An Inspector-Executor Algorithm for Irregular Assignment Parallelization

Manuel Arenaz, Juan Touriño, Ramón Doallo

Computer Architecture Group
Dep. Electronics and Systems, University of A Coruña, Spain
{arenaz,juan,doallo}@udc.es

Abstract. A loop with irregular assignment computations contains loop-carried output data dependences that can only be detected at run-time. In this paper, a load-balanced method based on the inspector-executor model is proposed to parallelize this loop pattern. The basic idea lies in splitting the iteration space of the sequential loop into sets of conflict-free iterations that can be executed concurrently on different processors. As will be demonstrated, this method outperforms existing techniques. Irregular access patterns with different load-balancing and reusability properties are considered in the experiments.

1 Introduction

Research on run-time techniques for the efficient parallelization of irregular computations has been frequently referenced in the literature in recent years [4, 5, 7, 8, 10, 14, 15]. An *irregular assignment* pattern consists of a loop with f_{size} iterations, f_{size} being the size of the subscript array f (see Figure 1). At each iteration h, value $rhs(h)$ is assigned to the array element $A(f(h))$. Neither the right-hand side expression $rhs(h)$ nor any function call make within it contain occurrences of A, thus the code is free of loop-carried true data dependences. Nevertheless, as the subscript expression $f(h)$ is loop-variant, loop-carried output data dependences may be present at run-time (unless f is a permutation array). This loop pattern can be found in different application fields such as computer graphics algorithms [3], finite elements applications [12], or routines for sparse matrix computations [11].

Knobe and Sarkar [6] describe a program representation that uses *array expansion* [13] to enable the parallel execution of irregular assignment computations. Each processor executes a set of iterations preserving the same relative order of the sequential loop. Array A is expanded in order to allow different processors to store partial results in separate memory locations. For each array entry $A(j)$, with $j = 1, ..., A_{size}$, the global result is computed by means of a reduction operation that obtains the partial result that corresponds with the highest iteration number. Each processor computes this reduction operation for a subset of array elements.

An optimization to perform element-level dead code elimination at run-time is also presented in [6]. In irregular assignments, the same array element may be

```
A(...) = ...
DO h = 1, f_size
   A(f(h)) = rhs(h)
END DO
... = ...A(...)...
```

Fig. 1. Irregular assignment pattern.

computed several times, though only the last value is used after the loop ends. Consequently, intermediate values need not be computed. Classical dead code elimination typically removes assignment statements from the source code. This technique eliminates unnecessary array element definitions at run-time.

In this paper we use the inspector-executor model to parallelize irregular assignments on scalable shared memory multiprocessors. We show that this model can be efficiently applied to the parallelization of static/adaptive irregular applications, preserving load-balancing and exploiting uniprocessor data write locality. A preliminary work [1] did not include a theoretical performance analysis based on a formal characterization of static/adaptative irregular applications, and presented a quite limited performance evalutation. The technique described in this paper is embedded in our compiler framework [2] for automatic kernel recognition and its application to automatic parallelization of irregular codes.

The rest of the paper is organized as follows. Our parallelization method is presented in Section 2. The performance of our technique is compared with the array expansion approach in Section 3. Experimental results conducted on a SGI Origin 2000 using a rasterization algorithm as case study are shown in Section 4. Finally, conclusions are discussed in Section 5.

2 Parallel Irregular Assignment

In this section, we propose a run-time technique that uses the inspector-executor model to parallelize irregular assignments. The basic idea lies in reordering loop iterations so that data write locality is exploited on each processor. Furthermore, the amount of computations assigned to each processor is adjusted so that load-balancing is preserved.

