The Design and Evaluation of "CAPTools" — A Computer Aided Parallelization Tool-kit

Authors

Jerry Yan, Michael Frumkin, Michelle Hribar, Haoqiang Jin, and Abdul Waheed

MS T27A-1, NASA Ames Research Center

Moffett Field, CA 94035-1000, USA

Steve Johnson, Mark Cross, Emyr Evans, Constantinos Ierotheou, and Pete Leggett

Parallel Processing Research Group,

University of Greenwich,

London SE18 6PF, UK

All correspondences should be addressed to Dr. Jerry Yan, 2 Pentland Gardens, London, SW18, 2AN (I am currently on sabbatical in London). E-mail: yan@nas.nasa.gov, Phone: +44-181-331-8588; FAX: +44-171-371-0021

The Design and Evaluation of "CAPTools" — A Computer Aided Parallelization Tool-kit

Jerry Yan, Michael Frumkin, Michelle Hribar, Haoqiang Jin, and Abdul Waheed

MS T27A-1, NASA Ames Research Center
Moffett Field, CA 94035-1000, USA

Steve Johnson, Mark Cross, Emyr Evans, Constantinos Ierotheou, and Pete Leggett

Parallel Processing Research Group,
University of Greenwich,
London SE18 6PF, UK

Key Words. Automatic Parallelization, Software Tools, Performance Evaluation

Abstract. *Writing applications for high performance computers is a challenging task. Although writing code by hand still offers the best performance, it is extremely costly and often not very portable. The Computer Aided Parallelization Tools (CAPTools) are a toolkit designed to help automate the mapping of sequential FORTRAN scientific applications onto multiprocessors. CAPTools consists of the following major components: an inter-procedural dependence analysis module that incorporates user knowledge; a "self-propagating" data partitioning module driven via user guidance; an execution control mask generation and optimization module for the user to fine tune parallel processing of individual partitions; a program transformation/restructuring facility for source code clean up and optimization; a set of browsers through which the user interacts with CAPTools at each stage of the parallelization process; and a code generator supporting multiple programming paradigms on various multiprocessors. Besides describing the rationale behind the architecture of CAPTools, the parallelization process is illustrated via case studies involving structured and unstructured meshes. The programming process and the performance of the generated parallel programs are compared against other programming alternatives based on the NAS Parallel Benchmarks, ARC3D and other scientific applications. Based on these results, a discussion on the feasibility of constructing architectural independent parallel applications is presented.*

1. Introduction

1.1. Motivation

Over the past decade, high performance computers based on commodity microprocessors have been introduced in rapid succession from at least seven vendors/families. All of them supported some form of message passing libraries, while the latest players in the market also supported

some form of distributed share memory remote access primitives[1]. Many users have also constructed computing clusters using workstations and PC's. They re-wrote their applications using message-passing libraries (such as MPI and PVM) and reported very good price/performance numbers.

Nevertheless, these advances have created a new class of problems involving multiple code versions. If a computer center were to procure only a quarter of these machines over the past ten years, each machine would last no more than three years on average. In fact, a few such architectures would often coexist at a site simultaneously. The average user would have to struggle with two issues every time a new machine is introduced:

1. "Should I port my application to the new machine?" The user really has no choice but to do so because the old machines will be decommissioned. Furthermore, there is some pressure to attempt to fully utilize the computing performance potential of these new machines.
2. "How many versions of the source code should now be maintained?" Even though this depends on the user base of the application, the user may still have to maintain three code versions supporting: message passing standards, shared memory directives as well as vector processing.

Unlike scientists in the research laboratories, users in the commercial sector (e.g., aircraft industry) have remained dependent on traditional vector architectures (e.g., Cray C90) because of two reasons:

1. They are unwilling to abandon their trusted applications and develop new ones for the new machines. These "legacy" applications have proven their worthiness well over many years on specific architectures. The amount of time and investment required to validate complete new applications is prohibitive.
2. They are unwilling to port (or rewrite) these trusted "legacy" applications onto the new machines. Porting large applications is very expensive and time consuming. Furthermore, there is no guarantee that their investment can be protected because of the short life time these multiprocessors may exhibit.

Even more recently, a large effort has been expended in the development of underlying infrastructure to support the creation of wide area networks of computers, including large-scale machines, to enable user access to them as a single computing resource. Projects such as NPACI [1], and

---

[1] Examples included: Intel's IPSC/860, Delta and Paragon: all supported the NX message-passing library; TMC's CM-2, and CM-5 (which also operated in MIMD mode supporting an active messaging library called CMMD); IBM's SP1 and SP2: both support MPI and (IBM's propriety) MPL libraries; Cray's T3D and T3E support both MPI as well as remote memory access; and SGI Origin 2000 and Sun's Enterprise and HPC2 servers supports both MPI as well as share memory messaging.

PACI [2] have aimed to provide users transparent access to such a "computational grid", which is essentially a distributed, heterogeneous collection of parallel computers. Such grids pose many new requirements with respect to programming models, compilation strategies and execution environments. With increasing application complexity and problem size, scientists may no longer be able to afford to continue porting efforts every time a new machine is launched. Furthermore, with the anticipation of the decommissioning of old machines, as well as demands for shorter execution time, alternative approaches to porting by hand must be investigated. In other words, the production of efficient "architecture independent" parallel programs has become the next big challenge in high performance computing.

