy. In Section 4 the performance of the strategies are analyzed and compared with another method through detailed experimental results. Section 5 presents related work; specifically, we focus on related research on data locality exploitation and improvement in memory hierarchy performance. Finally, concluding remarks are given in Section 6.

*This work was funded by the Ministry of Science and Technology of Spain and FEDER funds of the European Union (ref. TIC2001-3694-C02)

## 2. Previous works

The methods to parallelize this kind of loops can be basically classified into two major groups: speculative execution and inspector-executor strategies.

In the speculative execution methods [5] [16], the loop is executed in parallel at the same time that dependences are analyzed. If the analysis determines that the loop is not parallel, the whole computation is rolled back and the loop is executed serially. The main drawback of this strategy is the associated overhead when the loop is not parallel.

In the inspector-executor strategy, the inspector determines data dependences at run-time, and the executor runs loop iterations in parallel following the order fixed by the dependences. One of the first inspector-executor approaches was proposed by Zhu and Yew [23]: loop iterations are divided into subsets called *wavefronts*, which contain iterations that can be executed in parallel. This approach has two limitations: first, inspector and executor are tightly coupled and, thus, the inspector is not reused across invocations of the same loop, even if the dependences do not change. Second, the execution of consecutive reads to the same array entry is serialized (RAR). Midkiff and Padua [15] improve this strategy by allowing concurrent reads of the same entry. Saltz et al. [18] propose an alternative solution but restricted to the particular case of loops with no output dependences. In this strategy, inspector and executor are uncoupled. Leung and Zahorjan [12] extend the previous work to consider output dependences and propose different strategies to parallelize the inspector.

These proposals exploit iteration-level parallelism. A different approach can be found in [2], where finer grain parallelism (operation-level) is exploited also using an inspector-executor method called CYT algorithm. Dependences are analyzed in the inspector phase; if the indirection arrays do not change between invocations of the loop, then the inspector can be reused. In the executor stage, iterations are cyclically distributed among the processors, and each processor goes on with the execution as dependences are fulfilled. Operation-level synchronizations are performed to guarantee a correct execution order. The advantage of this algorithm is the extraction of a higher degree of parallelism.

The goal of all these inspector-executor approaches is to maximize parallelism and minimize the overhead of the analysis phase. A comparison between strategies based on iteration-level and operation-level parallelism is presented in [21]. The strategies were evaluated for loops with different structures, memory access patterns and computational workloads. This work shows experimentally that operation-level methods outperform iteration-level methods.

This paper describes two new operation-level algorithms: Local-CYT (LCYT from now on) and Low Overhead LCYT (LO-LCYT). They are based on the approach developed in [2], but they use different iteration distribution schemes to exploit data locality. The effectiveness of our algorithms is assessed on an SGI Origin 2000 and the results are compared with those obtained with the CYT proposal.

## 3. LCYT algorithms

Our methods are split in two phases: inspector, where memory accesses are analyzed and an iteration distribution is performed accordingly; and executor, where the assigned iterations are executed in parallel. Both phases are independent, which allows to reuse the dependence information of the inspector if the same loop is executed several times.

### 3.1. LCYT inspector phase

In this stage memory access and data dependence information is collected. The access information, which determines the iteration partition approach, is stored in a graph structure. Dependence information is stored in a table called *Ticket Table* [2]. The inspector phase consists of three parts:

*1. Construction of a graph* representing memory accesses. It is a non-directed weighted graph; both nodes and graph edges are weighted. Each node represents $m$ consecutive elements of array $A$, $m$ being the number of elements of $A$ that fit in a cache line. The weight of each node is the number of iterations that access that node for write. Moreover, each node is assigned a table which contains the indices of those iterations. The edges join nodes that are accessed in the same iteration. Each edge has a weight that corresponds to the number of times that the pair of nodes is accessed in an iteration.

*2. Graph partitioning*. The graph partitioning will result in a node distribution (and, therefore, an iteration distribution) among processors. Our aim is to partition the graph so that a good node balance is achieved and the number of edges being cut is minimum. Node balance results in load balance and cut minimization involves a decrease in the number of cache invalidations, as well as an increase in cache line reuse. Besides, as each node represents a cache line with consecutive elements of $A$, false sharing is eliminated. We have used the *pmetis* program [10] from the METIS software package to distribute the nodes among the processors according to the objectives described above. The *pmetis* partitioning algorithm is based on multilevel recursive bisection. The algorithm consists of three stages: in the first stage the size of the graph is reduced by collapsing nodes and edges (*coarsening*). Next, the smaller graph is partitioned. In the third stage the partition is successively projected back towards the original graph (*uncoarsening*).

