Design and Evaluation of an Environment APE for Automatic Parallelization of Programs *

Vipin Chaudhary, Chengzhong Xu, Sumit Roy, Jialin Ju, Vikas Sinha, and Laiwu Luo
Parallel and Distributed Computing Laboratory
Department of Electrical and Computer Engineering
Wayne State University, MI 48202

Abstract

In this paper, we have presented the design and evaluation of a compiler system, called APE, *for automatic parallelization of scientific and engineering applications on distributed memory computers.* APE *is built on top of SUIF compiler. It extends SUIF with capabilities in parallelizing loops with non-uniform cross-iteration dependencies, and in handling loops that have indirect access patterns. We have evaluated the effectiveness of SUIF with several CFD test codes, and found that SUIF handles uniform loops over dense and regular data structures very well. For non-uniform loops, an innovative and efficient parallelization approach based on convex theory have been proposed and is being implemented. We have also presented a class of scalable algorithms for parallel distribution and redistribution of unstructured data structures during parallelizing irregular loops.*

1 Introduction

Parallel computers offer the promise of a quantum leap in computing power. To take full advantage of their potential, however, programmers are often forced to manually distribute code and data onto memory systems in addition to specifying how parallel processors cooperate over the execution course. Such hand-coding is time-consuming and error-prone. What is worse, the parallel codes generated in such a way are rarely portable across different machines. A grand challenging approach to relieving programmers from artful parallel programming is automatic program parallelization. Programmers write sequential portable programs in conventional languages, leaving the machine-specific parallelization work to compilers.

Compiler techniques for the automatic detection of parallelism have been studied extensively over the last two decades, and achieved significant progresses in recent years. Current parallelizing compilers, however, are mostly directed towards optimizing control flows over regular data structures on parallel machines with shared global address space. They have limitations in parallelization of loops with non-uniform cross-iteration dependencies and loops with indirect access patterns. Automatic parallelization of programs on distributed memory machines is far beyond the capability of today's compilers because of their weakness in orchestrating data. In order to make parallel computing totally transparent to programmers, parallelizing compilers need breakthroughs in exploiting parallelism and orchestrating data across distributed memory machines.

This paper presents our recent efforts along these two directions. We design and evaluate an environment, named APE, for automatic parallelization of scientific and engineering applications on parallel computers. The APE system is built on top of SUIF compiler. SUIF is a compiler infrastructure that embodies a broad set of program analyses and optimization techniques [7]. It offers a standard intermediate format that facilitates experimentation and development of new compiler techniques. We first evaluate the SUIF system on shared memory multiprocessors and find it is effective in handling loops with uniform data dependences. For nested loops with non-uniform dependences and loops having indirect memory access patterns, SUIF becomes somewhat impotent. Non-uniform and irregular loops appear frequently in applications like computational fluid dynamics (CFD) and molecular dynamics (MD). We first extend SUIF with an efficient tool based on convex theory and linear programming for handling non-uniform loops. The enhanced SUIF is further integrated with run-time libraries in the APE, including a scalable data redistribution strategy, for parallelizing irregular loops on distributed memory machines. The APE targets both message-passing and shared-memory programming models on distributed memory platforms.

*This work was sponsored in part by NSF MIP-9309489, US Army Contract DAEA-32-93-D-004, and Ford Motor Company Grant #0000952185

Related work APE bears a resemblance to Paradigm of Illinois in design philosophy, but Paradigm is built on top of Parafrase-2 and targets solely message-passing models on distributed memory multicomputers [1]. Compilers for Fortran extensions like HPF and Fortran D are assisted by programmer-provided directives, which specify how to distribute data and whether a DO-loop is parallelizable. They relieve the programmers of architecture-dependent parallelization tasks to a limited extent because their effectiveness relies heavily on the user-provided directives.

In the rest of this paper, Section 2 discusses fundamental issues for automatic parallelization of scientific and engineering applications. Section 3 presents the structure of the APE, which integrates our strategies for the fundamental issues. In Section 4, we summarize our preliminary results in parallelization of loops with uniform and non-uniform data dependencies, and redistribution of irregular data sets. Section 5 concludes with discussions about the future work.