The method is as follows. In the inspector code shown in Figure 2, array A is divided into subarrays of consecutive locations, A_p ($p = 1, ..., P$ where P is the number of processors), and the computations associated with each block are assigned to different processors. Thus, the loop iteration space $(1, ..., f_{size})$ is partitioned into sets f_p that perform write operations on different blocks A_p. The sets f_p are implemented as linked lists of iteration numbers using two arrays, $count(1:P)$ and $next(1:f_{size}+P)$. Each processor p has an entry in both arrays, $count(p)$ and $next(f_{size}+p)$. The entry $next(f_{size}+p)$ stores the first iteration number h_1^p assigned to processor p. The next iteration number, h_2^p, is stored in array entry $next(h_1^p)$. This process is repeated $count(p)$ times, i.e. the number of elements in the list. In the executor code of Figure 3, each processor

```
! Accumulative frequency distribution
his(1 : A_size) = 0
DO h = 1, f_size
    his(f(h)) = his(f(h)) + 1
END DO
DO h = 2, A_size
    his(h) = his(h) + his(h − 1)
END DO

! Computation of the linked lists
Refs = (his(A_size)/P) + 1
count(1 : P) = 0
DO h = 1, f_size
    thread = (his(f(h))/Refs) + 1
    IF (count(thread).eq.0) THEN
        next(f_size + thread) = h
    ELSE
        next(prev(thread)) = h
    END IF
    prev(thread) = h
    count(thread) = count(thread) + 1
END DO
```

Fig. 2. Inspector code.

```
A(...) = ...
DOALL p = 1, P
    h = next(f_size + p)
    DO k = 1, count(p)
        A(f(h)) = ...
        h = next(h)
    END DO
END DOALL
... = ...A(...)...
```

Fig. 3. Executor code.

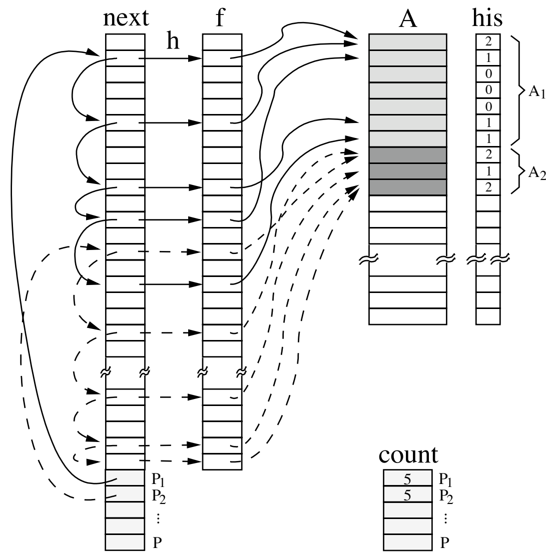


Fig. 4. Inspector-executor approach.

p executes the conflict-free computations associated with the loop iterations contained in a set f_p. Figure 4 shows a graphical description of the method. The figure represents the linked-lists f_1 and f_2 of processors p_1 and p_2 as solid and dashed lines, respectively. The corresponding subarrays A_1 and A_2 are depicted as shaded regions within array A.

Load-balancing is preserved by splitting array A into subarrays A_p of different size in the inspector stage. As shown in the code of Figure 2, the inspector first computates the accumulative frequency distribution $his(1 : A_{size})$. For each array entry $A(j)$ with $j = 1, ..., A_{size}$, $his(j)$ stores the sum of the number of write references to $A(1), A(2), ..., A(j)$. The second step consists of building the linked lists f_p by determining the list corresponding to each entry of the subscript array f (see variable *thread* in Figure 2). The appropiate list is easily computed as $his(f(h))/Refs + 1$, where $Refs$ is the mean number of iterations of the sequential loop per processor. As illustrated in Figure 4, load-balancing is preserved because, as A_1 and A_2 have different sizes (7 and 3, respectively), processors P_1 and P_2 are both assigned 5 iterations of the sequential loop.

Element-level dead code elimination can be implemented in the inspector-executor model, too. In this case, the linked lists only contain the last iteration at which array elements, $A(j)$, are modified. This difference is highlighted in Figure 5 where, unlike Figure 4, there are dotted arrows representing the loop iterations that are not computed. The code of the optimized inspector (the executor does not change) is shown in Figure 6. The accumulative frequency distribution array, $his(1 : A_{size})$, contains the number of array entries in the range $A(1), A(2), ..., A(j)$ that are modified during the execution of the irregular assignment. Note that an additional array, *iter*, is needed to store the last iteration number at which the elements of array A are modified. Finally, the phase that computes the linked lists is rewritten accordingly.