*3. Creation of a Ticket Table* containing data dependence information: it is a table $N \times r$, $N$ being the number of loop iterations, and $r$ the number of accesses per iteration to the

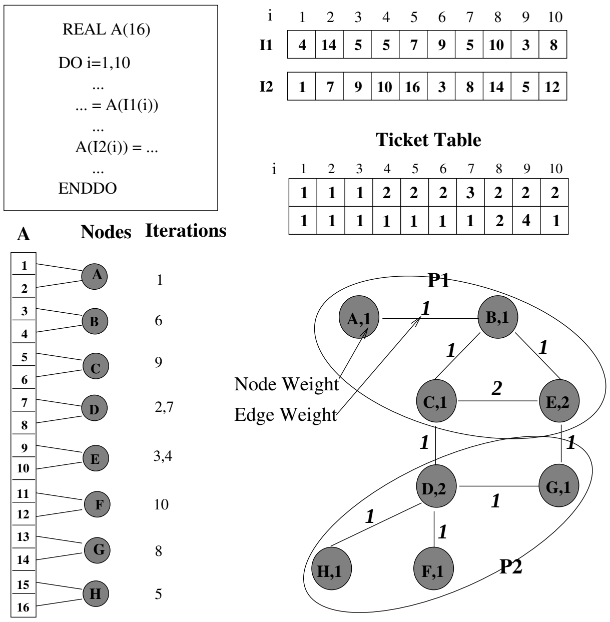


**Figure 2. Example of LCYT graph partitioning**

target array. In the loop of Figure 1 $r = 2$ because array $A$ has one write and one read access. The table stores the number of times each array entry is accessed. So, for the loop of Figure 1, $TABLE(i, 1)$ represents the number of accesses to element $A(I1(i))$ from the first up to the $i^{th}$ iteration. The creation of the Ticket Table is independent of the graph construction and partitioning, and thus these stages can be performed in parallel.

Figure 2 shows an example of the three steps of the inspector for $N = 10$, an array $A$ of 16 elements, a cache line size of 2 elements and two array accesses per loop iteration ($r = 2$). The figure shows the mapping of array entries in nodes, the generation of the graph using access information, and the graph partitioning for two processors. The corresponding Ticket Table is also depicted.

The graph partitioning is the most costly part of the LCYT inspector. This overhead strongly depends on the number of nodes ($\lceil M/m \rceil$, $M$ being the number of entries of array $A$) and can be reduced by increasing the number of cache lines of $A$ elements per node (and thus reducing the number of nodes) at the expense of load imbalance.

### 3.2. LO-LCYT inspector phase

We propose an inspector with lower overhead, named LO-LCYT, that considers a simpler iteration partitioning to optimize memory accesses (although not as good as LCYT). Unlike LCYT, only write accesses are considered for the iteration distribution. The graph construction and partitioning stages of LCYT are then replaced by the following procedure in LO-LCYT. Array $A$ is split into blocks of $m \times l$ consecutive elements of $A$, $l$ being the number of cache lines considered in the block. Blocks are cyclically assigned and each processor executes the iterations that access the elements of its blocks for write; that is, the iteration distribution is driven by a block-cyclic assignment of array $A$. The number of iterations assigned to each block are not taken into account for the distribution of blocks; nevertheless, in general, the cyclic distribution provides a good balance. This balance is worse as $l$ increases but, in contrast, data locality is better exploited. The value of $l$ is chosen as a tradeoff between both parameters. The LO-LCYT inspector overhead depends on the number of iterations $N$ and, in general, it is lower than the LCYT partitioning overhead, except for those cases in which $M << N$. As in LCYT, the iteration partitioning and the Ticket Table can be calculated in parallel because they are independent tasks.

### 3.3. Executor phase

The executor is the same for both algorithms and uses the dependence information recorded in the Ticket Table to execute, in each processor, the iterations assigned in the inspector phase. An array reference can be performed if and only if the preceding references are finished. The executor uses a shared variable (*Ready*) to count the number of times each entry of $A$ is accessed. In the course of the executor, before accessing the $x^{th}$ entry of $A$ in the $i^{th}$ iteration for read ($r = 1$) or write ($r = 2$), it is checked if $Ready(x) = TABLE(i, r)$. If not, the processor will wait until the condition is fulfilled and just then, the access is performed and $Ready(x)$ is incremented by one. All array accesses are performed in parallel except for the dependences specified in the Ticket Table. Iterations with dependences can be partially overlapped because we consider dependences between accesses instead of between iterations.

RAR occurrences are serialized, which introduces pseudo-dependences among truly concurrent iterations. Xu and Chaudhary [22] have proposed an improvement to the CYT algorithm that allows concurrent reads of the same array entry. The inclusion of this improvement in our LCYT proposals would be straightforward.