2 Issues in Automatic Parallelization

For automatic parallelization of scientific and engineering applications on parallel machines, many issues need to be addressed. Below we identify three crucial issues for achieving high performance: *parallelism exploitation*, *data distribution*, and *communication scheduling*.

2.1 Parallelism exploitation

A major task of parallelizing compilers is to discover maximum parallelism in loops. In the literature, there are a large number of parallelizing techniques for cross-iteration independent loops, and for loops with uniform and static cross-iteration dependences. For *non-uniform* nested loops, like Loop 1(a), with non-constant cross iteration dependencies, however, few efficient exploitation techniques are available. Non-uniform loops appear quite frequently in real applications. In an empirical study, Shen *et al.* observed that nearly 45% of two dimensional array references are coupled [6]. Coupled references tend to lead to non-uniform dependences.

Loop 1(a): Loop with non-uniform dependences

```
DO I = 1, 10
    DO J = 1, 10
        A(2*J+3,I+1) = ......
        ...... = A(2*I+J+1,I+J+3)
    ENDDO
ENDDO
```

Loop 1(b): Loop with indirect access patterns

```
DO I=1, 10
    A(u(I)) = ...
    ... = A((v(I)) + ...
ENDDO
```

In scientific and engineering applications, another important class of potentially parallelizable loops are *irregular* loops that have indirect access patterns. Under the edge-oriented representation of an unstructured and sparse finite element graph, a sequential CFD code has a typical irregular loop like Loop 1(b), where indirect arrays u and v store pairs of nodes of edges. The index array may also be changing over time. In MD codes, for example, an index array through which the physical values associated with atoms are assessed stores the non-bonded lists of all atoms within a certain cutoff range. The spatial positions of the atoms change after each simulation step, consequently the same set of atoms may not be included in subsequent time steps. Hence, a new non-bonded list must be generated. In order to realize the full potential of parallel computing, evidently, compile time analysis must be complemented by new methods capable of automatically exploiting parallelism at run-time.

2.2 Data distribution

In addition to parallelism exploitation, one more key decision that a parallelizing compiler has to make on distributed memory machines is how to distribute data across processors. Data distribution affects parallel efficiency to a great extent. In the case that loops are partitioned by the *owner-compute* rule (*i.e.*, iterations of a loop are assigned to the processor that owns the destination of operations), data distribution determines the computational workload of processors and the inherent interprocessor communication volume. Data distribution is also crucial on DSM machines due to the big difference between local and remote memory accesses.

Regular data distributions like block and cyclic distributions are appropriate for parallelization of the loops operating on dense and structured data structures in terms of the communication to computation ratio. They also simplify the management of parallelism. However, it may not be the case for loops over sparse and unstructured data structures. In CFD codes on an unstructured mesh, blocked partition of the mesh nodes and index arrays may produce extremely high interprocessor communication volumes because the labeling of the mesh nodes does not necessarily reflect the connectivity of the mesh. Distribution of irregular data structures thus needs special attentions in the parallelization of scientific and engineering applications.

Parallelization of a program consisting a sequence of loops requires to consider the issue of data distribution for multiple loops at a time. Consecutive loops might have different requirements for load distribution. An approach to resolving their conflicting constraints is to develop a complex distribution strategy across loops so as to compromise their requirements. The compromising approach is intractable when more than two loops in a program segment are counted on. An alternative is *data redistribution* at run-time. Each parallelizable loop in a program adopts its favorite data distribution. When consecutive loops impose conflicting requirements on their data distributions, run-time support systems are used to adjust distribution across loops. Data redistribution incurs non-negligible overhead, even though interprocessor bulk data movement is commonly supported by most machines. When and how to redistribute data become challenging in the design of parallelizing compilers.