3 Performance Analysis

Memory overhead complexity of the array expansion technique proposed in [6] is $\mathcal{O}(A_{size} \times P)$ which, in practice, prevents the application of this method for large array sizes and a high number of processors. In contrast, memory overhead of our inspector-executor method is $\mathcal{O}(max(f_{size} + P, A_{size}))$. Note that the extra memory is not directly proportional to the number of processors. In practice, the complexity is usually $\mathcal{O}(f_{size})$, as $f_{size} \gg P$, or $\mathcal{O}(A_{size})$.

The efficiency of the parallelization techniques for irregular assignments is determined by the properties of the irregular access pattern. In our analysis, we have considered the following parameters proposed in [15] for the parallelization of irregular reductions: *degree of contention (C)*, number of loop iterations referencing an array element; *sparsity (SP)*, ratio of different elements referenced in the loop ($A_{updated}$) and the array size; *connectivity (CON)*, ratio of the number of loop iterations and the number of distinct array elements referenced in the loop; and *adaptivity* or *reusability (R)*, the number of times that an access pattern is reused before being updated.

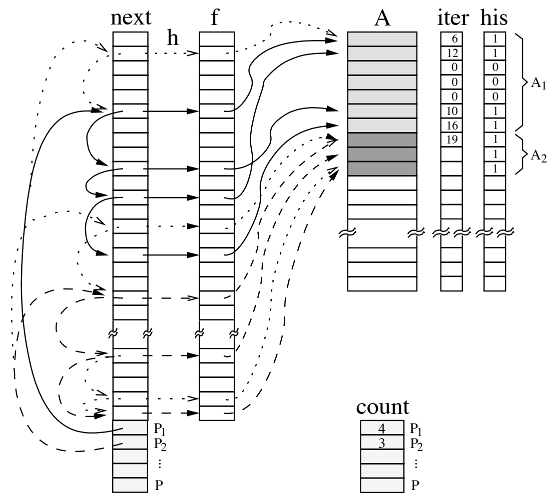


Fig. 5. Inspector-executor approach when dead code elimination is applied.

```
! Accumulative frequency distribution
iter(1 : A_size) = 0
his(1 : A_size) = 0
DO h = 1, f_size
    iter(f(h)) = h
    his(f(h)) = 1
END DO
DO h = 2, A_size
    his(h) = his(h) + his(h − 1)
END DO

! Computation of the linked lists
Refs = (his(A_size)/P) + 1
count(1 : P) = 0
DO h = 1, A_size
    IF (iter(h).gt.0) THEN
        thread = (his(h)/Refs) + 1
        IF (count(thread).eq.0) THEN
            next(f_size + thread) = iter(h)
        ELSE
            next(prev(thread)) = iter(h)
        END IF
        prev(thread) = iter(h)
        count(thread) = count(thread) + 1
    END IF
END DO
```

Fig. 6. Inspector when dead code elimination is applied.

Unlike the array expansion approach, the inspector-executor technique takes advantage of the adaptive nature of irregular applications. The computational overhead is associated with the inspector stage because the executor is fully parallel (it performs conflict-free computations). In static codes, the inspector overhead is negligible because it is computed only once and then reused during the program execution ($R \to \infty$). Thus, as the parallel execution time can be accurately approximated by the time of the executor, the efficiency $E \to 1$ as reusability R increases. In dynamic codes, the inspector is recomputed periodically. Supposing that the access pattern changes every time the executor is run ($R = 0$), a lower bound of the efficiency is

$$E = \frac{\#iters\ t_s}{P(T_s^{INSP} + \frac{\#iters}{P} t_s)} = \frac{\#iters\ t_s}{PT_s^{INSP} + \#iters\ t_s} \tag{1}$$