## 4. Performance evaluation

In this section, the CYT, LCYT and LO-LCYT strategies are compared experimentally. We begin with a description of the experimental conditions in Section 4.1. Our primary concern is to quantitatively assess the improvement in the executor stage by taking into account the access locality. Thus, in Section 4.2 we mainly focus on the executor evaluation in terms of the comparison of performance, cache line reuse, false sharing and load balancing. The effect of the inspector overhead on the results is also discussed.

```
REAL A(M)
DO i = 1,N
   tmp1 = A(INDEX(i*2-1))
   A(INDEX(i*2)) = tmp2
   DO j = 1,W
     dummy loop simulating useful work
   ENDDO
ENDDO
```

**Figure 3. Experimental workload**

### 4.1. Experimental conditions

The parallel performance of the irregular loop is mainly characterized by three parameters: loop size, workload cost and access pattern. In order to evaluate their influence on performance, we use the loop pattern shown in Figure 3, following the same approach taken by other authors [2] [22]; $N$ represents the problem size, the computational cost of the loop is simulated by the parameter $W$ and the access pattern is determined by array $INDEX$ and the size of array $A$.

Examples of this loop pattern appear in the solution of sparse linear systems (e.g. routines $lsol$, $ldsol$ and $ldsoll$ of the Sparskit library [17]), where the loop size and the access pattern depend on the sparse coefficient matrix. These systems are solved in a wide variety of codes: linear programming applications, process simulation, finite element and finite difference applications, optimization problems... Therefore, we have used in our experiments as indirection arrays the patterns of sparse matrices that appear in real codes. These matrices were extracted from the Harwell-Boeing (HB) collection [4]. We have also considered synthetic access patterns to cover a wider range of cases. The patterns (5 HB and 4 synthetic) are characterized in Table 1, where $2 \times N$ is the size of the indirection array $INDEX$ and $M$ is the size of array $A$. The parameter $CP$ (Critical Path, $1 \leq CP \leq N$) is the length of the longest dependence chain in the loop and gives an estimate of how parallel the loop is (if $CP = 1$ the loop is fully parallel; if $CP = N$ it is fully serial). The rightmost column of Table 1 normalizes $CP$ with respect to the number of iterations $N$.

Two kinds of synthetic patterns were considered: uniform and non-uniform. The uniform pattern assumes all array elements have the same probability of being accessed. It is denoted as *xxx_U* in Table 1, *xxx* being the size of array $INDEX$. The non-uniform pattern (denoted as *xxx_90_10*) was generated so that 90% of references are only to 10% of array elements. This pattern reflects hot spots in memory accesses and results in longer dependence chains. In all synthetic patterns, $M = 2 \times N$.

The target machine is an SGI Origin 2000 with R10K at 250 MHz. The R10K has a 2-level cache hierarchy with L1 instruction and data caches of 32KB each, and L2 unified cache of 4MB (L2 cache line of 128 bytes). The tests were written in Fortran+OpenMP directives (MIPSpro compiler with -O3 optimization level), and executed in single-user mode. All data structures were cache aligned. In our experiments, the cost per iteration, $W$, of loop $i$ of Figure 3 can be modeled as $T(W) \approx 8 \times 10^{-5} \times (1 + W)$ ms. $W$ depends on the application; typical values range from 5 to 30 using HB matrices for the loop patterns of the aforementioned Sparskit routines that solve sparse linear systems.

**Table 1. Benchmark matrices**

| Matrix | $2 \times N$ | $M$ | $CP$ | $CP * 100/N$ |
|---|---|---|---|---|
| *gemat1* | 47368 | 4929 | 4938 | 20.85 |
| *gemat12* | 33110 | 4929 | 49 | 0.30 |
| *mbeacxc* | 49920 | 496 | 487 | 1.95 |
| *beaflw* | 53402 | 507 | 500 | 1.87 |
| *psmigr_2* | 540022 | 3140 | 2626 | 0.97 |
| *25600_U* | 25600 | 25600 | 9 | 0.07 |
| *25600_90_10* | 25600 | 25600 | 45 | 0.35 |
| *51200_U* | 51200 | 51200 | 11 | 0.04 |
| *51200_90_10* | 51200 | 51200 | 46 | 0.18 |

### 4.2. Experimental results

Figure 4 shows the reduction in the execution times by applying the LCYT and LO-LCYT algorithms as compared to the CYT method for some of the matrices. LCYT($n$) means that each node represents $n$ cache lines of $A$ elements; LO-LCYT($n$) takes blocks of size $n$ cache lines. As expected, LCYT partitioning is better for a finer node definition (that is, only one line per node). Regarding LO-LCYT, the best results are obtained, in general, for blocks of two lines, being a tradeoff between load balancing and memory reuse. It can also be observed that, as the LCYT partitioning analyzes not only write accesses, but also read accesses, it improves the results of the LO-LCYT method.