2.3 Communication scheduling

A major difference between compilation for shared-memory and message-passing models is that shared-memory model provides a global name space, so that any particular data items can be referenced by any processor using the same names as in the sequential code, even though the data may be physically distributed across processors. Shared memory model simplifies the design of parallelizing compilers. On DSM systems, however, shared memory synchronization primitives such as lock and barrier cause excessive amounts of communication due to the distributed nature of the memory system. Optimization performed by existing shared memory compilers are not sufficient for DSM systems.

On message-passing machines, in contrast, data set are distributed across processors, each of which has its own private address space. As a result, any reference, issued by a processor, to elements beyond its local partition is realized by explicit interprocessor communication. A simple-minded approach is to generate a message for each off-processor data item referenced. It usually produces extremely high communication overhead due to the startup cost per message transmission. Thus, there is a great need for compilers to include scheduling techniques for communication optimization. One of the main techniques is *message coalescing*. It tends to organize the communication of many data items destined for the same processor into a single message so as to reduce the number of message startups.

The communication optimization techniques discussed above are based on the assumption that the communication patterns are known *a priori*. Irregular loops with indirect access patterns, however, hide the data dependence among processors at compile-time. Parallelization of irregular loops, therefore, requires communication scheduling at run-time. The *inspector/executor* mechanism schedules interprocessor communication in the absence of compile-time knowledge [2].

3 Architecture of APE

Our implementation assumes a base architecture, as illustrated in Figure 1. It comprises a number of processing nodes connected by a scalable network, Each node is composed of one or more commodity processors with caches, memory, and a module of network interface (NI). Examples of the base architecture in-

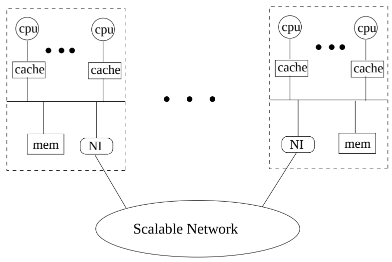


Figure 1: Targeted Base Machine of APE

clude cluster of workstations and distributed memory parallel machines, such as CM-5, Intel Paragon, IBM SP2, Cray T3D and Convex Exemplar. Centralized shared memory machines, like SGI POWERChallenge and SUN SPARCcenter, can serve as a processing node in the base architecture.

APE takes as input a sequential program and produces an output program targeting either shared-memory or message-passing programming models. Shared memory programming on distributed memory platforms are coming into practice. In addition, targeting message passing libraries PVM and MPI, the *de facto* standard libraries, APE ensures the portability of the resulting parallel codes. APE integrates compile-time analyses and parallelizing techniques and run-time scheduling support. Compile-time techniques for parallelization of non-uniform loops are encapsulated in an enhanced SUIF compiler, and run-time scheduling system is provided for parallelization of irregular loops, management of data redistribution, and communication optimization. In addition, a mechanism for on-line visualization of scientific and engineering data is also integrated for the purpose of practical use. APE comprises of six modules: xSUIF, SPMD-converter, data distribution data redistribution, monitor, communication scheduling, which are organized as in Figure 2.

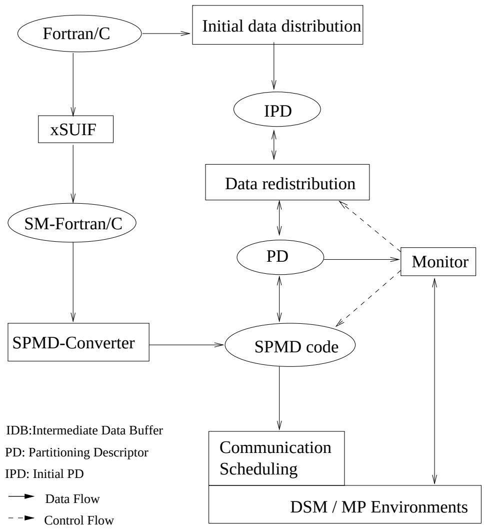


Figure 2: Architecture of APE

xSUIF, an enhanced SUIF analyzer, takes as input a sequential program in Fortran or C and produces an output program in the intermediate format of SUIF targeting the centralized shared memory model. In addition to the analyses and parallelization techniques embodied in SUIF, the xSUIF expands capabilities in handling loops with non-uniform memory access patterns and irregular loops that have input-data dependent memory access patterns. xSUIF compiler generates parallel codes in the master-slave paradigm. The execution of the program is controlled by a master thread. On arriving at a parallel loop, the master thread creates or activates a number of slave threads for cooperation. All slave threads join the master at the end of the loop.