1.2. Outline of the Paper

In Section 2, we first discuss the alternative approaches to parallelizing and maintaining legacy applications. These include programming by hand, relying on parallelizing compilers supplied by the vendor, annotate/rewrite application using data- and task- parallel directives/languages, as well as rewriting application codes using semi-custom building blocks (or libraries). Given the state of the art, an interactive toolkit for parallelization seems to be most fitting because the user can supply his knowledge and influence the parallelization strategy while leaving the mundane error-prone program transformation process to the toolkit. The design philosophy and architecture of CAPTools are presented in Section 3. Implementation strategies that enable high performance code optimization is then presented. These include: use of well-know techniques to handle computations involving structured- and unstructured-meshes, techniques for dependence analysis and data partitioning, parallel execution control determination, as well as communication identification, migration and merger. The usage of CAPTools as well as the performance of the generated code is presented and evaluated in Section 4. Results obtained from a detailed study based on NAS Parallel Benchmarks indicate that CAPTools generated code that performs within 10% of hand-written code. Finally, Section 0 presents a conclusion and a brief discussion of future work.

2. A Spectrum of Alternatives for Legacy Code Modernization

This section presents a state-of-the-art survey for these approaches and evaluates their suitability for parallelizing legacy applications and their long-term maintenance.

2.1. Rewriting Applications by Hand

Although writing parallel programs by hand gives the best performance, it also requires the most significant amount of effort. When a user rewrites a sequential program using message passing (or remote memory access) primitives, they have complete control over data distribution and parallelization strategies. Given sufficient feedback, the user can tune the program by improving load balancing, minimizing data redistribution, and overlapping communication with computation to minimize processor idle time. The major disadvantage with this approach is the immense re-

sponsibility of ensuring the correctness of the implementation that comes from the user's explicit management of, for example, domain decomposition. This use of Single-Program-Multiple-Data (SPMD) paradigm requires data to be distributed in a consistent fashion across the multiprocessors. Furthermore, communication must occur whenever a processor modifies a value required by another processor. This update schedule must be performed in the right order and with maximum efficiency to ensure correctness and performance. Message-passing programs, therefore, cannot be developed gradually by piecemeal conversion of a serial code. The entire program must be converted all at once, putting message-passing at a distinct disadvantage compared to the shared-memory paradigm, which at least, gives the appearance that it allows gradual parallelization via insertion of parallelizing directives. Finally, as outlined in the Section 1.1, the anticipation of repeating this expensive, tedious, time consuming, error prone process every few years makes this approach very unattractive.

2.2. Parallelizing Compilers with Directives

Some users may opt to rely on parallelizing compilers provided from the vendor. They anticipate that the insertion of a few "directives" together with a small amount of re-writing would enable their sequential program to perform well on these new machines[2]. The success of this approach depends on three important factors:

1. THE LEVEL OF SOPHISTICATION OF THE COMPILER. The user is completely dependent on the compiler's ability to discover parallelism, distribute data and accurately detect data dependencies. The performance obtained depends on many factors, including thoroughness of the interprocedural analysis as well as the ability to consider user knowledge about, for example, application input parameters.
2. THE STRATEGIC PLACEMENT OF PARALLELIZATION DIRECTIVES. These directives take the form of structured comments, they are ignored by non-parallelizing compilers. When no directives are given, some parallelization or vectorization may still occur if the source code is simple enough. The user may supply directives either to override data dependencies the compiler failed to disprove or to enforce certain data placement strategies. Either way, it requires a level of expertise not commonly possessed by application scientists.
3. USER'S ABILITY TO TUNE THE APPLICATION. Parallel code execution may not necessarily improve performance. For example, loops can be interchanged to increase the grain-size of individual parallel tasks to reduce tasking and communication overhead. The user must be prepared to iteratively inspect performance data and modify the program accordingly.

---

[2] Vectorization and multi-tasking compiler directives have been defined and supplied by vendors such as Cray Research [3], Advanced Parallel Research (FORGE) [4], Kuck and Associates (KAP/Pro Toolset) [5] and, Silicon Graphics (MIPSpro Power Fortran (X3H5 compliant) and C (pragma-based directives)) [6].

Furthermore, simple directives, such as `$DOACROSS`, do not allow parallelism to be conveniently specified across loop nests. For example, one of the most useful control structures in parallel programming, the pipeline, cannot be expressed without making the domain decomposition explicit. This, compounded with a lack of control over data placement, may lead to severe performance limitations. Finally, the use of parallelizing compilers suffers the same constraints as vectorizing compilers in that it is only applicable on the multiprocessor for which the compiler was designed. Nevertheless, the development of standards (such as OpenMP [7]) cannot be simply overlooked. It offers a level of abstraction that may survive architectural evolution as well as a goal towards which compiler writers may work.

2.3. Data and Task Parallel Languages

A third option involves the annotation of a sequential program using data distribution directives offered in HPF [8] and related projects[3]. While the standardization of HPF implies portability, its performance as well as its applicability over a wide range of scientific applications is still very much in question. Users have reported that HPF programs run at least 2 times slower than their message passing counterparts (even on one processor) [17]. Possible explanations include bookkeeping overheads as well as the lack of control for communication granularity [18]. Delivering a portable but slow parallel program defeats the purpose of high performance computing. Although HPF is simple and elegant, it is also limited in its ability to express parallelism. The proposal of a follow-on standard, HPF-2, will cater for better handling of indexed arrays and task parallelism. Unfortunately, this presents an even more daunting task before the compiler writers. In the final analysis, inserting data partitioning directives manually in a large and complex application is not straightforward. Until computer aided analysis [19, 20] could provide some guidance in this process, modernizing legacy codes via rewrite using HPF is not a viable option even if HPF performance were reasonable.