The largest reductions are obtained in loops with small computational cost per iteration ($W$) because, in this kind of loops, memory accesses to array $A$ have a greater influence on the overall execution time. As $W$ increases, the improvement falls (it can even be negative) because load balancing and waiting times become critical factors for performance.

We define the load balancing parameter for $p$ processors as $bal = N/(p \times itmax)$, $0 < bal \leq 1$, where $N$ is the number of iterations and $itmax$ the maximum number of iterations assigned to one processor. As CYT distributes iterations cyclically, load is always balanced. Table 2 presents $bal$ values for $p$=8 using the different algorithms. In general, load balancing obtained for our test matrices is very good, except for those ones with small $M$. For instance, the size of array $A$ is over 500 elements for matrices $mbeacxc$ and $beaflw$, which results in a small number of nodes (in LCYT) and blocks (in LO-LCYT) and, therefore, it is difficult to obtain an optimal distribution.

The reduction in the execution times illustrated in Figure 4 is a consequence of the improvement in data locality

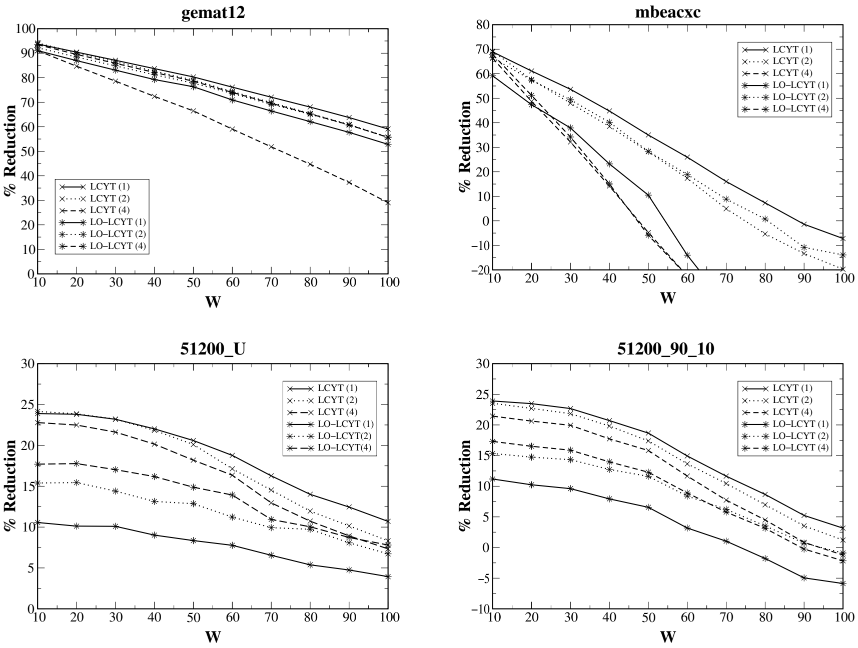


**Figure 4. Percentage of reduction in execution times on 8 processors for some test matrices**

**Table 2. Load balancing of LCYT/LO-LCYT**

| | LCYT($n$) | | | LO-LCYT($n$) | | |
|---|---|---|---|---|---|---|
| | $n$=1 | $n$=2 | $n$=4 | $n$=1 | $n$=2 | $n$=4 |
| *gemat1* | .971 | .912 | .854 | .948 | .941 | .920 |
| *gemat12* | .985 | .918 | .822 | .956 | .953 | .922 |
| *mbeacxc* | .639 | .550 | .331 | .555 | .643 | .336 |
| *beaflw* | .582 | .553 | .332 | .568 | .644 | .338 |
| *psmigr_2* | .919 | .853 | .768 | .797 | .711 | .699 |
| *25600_U* | .997 | .988 | .956 | .972 | .973 | .958 |
| *25600_90_10* | .982 | .981 | .947 | .913 | .925 | .926 |
| *51200_U* | .997 | .992 | .977 | .968 | .983 | .975 |
| *51200_90_10* | .998 | .983 | .985 | .909 | .963 | .953 |

of our approaches. We used the R10K event counters to measure L1 and L2 cache misses, as well as the number of L2 invalidations. Figure 5 shows the results (normalized with respect to the CYT algorithm) for each matrix on 8 processors using LCYT(1) and LO-LCYT(2). The reduction in the number of cache misses and invalidations is very significant, mainly for the HB matrices, since $M << N$ in these matrices, and thus the probability of reuse is higher. The best memory hierarchy optimization achieved by *gemat12* results in the highest reduction in execution time.