SPMD Converter converts the master-slave parallel codes from xSUIF into SPMD (single-program and multiple-data) parallel codes targeting distributed memory machines. SPMD paradigm not only facilitates data locality exploitation, but also helps in reducing synchronization overhead. Furthermore, we distinguish shared memory and message passing models in the conversion, and construct two converters correspondingly. DSM-converter generates *multithreaded* parallel codes on DSM systems, and MP-converter generates PVM-based codes.

Initial data distribution is dedicated to irregular partition of sparse and unstructured data structures. Dense and structured arrays are regularly partitioned by xSUIF. Irregular partition takes as inputs the problem domain descriptor and topological information of the underlying machine, and generates an initial partition descriptor (IPD) for each processor. We separate the distribution policy and the mechanism and leave the policy open to external libraries.

Data redistribution redistributes the domain data set across processors at run-time for balancing processors' workloads and reducing interprocessor communication cost. The original data distribution can be regular or irregular. Redistribution of regular structures are implemented through dynamic programming approaches. Redistribution of irregular data structures takes its local partition descriptor (PD) (or IPD, initially) as input, and generates a new PD by communicating with other processors. Nearest neighbor iterative strategies will be implemented for ensuring the scalability of the system in MPP systems.

Monitor keeps track of the whole computation at run-time, and decides whether to invoke a redistribution operation according to the run-time execution information. Its decision can also be interactively steered by end-users.

Communication scheduling provides communication optimization techniques for both shared-memory and message-passing models. In the case that interprocessor communication patterns are unknown at compile-time, an inspector/executor mechanism is used in SPMD converter for determination of the communication pattern at run-time. This module is built on top of DSM systems and PVM environments, which subsumes the architecture details from system designers.

4 Preliminary results

4.1 Experiences on SUIF for regular computations

SUIF takes as input sequential program in Fortran 77 or C. The Fortran code is first converted to C, taking care to annotate the code so as not to loose information. The C code is then translated to an intermediate format. The intermediate format allows various passes to annotate the code further, to do dependence analyses and code transformation. The output of one pass can be given as the input of the next pass. The final pass converts the intermediate format back to C with calls to a runtime library. The library is machine specific, and the SUIF source provides support for parallel machines including Stanford DASH, SGI Challenge, and KSR.

Our initial evaluation of SUIF on regular data structures was conducted on a SUN symmetric multiprocessor SPARCserver 630MP running Solaris 2.4. We first ported the SUIF runtime library onto the machine targeting Solaris threads, so that parallel executables could be created.

We tested the effectiveness of SUIF in parallelizing matrix multiplications. Figure 3 plots the execution of the parallel code generated by SUIF for various matrix sizes on 4 processors. For comparison, the execution time of the codes generated by vendor-provided Fortran and C compilers and the hand-coded programs are also included. It can be seen that SUIF produces codes that are as efficient as manually-tuned codes. SUIF outperforms the F77 and C compilers because the codes by F77 and C are unable to take advantage of parallel computing.

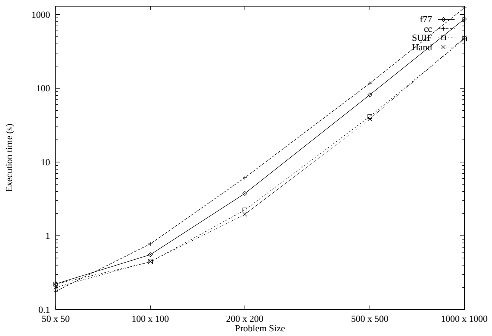


Figure 3: Execution times of the matrix multiplication codes generated by SUIF, Fortran and C compilers, manually

Using SUIF, we also parallelized several CFD test codes. The test codes have a parameter of matrix size reflecting the problem scale. Figure 4 plots the speedup of the first test code generated by SUIF using various number of threads. It can be seen that SUIF is able to exploit sufficient parallelism in the program of interest and provides good results with its run-time library for small scale problems. The degradation of performance for large scale problems is partly due to inappropriate data distribution across processors. Detailed analyses of the causes of the performance loss are presented in [3].