Fx [21] and Fortran M [22] are two of the few[4] FORTRAN-based task parallel languages aimed at supporting multi-disciplinary applications. Disjoint tasks may execute concurrently and communicate[5]. There is no support for global (shared) data types within the tasks other than in the data-parallel sense. Just as in the 'bare' message-passing environment, the user is responsible for interpreting the meaning of arrays owned by the individual processors.

In summary, parallelizing legacy applications using data- or task-based parallel languages implies a major re-coding effort. Research is still being carried out today to produce a language standard

---

[3] HPC++ [9], Vienna Fortran [10], C*[11], Annai [12], CM Fortran [13], PC++/Sage++ [14], Fortran D [15], and Mentat [16].

[4] Most of the proposed are not FORTRAN-based. These include Shared Data Abstractions (SDA's) [23], Sisal [24], and Split C [25].

[5] input and output mapping directives in Fx; channels and ports in Fortran M

that is both flexible in its ability to express parallelism and at the same time, allows efficient compilation to take place so as to generate high performance programs.

2.4. Using Semi-Custom Building Blocks (or Libraries)

Ideally speaking, the user should not be forced to choose between flexible task parallelism (with private, non-shared data types) and the convenience of using shared data types in data languages. Parallel libraries accomplish this encapsulation by supporting atomic tasks on globally defined, shared data types. The implementation of the library can be tailored to individual multiprocessor architecture and be made hidden from the user. This offers a degree of portability as the library writers are responsible for staying on top of evolving machine architecture. Nevertheless, opting for libraries represents a compromise, since no library can be completely general-purpose. For example, ScaLAPACK [26], a distributed-memory version of LAPACK, only supports those problems that can be cast as numerical linear-algebra problems. Unfortunately, numerical linear algebra problems are often embedded in larger applications. For instance, a finite-element code may generate the stiffness matrix using a three-dimensional (3-D) block decomposition while the resulting equation can only be solved in ScaLAPACK using a two-dimensional (2-D) decomposition. Remapping in this case will require a costly global exchange operation. Structured-grid applications that do not construct system matrices explicitly will not benefit from ScaLAPACK. Many special-purpose parallel packages have also been developed. PETSc [27], for example, offers a set of functions for manipulating[6] high level distributed data types, and a collection of linear and nonlinear equation solvers. However, several data partitioning strategies and remote access mechanisms[7] required in complex CFD production codes are not supported.

In summary, parallel libraries derive their utility from execution efficiency, combined with their ease of use. While customization improves user convenience, it sacrifices generality and expandability for efficiency and simplicity. In principle, some legacy codes may be re-written using parallel libraries. The amount of rewriting required and the performance of the resultant code is yet to be determined. A detailed survey of the state-of-the-art developments in parallel library projects can be found in [28].

---

[6] Distributed data types are created collectively, but may be manipulated collectively as well as individually. One-, two- and three-dimensional distributed arrays are used to support structured-grid computations. Provisions are made for overlap zones (ghost points) that can act as buffers for copies of data elements on geometrically neighboring processors. With the proper use of the assembly routines, it is possible, in principle, to program pipeline control structures explicitly, which has the advantage that the grouping factor is under the control of the user.

[7] PETSc only allows block-block distribution for its vectors and distributed arrays. There is no support for more advanced domain decompositions, such as multi-partitioning or dynamic decompositions, such as those required by transpose-based parallel algorithms. There is also no support for non-neighbor communications.

2.5. Towards Computer Aided Parallelization Tools

Table 1 presents a concise comparison of the aforementioned approaches available for parallel programmers. Basically, message-passing codes produced by hand exhibit the highest performance because the user can utilize their knowledge about the code to formulate a suitable parallelization strategy and tailor the implementation to the architecture. The use of libraries as well as data- and task- parallel languages reduces the user's effort by shifting the machine-dependent implementation details to compiler writers and library builders. Unfortunately, applications which do not fit into pre-defined computation models and templates offered in the language/library either cannot be implemented or must execute at a reduced level of performance. If portability were not an issue, machine-specific parallelizing compilers, combined with detailed profiling and user tuning produces acceptable performance for small codes. However, the need to limit compile time reduces the thoroughness in which inter-procedural dependence analysis could be applied, thus affecting the quality and granularity of the parallel code produced for complex applications. In light of these concerns, interactive parallelization tool-kits should be the most promising approach to be investigated to assist the production of architecture-independent codes.

Table 1. A Comparison of Various Approaches Available for Development and Maintenance of Parallel Applications.

Process	Time/Effort	Performance	Portability	Applicability/ Limitations
Rewritten by hand	Extensive code revision required; error-prone	Excellent when implementation is tailored to machine	Dependent on portability of standards (e.g. MPI, PVM); tuning required	Applicable to any code
Parallelizing compilers	Minimal code modification; directives inserted as needed; tuning could be time consuming	Completely dependent on compiler	Dependent on portability of standards (e.g. OpenMP); not portable to network of workstations to-date;	Performs well for codes with simple structure and loop-level parallelism
Data and task parallel language	Annotation required; need code restructuring to match language's programming paradigm	Compiler performance still in question to-date	Dependent on portability of standards (e.g. HPF)	Does not handle unstructured meshes and computational pipelines efficiently
Parallel libraries	Selective replacement of code sections with library calls; restructuring may be required to fit library structure	Good when library is tailored to machine	Dependent on the machines to which the library has been ported.	Applicable to specific class of codes for which the library has been designed.