Figure 6 shows the executor speedups on 8 processors for different workloads using the CYT, LCYT(1) and LO-LCYT(2) algorithms. In general, LCYT has a better behavior, although LO-LCYT achieves acceptable results in almost all cases. Our proposals work better for loops with low $W$ because in this case memory hierarchy performance has a greater influence on the overall execution time. The maximum achievable speedup is limited by the degree of parallelism of the loop. Loops with long dependence chains prevent parallelism. The CYT algorithm obtains the best speedups for the synthetic matrices, which have a shorter $CP$; and the worst speedup is for *gemat1*, which has the longest $CP$ of the test suite of Table 1. Although matrix *gemat12* has a short $CP$, the speedup achieved by CYT is very low. Note that the critical path is only an estimate of parallelism degree and speedup is strongly influenced by the iteration distribution among processors, so that the number of waits is minimized. Moreover, speedup increases with $W$ because the executor overhead becomes negligible.

In order to represent the impact of the inspector overhead, Figure 7 shows, for an HB and a synthetic matrix, the overall execution time of the algorithms CYT, LCYT(1) and LO-LCYT(2), given by $T = T_i + N_{it} \times T_e$ ms, where $T_i$ and $T_e$ are the inspector and executor times, respectively, and $N_{it}$ is the number of times the inspector is reused in the code (from 1 to 100 times in our experiments). In many applications, the loop to be parallelized is contained in one or more sequential loops. In this case, if access patterns to array $A$ do not change across iterations, the inspector can be reused (e.g. iterative sparse linear system solvers).

The overall execution times of LCYT and LO-LCYT are always lower than those of CYT for the HB matrix and the difference increases as $W$ diminishes. LCYT achieves the best results in *gemat12* for all parameter combinations because $M$ is relatively small and it results in a low graph partitioning cost. Regarding the synthetic matrix, as $M$ is much higher, the LCYT inspector overhead is more significant and, therefore, this algorithm is only advantageous as compared with CYT if the inspector is reused a certain