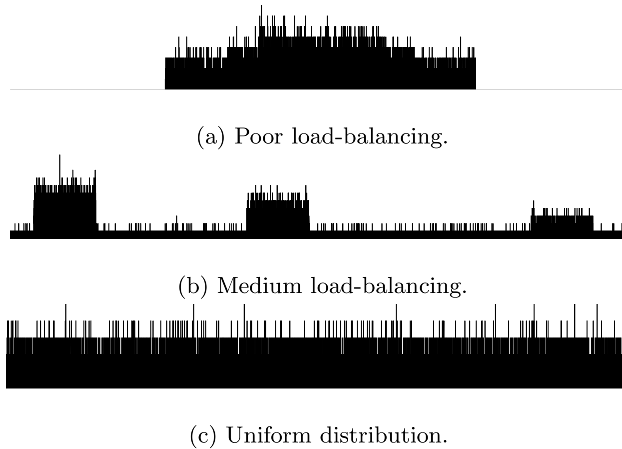
where t_s is the execution time of one iteration of the sequential irregular loop, T_s^{INSP} represents the execution time of the sequential inspector, and $\#iters$ is the number of loop iterations actually executed: f_{size} when dead-code elimination is not applied, and $A_{updated}$ when dead-code is applied. The execution time of the parallel irregular assignment is given by $T_p = T_s^{INSP} + \frac{\#iters}{P} t_s$.

As a result, the efficiency of the inspector-executor approach for any R is bounded as follows:

$$\frac{f_{size}\ t_s}{PT_s^{INSP} + \#iters\ t_s} \leq E \leq 1 \tag{2}$$

Lower efficiencies are obtained as R decreases because the irregular access pattern changes more frequently. From now on we will assume a fixed array size A_{size}. When dead code is not applied, T_s^{INSP} increases as f_{size} raises (if SP is constant, CON and f_{size} raise at the same rate). Thus, a higher lower bound is achieved if the time devoted to useful computations ($f_{size} t_s$) grows faster than the computational overhead (PT_s^{INSP}). Supposing that SP is constant, when dead code elimination is applied, the lower bound does not change because both useful computations ($A_{updated} t_s$) and overhead T_s^{INSP} remain constant as f_{size} raises.

The inspector-executor method presented in this paper preserves load-balancing, the exception being the case in which dead code elimination is not applied and the access pattern contains hot spots, i.e. array entries where most of the computation is concentrated ($SP \to 0$ and $C \to \infty$). On the other hand, the array expansion approach may unbalance workload if dead code elimination is applied. This is because as $rhs(h)$ (see Figure 1) is computed during the reduction operation that finds the partial result corresponding to the highest iteration number, it is only computed for $A_{updated}$ array elements. As a result, workload will be unbalanced if computations associated with modified elements are not uniformly distributed among processors. In other words, load-balancing is achieved if $SP \to 1$. Otherwise, the array expansion approach does not assure load-balancing because the contention distribution C of the irregular access pattern is not considered in the the mapping of computations to processors.



(a) Poor load-balancing.

(b) Medium load-balancing.

(c) Uniform distribution.

Fig. 7. Irregular access patterns.

4 Performance Evaluation

In this section we present experimental results to compare the performances of our technique and the array expansion method; different parameter combinations that characterize irregular assignments are considered. The target machine was a SGI Origin2000 cc-NUMA multiprocessor. OpenMP [9] shared memory directives have been used in the parallel implementation.

4.1 Experimental Conditions

In our experiments, we have considered the parameters degree of contention (C), sparsity (SP), connectivity (CON) and reusability (R), defined in Section 3. As case study, we use the generic convex polygon scan conversion [3], a well-known rasterization algorithm from computer graphics. This algorithm presents output dependences that arise from the depiction of a set of polygons, which compose an image/scene, on a display buffer, A. A typical size for the display buffer is $A_{size} = 512 \times 512 = 262,144$ pixels. We have also considered three access patterns that represent typical cases in which the scan conversion is used (see Figure 7): a pattern with poor load-balancing that represents an scene where all the objects are displayed on a region of the buffer ($SP = 0.36$, array elements with $C > 0$ are confined in a specific region); a second pattern presents medium load-balancing that is associated with an image where most objects are concentrated on several regions of the display ($SP = 0.30$, array elements with $C > 0$ are uniformly distributed along the array, but there exist several regions with a higher C); and a third pattern that is characterized by uniformly distributed objects ($SP = 0.32$). We have considered $5,000$, $10,000$ and $20,000$ polygons to cover a reasonable range typically found in rasterization. Assuming a fixed mean number of 20

pixels per polygon, the total number of references (i.e. loop iterations, f_{size}) to the array A is $100,000$ ($CON \approx 1.20$), $200,000$ ($CON \approx 2.41$) and $400,000$ ($CON \approx 4.81$), respectively. The experimental results presented in the following sections were obtained by fixing $A_{size} = 262,144$ and $SP \approx 0.33$, on average. As a result, conclusions can be stated in terms of CON and f_{size}.