An interactive parallelization tool-kit provides a set of software tools to assist in the analysis, browsing, editing, and transformation of a serial source code to produce a parallel program. The

user operates at a higher level of abstraction, leaving the tool-kit to perform the mundane, error-prone operations required to realize a particular parallelization strategy specified by the user.

Two major toolkit projects have been undertaken at Rice University and the University of Vienna to support data parallel FORTRAN program. The D editor [29] (derived from ParaScope [30]) is a browser that is intended to help users to develop Fortran D programs by providing information at the source level. The latest version of the editor incorporates performance data gathered by Pablo [31] to guide the user in the parallelization process. The Vienna FORTRAN Compilation System (VFCS) [32] provides a comprehensive toolset to assist the user to tune the performance of Vienna FORTRAN programs. A follow-on effort, HPF+ [33], is being built and is specifically targeted to handle computations involving unstructured meshes. Unfortunately these toolkits provide no assistance for the conversion from sequential to data-parallel FORTRAN.

A parallelization package capable of starting from sequential source code is Forge Explorer [4]. The user participates in a step-by-step dialogue to initiate data dependency analysis, insertion of the proper control structures and message passing calls, to generate the final parallel program. While Forge Explorer can recognize recurrences that can be resolved using computational pipelines, it is limited in its ability to detect complex interprocedural dependencies and properly control the granularity of the parallel tasks.

Other efforts being carried out in the universities, such as Stanford's SUIF Explorer [34] (derived from SUIF [35]), Illinois's Polaris [36], and Rice's dHPF [37] can not yet handle large industrial-based codes. Based on out experience at NASA Ames Research Center, the only effort that can parallel large sequential codes is CAPTools, developed at the University of Greenwich.

3. The Design of Computer Aided Parallelization Tools

3.1. Design Criteria

Throughout the development of CAPTools, a number of vital criteria were specified to ensure that industrial and scientific applications can be effectively parallelized onto a wide range of parallel architectures. These criteria are:

- handle real world Fortran application codes regardless of their perceived "quality";
- no allowance for performance limitations of the generated parallel code due to automation;
- generate recognizable code following well-understood parallelization techniques; and
- generate portable code for as wide range of parallel systems as is feasible.

Each of these criteria has a number of implications to the design and implementation of CAPTools. The requirement to handle real world applications without restrictions on FORTRAN

standards[8] forces all stages of the parallelization process to cater for commonly used code features (such as inter-procedural dimension mapping of arrays). Being able to handle these coding features allows the vast catalogue of legacy applications to be parallelized. Additionally, and equally as important, it allows the users to continue coding without regard to later parallelization, so that they may concentrate on the scientific field in which they specialize. This is crucial since many of these application programmers only employ serial programming as a tool and have no interest in further diluting their effort by considering parallelization. Forcing them to "think parallel" may also not be desirable since it may not be practical to acquire the expertise required to produce efficient, scalable parallel code. To ensure the success of parallelization tools, it is necessary to embed the "expertise" in parallelizing application codes within them. This will also satisfy the criteria of not suffering any performance loss due to automation. This approach is only possible since the tools are designed to follow the recognized "best manual practice" that has already been demonstrated by many groups [38-44]. The development of the parallelization tools then becomes a process of abstracting and automating the manual parallelization process into a series of easily understood stages. These stages can then be used for user interaction to allow monitoring and some control of the parallelization process, as well as enabling the system to explain its decisions to the user in as understandable way as possible. As more application codes are encountered, the expertise in manual parallelization that is required to overcome previously unforeseen circumstances can be embedded within the tools, thereby increasing their power.

The use of well understood parallelization techniques coupled with minimal changes to the original source code during parallelization enable the recognition criteria to be achieved. The parallelization process can then be made to consist of a few straightforward transformations only. These include the addition of communications, the adjustment of loop limits and the addition of execution control masks where necessary, allowing the remainder of the code to be left unaltered. Recognition of the generated code is obviously essential if meaningful user interaction is to be maintained throughout the parallelization process. It is also vital if further manual tuning of the generated code is required and allows maintenance of the parallel code.

Finally, portability is achieved by generating code that uses simple generic communication calls to the CAPTools library CAPLib [45]. The porting of a parallel code to a new system then requires only that CAPLib has been ported onto that system, allowing recompilation to be sufficient. Since CAPLib includes generic MPI and PVM versions, this process is often straightforward. It also allows for the use of efficient lower-level communication API's such as the `Shmem` library on the Cray T3D and T3E systems without changing the application code.

---

[8] e.g., Fortran 77 or Fortran 90, including legacy codes that may have originated in Fortran IV etc.

During the development of CAPTools, a number of other features were identified as being essential and desirable. These came as a result of observations made while using the tools for real world application codes where, for example, the generated parallel code did not attain the high quality of the equivalent manually created parallel code. These features specifically support:

- intensive computation required in dependence analysis and other symbolic algebra processing;
- exploitation of user supplied information, in the form of simple constraints on the values of the input variables of the application code, in all stages of the parallelization;
- presentation of clear and concise questions, typically relating to input variables of the application code, to elicit essential information from the user; and
- browsers for each stage of the parallelization process to allow user inspection and to provide explanation of why decisions were made.

These features aim to provide a simple interface between the user and the tools enabling the exploitation of user knowledge in terms that relate to their application. The emphasis on accurate dependence analysis alleviates the user from the need to prescribe parallelism throughout the application. User interaction is maintained at a higher level, typically this is minimal and relates only to those areas of an application code where, for example, the complex nature of the code prevented accurate analysis.