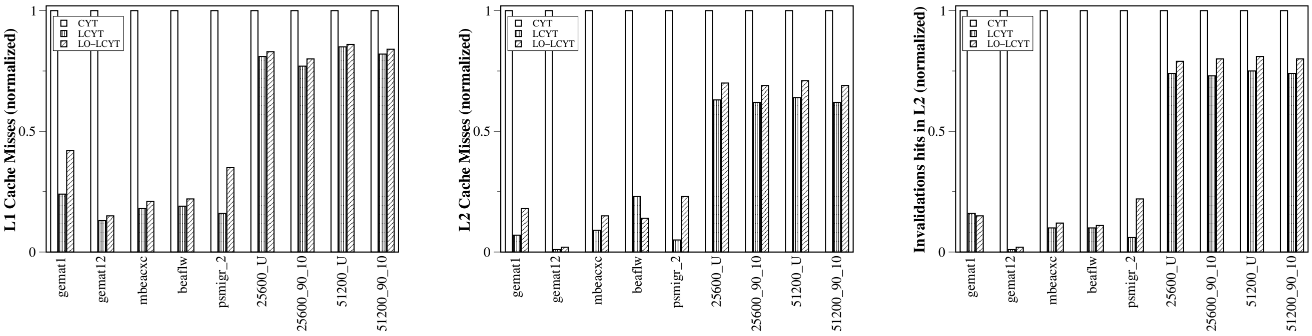


**Figure 5. Cache behavior of the three strategies**

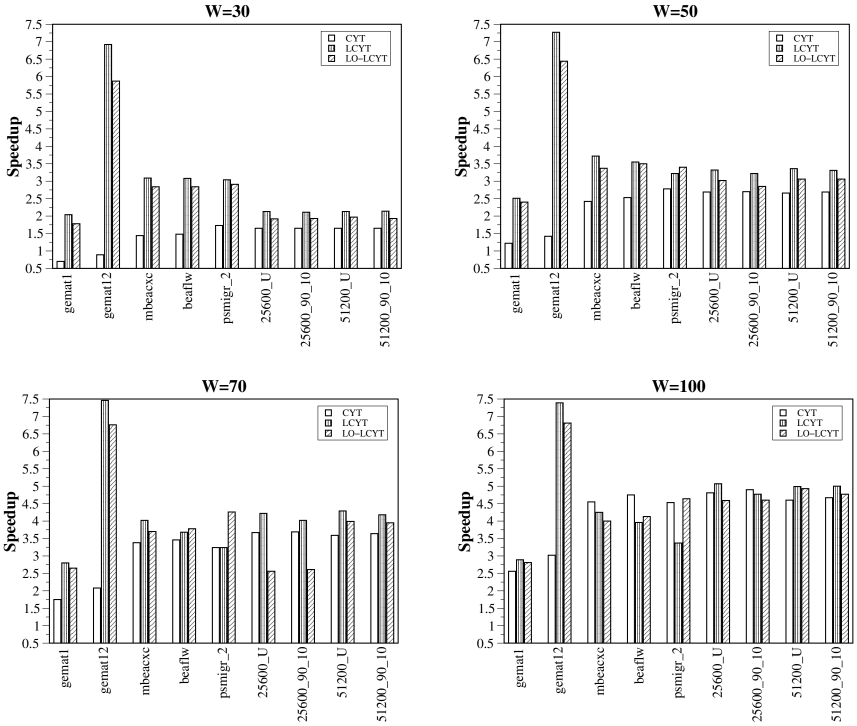


**Figure 6. Speedups on 8 processors for different workloads**

number of times (from $N_{it}$=20 for $W$=30, and $N_{it}$=26 for $W$=70). As in this case the inspector overhead of LO-LCYT is less, LCYT performance is better from $N_{it}$=49 for $W$=30, and $N_{it}$=64 for $W$=70. Below that threshold, the best overall performance is achieved by LO-LCYT.

Matrices *gemat12* and *51200_U* represent the behavior of the HB and the synthetic test matrices, respectively. Table 3 shows the overall behavior of all test matrices for different $W$. It contains the number of iterations ($N_{it}$) from which the overall execution times of LCYT and LO-LCYT (LO in the table) are lower than those of CYT. If this number is 1, it means that the execution time is always lower, and it is not necessary to reuse the inspector to improve the results. This is the case for the HB matrices using both LCYT and LO-LCYT and for the synthetic matrices using LO-LCYT (except for $W$=70). The entry with a dash means that the overall execution time is never improved because the executor time of CYT is lower (the only case is matrix *psmigr_2* using LCYT for $W \geq$70). The number in parentheses in the LCYT columns is the value of $N_{it}$ from which LCYT outperforms LO-LCYT. The results show that LCYT is better than LO-LCYT for most HB matrices, but for the synthetic matrices LCYT is better only if the inspector is reused because, as $M = 2N$, the graph partitioning cost is high.

## 5. Related work

Cache misses are becoming increasingly costly due to the widening gap between processor and memory performance. Therefore, it is a primary goal to increase the performance of each memory hierarchy level. Much research