4.2 Analysis of non-uniform data dependences

In the literature, there are a number of methods based on integer and linear programming techniques for analysis of non-uniform data dependences. A serious disadvantage with these techniques is their high time complexity. To analyze the cross-iteration dependencies for these loops, we apply results from classical convex theory and present simple schemes to compute the dependence information.

A set of *diophantine equations* is first formed from

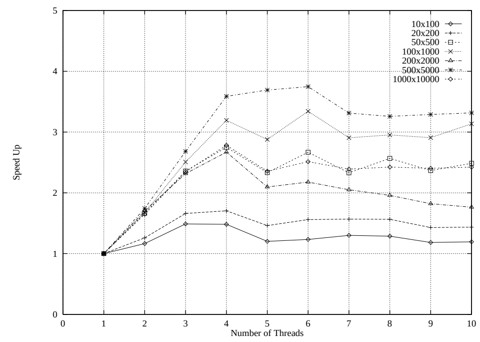
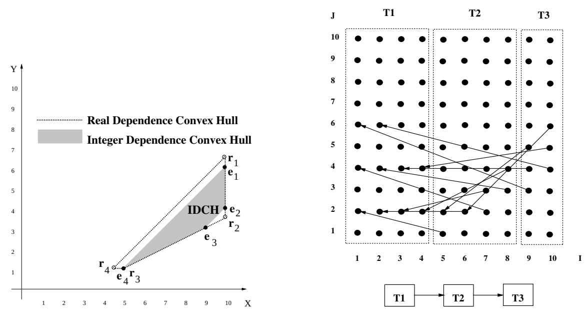


Figure 4: Performances of an automatic parallelized CFD Test Code using SUIF

the array subscripts of the nested loops. These diophantine equations are solved for integer solutions. The loop bounds are applied to these solutions to obtain a set of inequalities. These inequalities are then used to form a *dependence convex hull* (DCH) as an intersection of a set of halfspaces. Since the extreme points of a DCH are formed by intersections of a set of hyperplanes, they might have real coordinates. They need not be valid iterations. We developed an algorithm to convert these extreme points with real coordinates to extreme points with integer coordinates without losing useful dependence information [5]. We refer to the convex hull with all integer extreme points as *Integer Dependence Convex Hull* (IDCH). Figure 5(a) shows the DCH and IDCH for the nested loops in Loop 1(b). As can be seen from Figure 5(a) the IDCH is a subspace of DCH. So it gives more precise dependence information. We are interested only in the integer points inside the DCH.



(a) (b)

Figure 5: (a) DCH *and* IDCH (b) minimum dependence distance tiling for Loop 1(a)

Every integer point in the convex hull corresponds to a dependence vector in the iteration space. The dependence vectors of the extreme points form a set of extreme vectors for the dependence vector set. The minimum and maximum values of the dependence distance can be found from these extreme vectors. In [5] we proposed a number of theorems and schemes to compute the minimum dependence distance. Using the minimum dependence distances, we can tile the iteration space. The tiles are rectangular shaped, uniform partitions. Each tile consists of a set of iterations which can be executed in parallel. The minimum dependence distances can be used to determine the tile size. The minimum dependence distance of the program segment in Loop 1(a) d_{min} is equivalent to 4. So, we can tile the iteration space of size $M*N$ with d_{min}=4 as shown in Fig. 5(b). The number of tiles in the iteration space can be given as $T_n = \lceil \frac{N}{d_{min}} \rceil$ except near the boundaries of the iteration space, where the tiles are of uniform size $M*d_{min}$. Ignoring the synchronization and scheduling overheads, the speedup can be given as $Speedup = \frac{M*N}{T_n}$. Parallel code to synchronize the execution of the tiles is given in Loop 2.