4.2 Experimental Results

When element-level dead code elimination is not applied, computational load is measured as the maximum number of loop iterations that is assigned to the processors. Both methods preserve load-balancing by assigning approximately f_{size}/P iterations to each processor, P being the number of processors. Figures 8 and 9 present execution times and speed-ups for different CON and R values. The access pattern, which is defined in terms of SP and C, is not relevant in this case. Execution times increase as CON raises because CON is related to the amount of computational load assigned to processors; it does not affect workload distribution. Note that memory overhead $\mathcal{O}(A_{size} \times P)$ prevents the execution of the array expansion approach on more than 15 processors, which is a drawback if a high number of processors is needed.

The speed-ups of the array expansion approach (dotted lines) increase as CON raises (this method does not take advantage of reusability) because the computational overhead of this method mainly depends on the reduction operation that determines the value of each array element $A(j)$, $j = 1, ..., A_{size}$, by combining the partial results computed by the processors (A_{size} and SP are constants). In the figure, speed-ups increase approximately 35% on 15 processors when CON is doubled for $A_{size} = 262,144$ and $SP \approx 0.33$. In contrast, the speed-ups of our inspector-executor technique (shaded region) depend on CON and R. In static codes ($R \to \infty$), efficiency is approximately 1 in any case (solid-star line). However, in dynamic applications, the sequential inspector imposes an upper limit on the maximum achievable speed-up (see Section 3). The curve of speed-ups for totally dynamic codes is a lower bound of the speed-up of the inspector-executor approach (see Eq. (2)). The lower bound raises when CON is increased (solid lines with $R = 0$) because the time devoted to useful computations grows faster than the computational overhead. In particular, the increment is approximately 6% on 32 processors when CON is doubled. Lower speed-ups are obtained as R decreases because the access pattern has to be rescanned a higher number of times during the execution of the program.

4.3 Results with Dead Code Elimination

The generic scan conversion algorithm depicts all the polygons that represent an image on the display buffer although, at the end, only the visible regions of the polygons remain on the display. As a result, computational resources are consumed in the depiction of invisible polygons. When element-level dead code elimination is applied, only the visible regions of the polygons are printed on the display buffer, with the corresponding saving of resources. In this case,

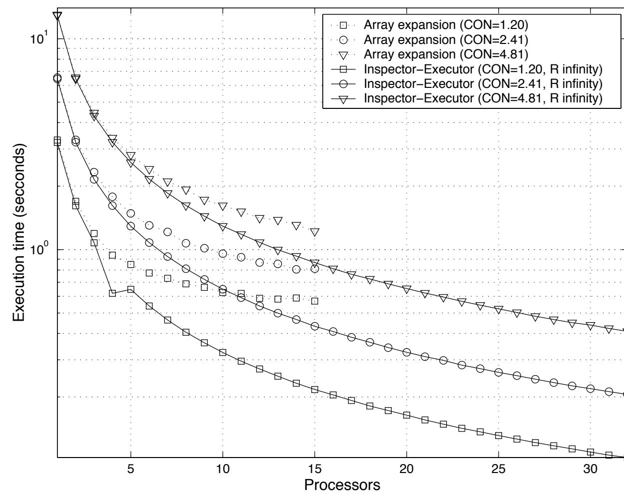


Fig. 8. Execution times.

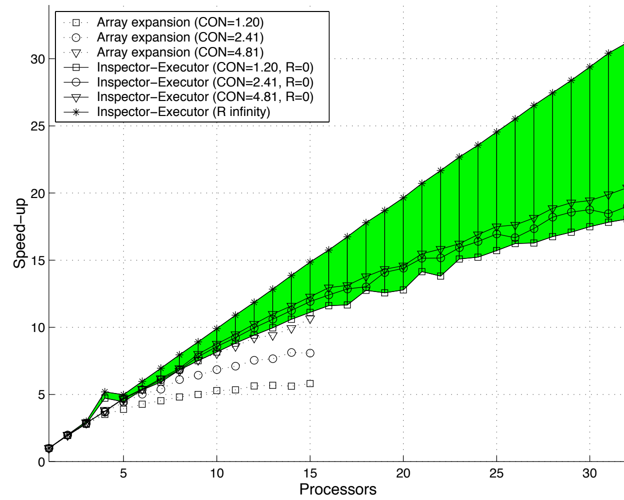


Fig. 9. Speed-ups.