3.2. "Best Manual Practice" Parallelization Strategies

To achieve the specified goals, work has focussed on specific, but widely used, parallel programming techniques. The first two techniques considered are single block, structured mesh (regular) application codes, such as Control Volume based codes, and unstructured mesh (irregular) application codes, such as Finite Element codes. Figure 1a and Figure 1b show structured and unstructured meshes and the related data partitioning techniques. In both cases, the core cells/elements are owned by the processors to which they are allocated with the overlap (also known as halo) cells being owned by other processors. The basic "owner compute" strategy is then used so that each processor performs computations relating to owned data only. The overlap data is updated by messages sent from the owning processor to the using processor.

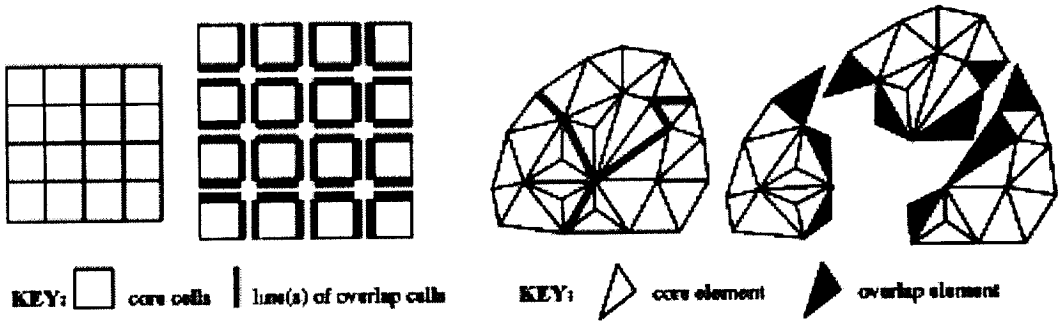

Figure 1. a) Structured Mesh Decomposition b) Unstructured Mesh Decomposition.

Both techniques are implemented to allow runtime determination of data partition details after problem details and processor topology have been specified. Although the techniques have many similarities, their implementation into application codes requires different approaches.

3.2.1. Structured Mesh Parallelizations

The runtime calculation of processor limits for the data partition is a simple process for structured mesh codes. Basically, it involves dividing the number of cells in each partitioned mesh dimension by the number of processors the user has requested for that dimension. To allow this flexibility, partition ranges are represented by *low* and *high* range variables on each processor (`CAP_L` and `CAP_H` respectively) indicating the range of owned data on that processor. The remainder of the parallelization process then consists of:

- adding execution control masks to determine if a processor should execute an instance of a statement,
- adjustment to loop limits to only cover the owned range of data, and
- adding communications to update overlap areas.

An execution control mask for such a parallelization is used to determine if the computation relates to the assignment of data owned on this processor, i.e.

```
IF (I.ge.CAP_L.and.I.le.CAP_H) A(I) = . . .
```

Similarly, alterations to loop limits involves the use of the partition range variables, also taking into account the original loop limits to ensure that only data processed in the original loop is processed in the parallel version:

```
DO I = MAX(2,CAP_L) , MIN(NI-1,CAP_H)
```

The communications used to update the overlap areas use the CAPLib routine `CAP_EXCHANGE` so that all processors can update one of their overlap areas in a single call:

```
CALL CAP_EXCHANGE(A(CAP_H+1),A(CAP_L),1,CAP_TYPE,CAP_RIGHT)
```

where CAP_TYPE indicates the data type involved in the communication and CAP_RIGHT indicates the processor to be communicated with (in this case, each processor's 'RIGHT' neighbor).

Reducing the memory requirement so that each processor holds only its core and overlap areas can then be implemented by adjusting declarations and any necessary array references [46].

3.2.2. Unstructured Mesh Parallelizations

The runtime calculation of the data partition for unstructured meshes is more complex. A graph partitioning tool such as JOSTLE [47] or MeTis [48] is used to process a graph representing the topology of an unstructured mesh and return a processor ownership array (CAP_P), relating each mesh element to a processor. The processor ownership array is then used to enforce execution control masks, taking the form:

```
IF (CAP_P(I).eq.CAP_PROCNUM) A(I) = . . .
```

where CAP_PROCNUM is a unique number identifying the executing processor. Updating overlapped elements on each processor is performed using a communication that collects data to be sent before transmission with all received data being unpacked. These communications follow pre-described communication sets that are calculated based upon the mesh interconnections. A typical manual parallelization would use an unstructured mesh library (for example CAPLib [45], or PARTI [49]) requiring a "standard" data structure that will very often be different to that used in the application. To overcome this, manually introduced loops are typically used to build a runtime graph of the mesh in the data structure required by the library routines, where these loops are in-effect "inspector" loops.

The use of local mesh renumbering reduces memory requirements by only storing core and overlap data. It also has the side effect of making many loops run over the locally owned set [50]. This is achieved by changing loop limits to be based upon the number of locally owned elements, enabling any execution control masks within those loops to be removed. The resulting code can then be made to resemble the original serial code very closely.

3.2.3. Abstraction of the Manual Parallelization Techniques

The automation of these techniques can be broken down into a series of stages where each stage is relevant to both structured and unstructured mesh codes. These stages are:

- dependence analysis;
- determination of the arrays to be partitioned and the components to be decomposed;
- addition of execution control masks;
- identification and addition of communication statements; and
- adjustments to implement memory reduction.

Although the details of many of these stages differ between the two types of code, the user process and associated browsers in CAPTools are the same for both. Since the details of the partitions are only determined at runtime, the automated parallelization processes is carried out based on symbolic variables for both the partition ranges in structured mesh codes (`CAP_L` and `CAP_H`) and also for the processor ownership array in unstructured mesh codes (`CAP_P`).