Loop 2: Parallel loop from Loop 1(b)

```
Tile num T_n = ⌈N / d_imin⌉
DOserial K = 1, T_n
    DOparallel I = (K-1)*d_imin+1, min(K*d_imin, N)
        DOparallel J = 1, M
            A(2*J+3,I+1) = ...... ;
            ...... = A(2*I+J+1,I+J+3);
        ENDDOparallel
    ENDDOparallel
ENDDOserial
```

Most of the existing techniques cannot parallelize nested loops with non-uniform dependencies. Though some advances have been made to solve this problem, the amount of parallelism that the existing techniques extract is very low compared to the available parallelism. Our minimum dependence distance tiling extracts maximum parallelism and gives significant speedup compared to the existing techniques. For Loop 1(a), our method gives a speedup of 40 whereas the dependence uniformization method presented by Tzen and Ni [8] gives a speedup of 5. A detailed comparative analysis is presented in [5].

4.3 Scalable algorithms for irregular data redistribution

Parallelization of scientific and engineering applications first requires to distribute the problem domain across processors. Figure 6 is a distribution of a sample data set across 64 processors. Generally, discovering an ap-

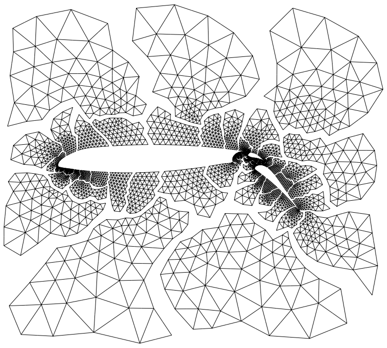


Figure 6: A distribution of an unstructured mesh around air-foil across 64 processors

propriate distribution for large unstructured meshes is time-consuming. It is a serial bottleneck in the overall course of a parallel computation. The bottleneck becomes severe with the increase of the scale of parallel computers. Being able to perform domain partitioning in parallel is essential not only for reducing the preprocessing time, but also for handling load imbalance at run time.

We proposed a nearest neighbor iterative strategy, called Generalized Dimension Exchange (GDE), for both domain mapping and re-mapping in parallel [9]. Our strategy starts from a simple domain distribution, and then iteratively refines the domain partition through local balancing at each processor within its immediate neighbors with respect to both the computational workload and interprocessor communication cost. For domain re-mapping, the nearest neighbor iterative refinement works directly on the original domain distribution. Figure 7 presents an illustration of redistribution of workloads along each border of a subdomain. Since the strategy is fully distributed, it can be easily

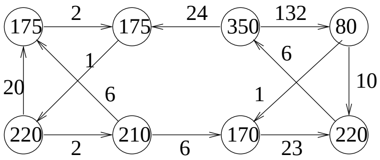


Figure 7: Illustration of remapping along each edge of a computational graph

scaled to operate in massively parallel computers of any size, and would tend to preserve the communication locality. More importantly, its overhead is proportional to the degree of imbalance of the original distribution.

Since the workload distribution mostly changes in an incremental fashion in the adaptive data parallel applications, our strategy is especially suitable for incremental re-mapping.

We experimented on the GDE-based algorithm in parallel distribution of unstructured meshes. The meshes in consideration are first partitioned using a recursive orthogonal bisection approach. The subdomains are next assigned onto processors using a simple Bohkari's algorithm. We then do mapping refinement according to the geometric information of vertices with respect to the *cut size* (the number of cut edges) and the shape of subdomains. The mapping quality and efficiency of our parallel domain mapping mechanism, together with those from other well-known sequential algorithms, are presented in Figure 8. The sequential algorithms include the KL method, the Inertial method(IN), the Spectral bisection(SB) , and the Multilevel(ML), which are available in Chaco library [4]. All sequential measurement were taken on Sun Sparc 10/50 with 96 MB memory. The parallel algorithm runs on a network of T805 Transputers. The computational power of a Transputer is about 17 times slower than the Sparc machine. The computed timing performances in the figure are scaled to take this into account. From the figure, it can be seen that the GDE-based mapping outperforms other geometric decomposition strategies (KL and Inertial methods) in quality. It generates slightly worse distributions than sequential spectral-based algorithms, but at very low cost.