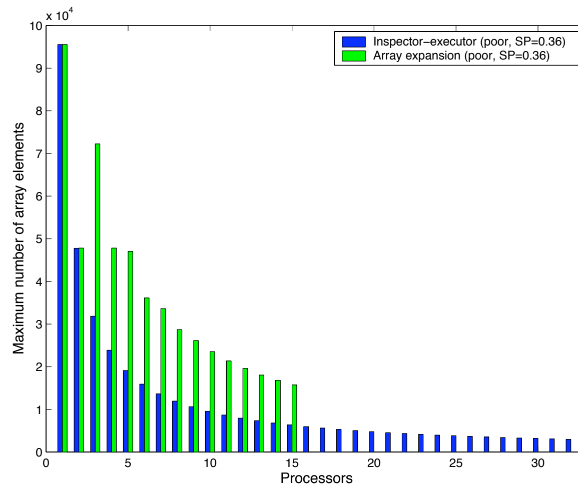


Fig. 10. Computational load when dead code elimination is applied.

computational load is measured as the maximum number of array elements that are computed by the processors. Figure 10 represents the computational load corresponding to the access pattern with poor load-balancing ($SP = 0.36$ and $C = 0$ for large subarrays of A) when dead code elimination is applied. Unlike our inspector-executor technique (black bars), the array expansion method (gray bars) presents load-unbalancing because array elements $A(j)$ are assigned to processors independently of the contention distribution C.

Note that workload depends on the distribution of modified array elements (SP and C), while it depends on CON if dead code elimination is not applied. Figures 11 and 12 show execution times and speed-ups when dead code elimination is applied. The parameter SP is ≈ 0.33 for all the access patterns described in Section 4.1 because load-balancing increases in the array expansion approach as $SP \rightarrow 1$. The inspector-executor method outperforms the array expansion technique which, in addition, is highly sensitive to the contention distribution of the access pattern.

5 Conclusions

A scalable method to parallelize irregular assignment computations is described in this work. Unlike previous techniques based on array expansion, the method uses the inspector-executor model to reorder computations so that load-balancing is preserved and data write locality is exploited.

Performance evaluation shows that our method outperforms the array expansion approach either using dead code elimination or not. Furthermore, the applicability of array expansion is limited by its memory requirements in practice. The inspector-executor model is appropriate to develop parallelization techniques that take advantage of the adaptive nature of irregular applications.

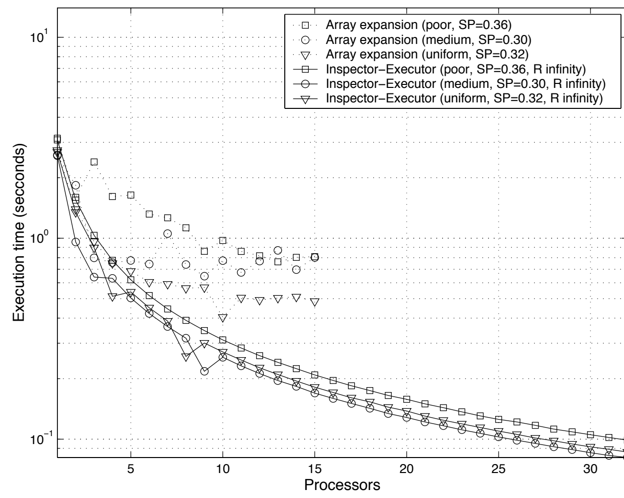


Fig. 11. Execution times when dead code elimination is applied.