The organization of CAPTools components is shown in Figure 2. There are six main components:

1. an inter-procedural dependence analysis module capable of incorporating user knowledge;
2. a "self-propagating" data partitioning module driven via user guidance as well as data dependencies;
3. an execution control mask generation and optimization module for the user to manage and fine tune parallel processing of individual partitions;
4. a code generator supporting multiple parallel programming paradigms on various multiprocessing platforms;
5. a program transformation/restructuring facility for source code clean up and optimization; and
6. a set of browsers through which the user may guide the parallelization process based on feedback of the intermediate results at each stage.

The first four components represent the aforementioned stages in which a code is parallelized. Iterative cycles are present at each stage before the final parallel version is produced. In the analysis stage, a basic dependence analysis (involving standard techniques found in "parallelizing compilers") may first be applied to produce an intermediate result for the user. At this point, the user may supply some information and opt to carry out a more extensive and time-consuming analysis to yield a more accurate representation of the dependencies within the code. The code partitioning stage is also iterative; involving refinement of the initial data partition strategy; changing the criteria for data partition inheritance; and performing incremental partitioning. The code generation stage will almost certainly undergo some iterative cycle to refine or optimize the placement of execution control masks [51] on statements and the generation of communication calls. Code optimization forms an obvious step in the successful parallelization of a code. Some optimizations are currently identified and performed by CAPTools, others are currently carried out by hand with (semi-) automation within CAPTools under development.

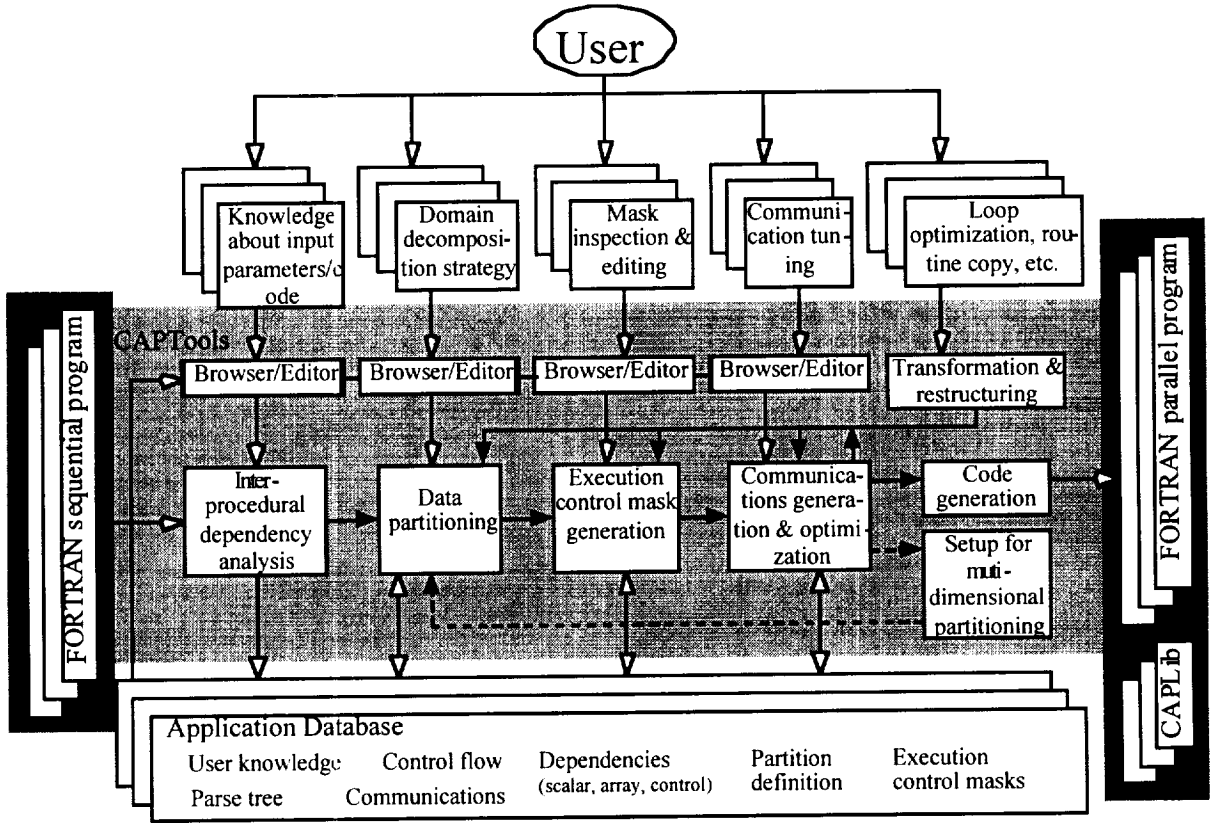

Figure 2. Overview of Computer Aided Parallelization Tools

3.3. Dependence Analysis Techniques Required to Achieve the Design Criteria

The criteria set for CAPTools placews a considerable burden on the dependence analysis phase of the parallelization process. All the following phases in the parallelization are fundamentally dependent on the accuracy of the dependence graph, where inefficient execution control, unnecessary communications and communications generated within loop nests are all potential consequences of deficiencies in this analysis. The criteria of efficient parallel code forces the computational effort required in dependence analysis to be considerable for complex codes. This perceived negative aspect of the tools, however, can be justified in the context of a parallelizing tool where, unlike in a compiler, it is "user time" and not "compute time" that is crucial. Since the user's task is considerably simplified by an accurate dependence analysis, the running of, for example, an overnight dependence analysis is quite justifiable if the parallelization can then be completed in a short space of time thereafter.