5 Concluding Remarks

In this paper, we have presented the design and evaluation of a compiler system, called APE, for automatic parallelization of scientific and engineering applications on distributed memory computers. APE integrates compile-time and run-time parallelization techniques focusing on non-uniform loops with coupled subscripts and irregular loops with indirect memory access patterns. APE is built on top of SUIF. It extends SUIF with capabilities in parallelizing non-uniform, and in handling irregular loops together with a run-time scheduling library. We have evaluated the effectiveness of SUIF with several CFD test codes, and found that SUIF handles well uniform loops over dense and regular data structures. We have proposed an effective approach for parallelizing loops with non-uniform dependences. We have also presented a class of scalable algorithms for parallel distribution and redistribution of unstructured data structures during parallelizing irregular loops.

This is an on-going project. Future work include integration of the non-uniform dependence analyses techniques into SUIF, extension of the SUIF run-time support with functions for irregular data redistribution and communication scheduling, and comprehensive evaluation of the APE based on the enhanced SUIF on parallelizing full scale CFD and MD codes.

Meshes	KL	IN	IN+KL	SB	SP+KL	ML + KL	GDE
grid2	357 (0.96)	432 (0.10)	330 (1.02)	373 (5.42)	328 (6.61)	317 (2.89)	379 (0.48)
airfoil1	860 (0.48)	503 (0.20)	400 (0.86)	372 (21.29)	409 (21.20)	325 (2.98)	382 (1.2)
crack	1738 (1.75)	797 (0.45)	639 (2.03)	671 (74.14)	577 (82.60)	615 (6.47)	660 (2.77)
big	1875 (2.00)	1219 (1.05)	995 (3.31)	863 (124.8)	675 (131.34)	605 (8.05)	913 (1.29)

Figure 8: Comparison of various algorithms in terms of cut size and running time (in brackets) for mapping various unstructured meshes onto 16 processors.

References

[1] P.Banerjee, J. A. Chandy, *et al.*, "The Paradigm compiler for distributed-memory multicomputers", *IEEE Computer*, Oct. 1995, page 37–47.

[2] H. Berryman, J. Saltz, and JJ. Scroggs, "Execution time support for adaptive scientific algorithms on distributed memory machines", *Concurrency: Practice and Experience*, 3(3):159–178, June 1991.

[3] V. Chaudhary, *et al*, "Evaluation of SUIF for CFD on Shared-Memory Multiprocessors", *First SUIF Compiler Workshop*, Stanford University, Jan. 1996.

[4] B. Hendrickson and R. Leland. The chaco user's guide. Technical Report Tech. Rep. SAND 93-2339, Sandia National Lab., USA, 1993.

[5] S. Punyamurtula and V. Chaudhary, "Minimum dependence distance tiling of nested loops with non-uniform dependences," in *Proceedings of the Sixth IEEE Symposium on Parallel and Distributed Processing*, (Dallas, Texas), IEEE Computer Society Press, October 1994.

[6] Z. Shen, Z. Li, and P.-C. Yew, "An empirical study on array subscripts and data dependencies," in *Proceedings of the International Conference on Parallel Processing*, pp. II–145 to II–152, 1989.

[7] B. Silson, *et al.*, An Overview of the SUIF Compiler System. available on *http://suif.stanford.edu/suif*

[8] T. H. Tzen and L. M. Ni, "Dependence uniformization: A loop parallelization technique," *IEEE Transactions on Parallel and Distributed Systems*, vol. 4, pp. 547 to 558, May 1993.

[9] C. Xu, B. Monien, *et al*, "Nearest neighbor algorithms for load balancing on parallel computers", *Concurrency: Practice and Experience*, Oct. 1995.