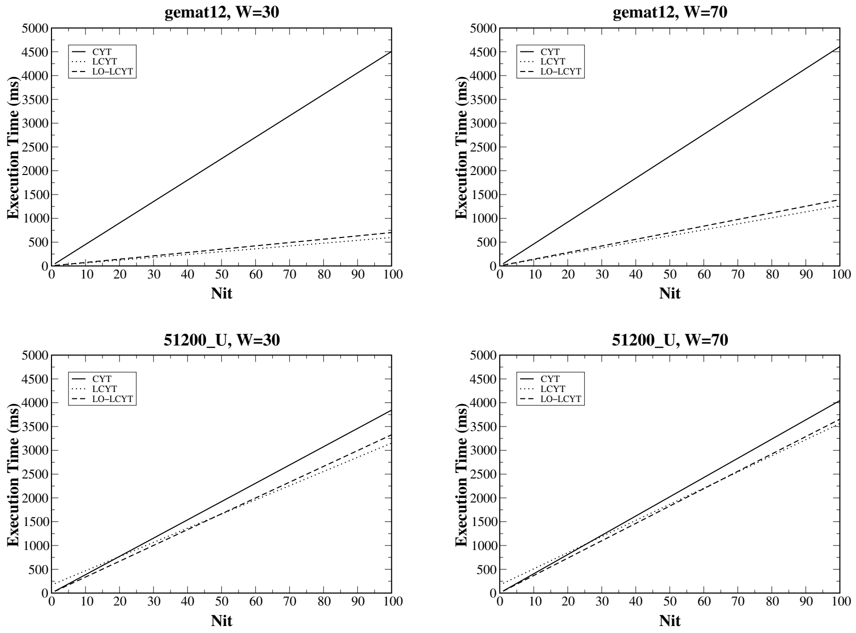


**Figure 7. Overall execution times on 8 processors for some benchmark matrices**

**Table 3. Threshold $N_{it}$ for outperforming CYT**

| | $W$=30 | | $W$=50 | | $W$=70 | |
|---|---|---|---|---|---|---|
| | LCYT | LO | LCYT | LO | LCYT | LO |
| *gemat1* | 1 (2) | 1 | 1 (1) | 1 | 1 (2) | 1 |
| *gemat12* | 1 (1) | 1 | 1 (1) | 1 | 1 (1) | 1 |
| *mbeacxc* | 1 (1) | 1 | 1 (1) | 1 | 1 (1) | 1 |
| *beaflw* | 1 (1) | 1 | 1 (1) | 1 | 1 (1) | 2 |
| *psmigr_2* | 1 (1) | 1 | 1 (-) | 1 | - (-) | 1 |
| *25600_U* | 18 (49) | 1 | 21 (49) | 1 | 31 (56) | 2 |
| *25600_90* | 17 (48) | 1 | 23 (56) | 1 | 46 (75) | 3 |
| *51200_U* | 20 (49) | 1 | 22 (56) | 1 | 26 (64) | 1 |
| *51200_90* | 18 (47) | 1 | 22 (56) | 1 | 33 (71) | 2 |

has been devoted to enhancing data locality of dense arrays with regular access patterns, by means of loop and/or data transformations [9] [13] [20]. Regarding irregular codes, there are different proposals to improve locality of sequential codes on uniprocessors. Al-Furaih and Ranka [1] focus on data reordering using METIS [10] and BFS (*Breadth First Search*). Ding and Kennedy [3] propose two transformations: one reorders data accesses to improve temporal locality (locality grouping) and the other reorders data layout to enhance spatial reuse (dynamic data packing). They also assess the performance improvement of applying a combination of both techniques. Mellor-Crummey et al [14] use space-filling curves to reorder data and/or computation.

There is much less research on the improvement of locality on multiprocessors, an important issue on NUMA systems. Han and Tseng evaluate in [7] the effect on the parallel execution of codes of uniprocessor techniques that improve locality, focusing on reduction operations. In [11] Leung and Zahorjan treat locality in the specific context of the parallel execution of a loop with no output dependences. Their strategy is based on array reordering to improve spatial locality. A recent proposal also based on data reordering using space-filling curves to enhance spatial locality of irregular codes on shared memory systems can be found in [8]. Our LCYT proposals, in contrast, are based on loop restructuring and their primary objective is to exploit both spatial and temporal locality, as well as avoid false sharing of data. Moreover, our strategies can be applied to any loop that follows the general pattern represented in Figure 1.

## 6. Conclusions

Kernels of grand challenge applications that use irregular structures make poor use of memory hierarchy on multiprocessors and performance degrades as a result. As techniques to enhance memory performance have become increasingly important, we have proposed two methods (LCYT and LO-LCYT) to parallelize loops with indirect array accesses using run-time support. Compared to existing research on irregular codes, our algorithms are designed not only to enhance parallelism, but also data locality, improving temporal and spatial locality and eliminating false sharing.

Experimental results show the effectiveness of both methods, which reduce the number of cache misses and invalidations. It results in a significant reduction in the execution times of the executor (except for high workloads), LCYT being the method that achieves the best results in this phase. The main drawback of LCYT is the overhead of the

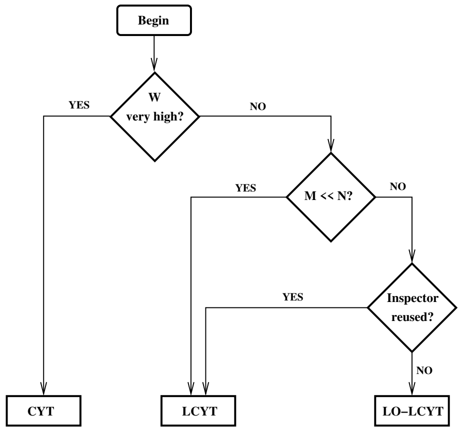


**Figure 8. Choice of the parallelization strategy**

graph partitioning when the size of the indirectly accessed array is not much less than the number of loop iterations. In these cases, the method is only advantageous if the inspector is reused in the code and thus the overhead is partially amortized. This is not the case of LO-LCYT, which obtains good results in almost all circumstances. We can conclude that, excepting very high workloads, the best overall results are achieved by the proposals LCYT or LO-LCYT depending on the input code, as shown in Figure 8.

Our techniques are in the domain of automatic parallelization and the final goal is to include the algorithms in a parallelizing compiler. Thus, the compiler would select the best strategy according to the loop and matrix parameters, following the decision diagram of Figure 8. The parameter thresholds should be empirically determined.

## References


[1] I. Al-Furaih and S. Ranka. Memory Hierarchy Management for Iterative Graph Structures. In *12$^{th}$ Int'l Parallel Processing Symposium*, Orlando, FL, 1998.

[2] D.-K. Chen, J. Torrellas, and P.-C. Yew. An Efficient Algorithm for the Run-Time Parallelization of DOACROSS Loops. In *Supercomputing Conference*, pages 518–527, Washington DC, 1994.