The combination of a number of traditional techniques [52, 53] with a set of new techniques aimed at the accurate analysis of complex real world applications, provides a powerful dependence analysis. The key features of this analysis are that it is symbolic, interprocedural and value-based to accurately capture data flow throughout the application. The symbolic processing involves a range of symbolic algebra techniques including GCD [53], SIDA [54], the symbolic

Omega test (SOmega [55]) with substitution of "symbolic constraint sets" through conditional branches [54]. The interprocedural array analysis uses a relatively cheap array-section analysis [56] pre-test followed, if necessary, by a more accurate atomic-based test [54, 57] where all control information is also incorporated to enhance the quality of the solution. The value based nature of the analysis is achieved using covering set analysis [54] to determine if any data can flow between the assignment and usage of a dependence without being overwritten in intermediate assignments. The following examples, taken from real application codes, indicate the complexity of code encountered and illustrate how these techniques provide an accurate dependence analysis.

3.3.1. The Checkerboard Technique

The checkerboard technique is commonly used to allow an implicit solution algorithm over a mesh to be performed in a few passes where all computations are independent within each pass (i.e. no recurrences). Although typically used to allow the exploitation of vectorization and parallelism, they represent a challenge to dependence analysis in the proving of the parallelism within each pass. A simple example of a checkerboard code fragment is shown in Figure 3.

```
DO IC = 0,1
  DO J = 1,NJ
     ISTART = MOD(J+IC,2)
     DO I=ISTART+1,NI,2
        A(I,J) = 0.25*(A(I,J-1)+
&        A(I,J+1)+A(I-1,J)+A(I+1,J))
     ENDDO
  ENDDO
ENDDO
```

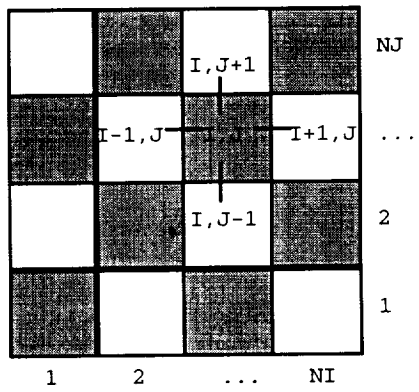

Figure 3. Checkerboard update scheme and pictorial example.

This is a two-color update scheme where the loop variable **IC** indicates which pass is being executed. When **IC** is zero, the **J** and **I** loops are parallel and the white cells in the diagram are updated. To prove this independence in dependence analysis requires symbolic algebra manipulation and a functional description of the intrinsic function used along with the accurate representation of integer divisions. The following is a brief description of the operation of these tests:

Test for dependence between iterations of **J** loop for the assignment **A(I,J)** to a later usage in the reference **A(I,J-1)** begins with a GCD [53] test. Considering index 1 adds the constraint $\mathbf{J_{new}-1} = \mathbf{J_{old}}$ as enforced by the necessary equality of index 2 (where $\mathbf{J_{new}}$ is the value of J in a later iteration than $\mathbf{J_{old}}$). The equation for equality of index 1 (where all DO loops have been normalized to run from 1 in steps of 1 in the analysis) is:

$$2I_{new}+\mathbf{ISTART_{new}} -1 = 2I_{old}+\mathbf{ISTART_{old}} -1$$

Substitution of variable **ISTART** with the **MOD** function gives:

$$2I_{new}+MOD(J_{new}+IC,2) = 2I_{old}+MOD(J_{old}+IC,2)$$

Applying the functional definition of **MOD** intrinsic function gives:

$$2I_{new}+J_{new}+IC-2\,\mathbf{(J_{new}+IC)/2} = 2I_{old}+J_{old}+IC-2\,\mathbf{(J_{old}+IC)/2}$$

Now the integer divisions can be removed by multiplying by denominators, introducing new variables to account for round-off. Since these are divisions by 2, the round-off must be either 0 or 1. Additionally, since we know that $J_{new}-1 = J_{old}$, then one numerator must be odd and the other even in the above equation. This allows the use of a single variable to represent the round-off in both cases, i.e. α and $(1-\alpha)$. Multiplying by the denominators and incorporating these round-off variables then gives:

$$4I_{new}+2J_{new}+2IC-2J_{new}-2IC+2\alpha = 4I_{old}+2J_{old}+2IC-2J_{old}-2IC+2(1-\alpha)$$

which can be symbolically simplified to:

$$4I_{new} - 4I_{old} +4\alpha = 2$$

Since all variables are integers, the GCD test proves that this equation cannot have a solution and therefore no dependence is set.

3.3.2. Eliminating False Dependencies using Covering Set Calculation

One of the many reasons for carrying out a dependence analysis is to determine what loops (if any) can be executed in parallel. Generally, the more of the compute intensive loops that can exploit parallel execution, the better the parallel performance is likely to be. The only dependencies that fundamentally serialize a loop are data-flow dependencies where some values can pass from assignment to usage. One common case when such dependencies appear to exist is when workspace arrays are re-assigned within an application code. A simple example in Figure 4 shows that the **J** loop should be parallel. In order to prove this, however, it must be proven that the values of array **A** used in S_3 cannot emanate from previous iterations of the **J** loop. In this case, that requires that it is proved that all values used in S_3 are assigned during the same iteration of the **J** loop. This is achieved through the use of covering set analysis to compare the set of intermediate assignments of array **A** against the data used and assigned in a potential loop carried dependence from S_1 or S_2 to S_3 [54].