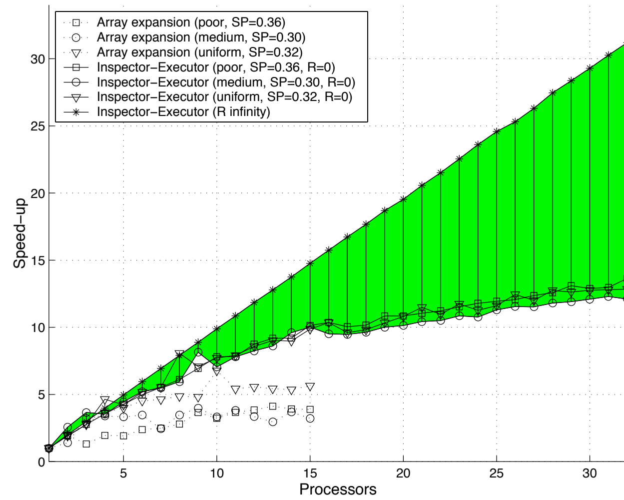


Fig. 12. Speed-ups when dead code elimination is applied.

Acknowledgements

We gratefully thank Complutense Supercomputing Center in Madrid for providing access to the SGI Origin 2000 multiprocessor. This work was supported by the Ministry of Science and Technology of Spain and FEDER funds under contract TIC2001-3694-C02-02.

References

1. Arenaz, M., Touriño, J., Doallo, R.: Irregular Assignment Computations on cc-NUMA Multiprocessors. In Proceedings of 4th International Symposium on High Performance Computing, ISHPC-IV, Kansai Science City, Japan, Lecture Notes in Computer Science, Vol. 2327 (2002) 361–369
2. Arenaz, M., Touriño, J., Doallo, R.: A GSA-Based Compiler Infrastructure to Extract Parallelism from Complex Loops. In Proceedings of 17th ACM International Conference on Supercomputing, ICS'2003, San Francisco, CA (2003) 193–204
3. Glassner, A.: Graphics Gems. Academic Press (1993)
4. Gutiérrez, E., Plata, O., Zapata, E.L.: Balanced, Locality-Based Parallel Irregular Reductions. In Proceedings of 14th International Workshop on Languages and Compilers for Parallel Computing, LCPC'2001, Cumberland Falls, KY (2001)
5. Han, H., Tseng, C.-W.: Efficient Compiler and Run-Time Support for Parallel Irregular Reductions. Parallel Computing **26**(13-14) (2000) 1861–1887
6. Knobe, K., Sarkar, V.: Array SSA Form and Its Use in Parallelization. In Proceedings ACM SIGACT-SIGPLAN Symposium on the Principles of Programming Languages (1998) 107–120
7. Lin, Y., Padua, D.A.: On the Automatic Parallelization of Sparse and Irregular Fortran Programs. In Proceedings of 4th Workshop on Languages, Compilers, and Run-Time Systems for Scalable Computers, LCR'98, Pittsburgh, PA, Lecture Notes in Computer Science, Vol. 1511 (1998) 41–56
8. Martín, M.J., Singh, D.E., Touriño, J., Rivera, F.F.: Exploiting Locality in the Run-time Parallelization of Irregular Loops. In Proceedings of 31st International Conference on Parallel Processing, ICPP 2002, Vancouver, Canada (2002) 27–34
9. OpenMP Architecture Review Board: OpenMP: A Proposed Industry Standard API for Shared Memory Programming (1997)
10. Rauchwerger, L., Padua, D.A.: The LRPD Test: Speculative Run-Time Parallelization of Loops with Privatization and Reduction Parallelization. IEEE Transactions on Parallel and Distributed Systems **10**(2) (1999) 160–180
11. Saad, Y.: SPARSKIT: A Basic Tool Kit for Sparse Matrix Computations. http://www.cs.umn.edu/Research/darpa/SPARSKIT/sparskit.html (1994)
12. Turek, S., Becker, C.: Featflow: Finite Element Software for the Incompressible Navier-Stokes Equations. User Manual. http://www.featflow.de (1998)
13. Wolfe, M.J.: Optimizing Supercompilers for Supercomputers. Pitman, London and The MIT Press, Cambridge, Massachussets (1989)
14. Xu, C.-Z., Chaudhary, V.: Time Stamp Algorithms for Runtime Parallelization of DOACROSS Loops with Dynamic Dependences. IEEE Transactions on Parallel and Distributed Systems **12**(5) (2001) 433–450
15. Yu, H., Rauchwerger, L.: Adaptive Reduction Parallelization Techniques. In Proceedings of the 14th ACM International Conference on Supercomputing, Santa Fe, NM (2000) 66–77