[3] C. Ding and K. Kennedy. Improving Cache Performance in Dynamic Applications through Data and Computation Reorganization at Run Time. In *ACM SIGPLAN'99 Conference on Programming Language Design and Implementation*, pages 229–241, Atlanta, GA, 1999.

[4] I. S. Duff, R. G. Grimes, and J. G. Lewis. User's Guide for the Harwell-Boeing Sparse Matrix Collection. Technical Report TR-PA-92-96, CERFACS, 1992.

[5] M. Gupta and R. Nim. Techniques for Speculative Run-Time Parallelization of Loops. In *Supercomputing Conference*, Orlando, FL, 1998.

[6] H. Han and C.-W. Tseng. Efficient Compiler and Run-Time Support for Parallel Irregular Reductions. *Parallel Computing*, 26(13-14):1861–1887, 2000.

[7] H. Han and C.-W. Tseng. Improving Locality for Adaptive Irregular Scientific Codes. In *13$^{th}$ Int'l Workshop on Languages and Compilers for Parallel Computing*, pages 173–188, Yorktown Heights, NY, 2000.

[8] Y. C. Hu, A. L. Cox, and W. Zwaenepoel. Improving Fine-Grained Irregular Shared-Memory Benchmarks by Data Reordering. In *Supercomputing Conference*, Dallas, TX, 2000.

[9] M. T. Kandemir, A. N. Choudhary, J. Ramanujam, and P. Banerjee. Improving Locality Using Loop and Data Transformations in an Integrated Framework. In *31$^{st}$ Annual IEEE/ACM Int'l Symposium on Microarchitecture*, pages 285–297, Dallas, TX, 1998.

[10] G. Karypis and V. Kumar. A Fast and High Quality Multilevel Scheme for Partitioning Irregular Graphs. *SIAM Journal on Scientific Computing*, 20(1):359–392, 1999.

[11] S.-T. Leung and J. Zahorjan. Restructuring Arrays for Efficient Parallel Loop Execution. Technical Report 94-02-01, Department of Computer Science and Engineering, University of Washington, 1994.

[12] S.-T. Leung and J. Zahorjan. Extending the Applicability and Improving the Performance of Runtime Parallelization. Technical Report 95-01-08, Department of Computer Science and Engineering, University of Washington, 1995.

[13] K. S. McKinley, S. Carr, and C.-W. Tseng. Improving Data Locality with Loop Transformations. *ACM Trans. on Programming Languages and Systems*, 18(4):424–453, 1996.

[14] J. M. Mellor-Crummey, D. B. Whalley, and K. Kennedy. Improving Memory Hierarchy Performance for Irregular Applications. In *ACM Int'l Conference on Supercomputing*, pages 425–433, Rhodes, Greece, 1999.

[15] S. P. Midkiff and D. A. Padua. Compiler Algorithms for Synchronization. *IEEE Transactions on Computers*, 36(12):1485–1495, 1987.

[16] D. Patel and L. Rauchwerger. Implementation Issues of Loop-Level Speculative Run-Time Parallelization. In *8$^{th}$ Int'l Conference on Compiler Construction*, pages 183–197, Amsterdam, The Netherlands, 1999.

[17] Y. Saad. *SPARSKIT: a Basic Tool Kit for Sparse Matrix Computations (Version 2)*, 1994.

[18] J. H. Saltz, R. Mirchandaney, and K. Crowley. Run-Time Parallelization and Scheduling of Loops. *IEEE Transactions on Computers*, 40(5):603–612, 1991.

[19] D. E. Singh, F. F. Rivera, and M. J. Martín. Run-Time Characterization of Irregular Accesses Applied to Parallelization of Irregular Reductions. In *30$^{th}$ Int'l Conference on Parallel Processing Workshops*, pages 17–22, Valencia, Spain, 2001.

[20] M. E. Wolf and M. S. Lam. A Data Locality Optimizing Algorithm. In *ACM SIGPLAN'91 Conference on Programming Language Design and Implementation*, pages 30–44, Toronto, Canada, 1991.

[21] C. Xu. Effects of Parallelism Degree on Run-Time Parallelization of Loops. In *31$^{st}$ Hawaii Int'l Conference on System Sciences*, Kohala Coast, HI, 1998.

[22] C. Xu and V. Chaudhary. Time Stamp Algorithms for Runtime Parallelization of DOACROSS Loops with Dynamic Dependences. *IEEE Transactions on Parallel and Distributed Systems*, 12(5):433–450, 2001.

[23] C.-Q. Zhu and P.-C. Yew. A Scheme to Enforce Data Dependence on Large Multiprocessor Systems. *IEEE Transactions on Software Engineering*, 13(6):726–739, 1987.