In Figure 4, the values of **A** are assigned in the routine **UPDATE** at statements S_5, S_6, S_7 and S_8. To apply the covering set test, the full set of intermediate assignments and the range of data they access along with their control information are gathered to form a set which potentially covers the dependence being tested. This set is then negated to represent the set of data not overwritten in intermediate assignments. In this case, the covering set consists of the union of the assignments in statements S_5, S_6, S_7 and S_8 incorporating the relevant control statements S_1, S_2 and S_4. This covering set is then substituted using the definite callers for each assignment with array **B** being

substituted with array **A**. This set is then tested against the assignment and usage of the loop carried dependence to prove that all data items in that set are overwritten and therefore the dependence can be eliminated. Consequently, the **J** loop can be executed in parallel.

```
   DO J=1,NJ
      IF (J.LE.JMIX) THEN
         . . .
S1       CALL UPDATE(A(1,J),NI,.TRUE.)
      ENDIF
      . . .
S2    IF ( J.GT.JMIX) THEN
         . . .
         CALL UPDATE(A(1,J),NI,.FALSE.)
      ENDIF
      DO I=1,NI
         ... = A(I,J)
      ENDDO
   ENDDO
```

```
   SUBROUTINE UPDATE(B,M,BLAY)
   REAL B(*)
   LOGICAL BLAY
S4    IF (BLAY) THEN
S5       B(1)=. . .
S6       B(M)=. . .
         DO I=2,M-1
S7          B(I)=. . .
         ENDDO
      ELSE
         DO I=1,M
S8          B(I)=. . .
         ENDDO
      ENDIF
```

Figure 4. Example of covering dependence sets.

3.3.3. Accurate Analysis of Linearized Array Accesses

A popular feature of scientific mesh-based codes is changing the dimension of an array when passing from one routine into another. It is used for a variety of reasons including memory management in caller routines, perceived optimization in computationally expensive routines etc. These features, however, mean that many dependence tests fail to correctly detect the existence or non-existence of data dependencies that are defined using these linearized expressions. Figure 5 shows a simple extract from a real application code where all referencing is linearized, and as such, commonly used dependence tests such as the Banerjee Inequality test [58] and the standard Omega test [59] fail to accurately detect the data dependencies present.

```
DO ITER=1,TOTALITER
      . . .
   DO J=2,NJ-1
      DO I=2,NI-1
         DO K=2,NK-1
            IJK=I+JOFFSET(J)+KOFFSET(K)
            IJKN=IJK+NI
            IJKS=IJK-NI
            IJKB=IJK-1
            IJKF=IJK+1
S2          X(K)=AE(IJK)*XO(K+1)+AW(IJK)*X(K-1)+AN(IJK)*F(IJKN)
      &     AS(IJK)*F(IJKS)+AB(IJK)*F(IJKB)+AF(IJK)*F(IJKF)
         ENDDO
      . . .
         DO K=2,NK-1
            IJK=I+JOFFSET(J)+KOFFSET(K)
S1          F(IJK)=A(K)*X(K)
            IF(F(IJK).LT.0.0)THEN
```

```
S3              F(IJK)=0.0
              ENDIF
            ENDDO
          ENDDO
        ENDDO
      ENDDO
```

Figure 5. Example of a one-dimensional array as used to represent a three-dimensional problem.

The integer arrays are set up earlier in the application code to indicate the start of lines and planes of the mesh, i.e.:

```
JOFFSET(J) = J*NI-NI        KOFFSET(K)=K*NI*NJ-NI*NJ
```

Considering the array reference **F(IJKS)** in S_2, forward substitution (also substituting for the integer arrays) extract **IJKS** to be the symbolic expression:

```
IJKS = I+J*NI-2*NI+K*NI*NJ-NI*NJ
```

indicating that the one-dimensional reference **F(IJKS)** is equivalent to a reference in three-dimensions of **F(I,J-1,K)** for a declaration of **F(NF,NI,NJ)**. Range-based tests, such as the Banerjee Inequality [52], fail in such cases since the loop ordering causes interlaced memory locations to be referenced (a loop ordering with **K** as the outer loop and **I** as the inner loop could be accurately handled by such a test). Another difficulty in achieving an accurate analysis involving these types of expressions is their symbolic non-linear nature.

To provide an accurate dependence analysis for non-linear, symbolic expressions, CAPTools uses the Symbolic Omega test (SOmega) [55], an extension to the standard Omega dependence test [59]. The two main phases of the SOmega test involve the Fourier-Motzkin elimination (FME) of variables from equation sets and the normalization of those equations to reduce the order of the contained terms. The SOmega algorithm is based on the use of symbolic factorization to accurately reduce non-linear terms, allowing continued use of the FME.

In the above example, the only correct true dependencies carried by the **J** loop are defined from the statement assignment in S_1 to the highlighted usage in statement S_2 and from the assignment in statement S_3 to the same usage in statement S_2, both of which are carried by the array **F**. Between iterations of the **J** loop each successive reference is a distance of **NJ** apart, therefore, only indices referenced in this way cause a dependence relation within the **J** loop.

3.3.4. User Interaction in Dependence Analysis

The result of a dependence analysis on industrial application codes will often require tuning and knowledge from the user to optimize the dependence graph. This is due to the analysis being unable to resolve dependencies because additional information was required about variables in the code. Knowledge from the user will typically be range information for integer variables whose

value is determined from run-time data. Using the *dependence graph browser*, these dependencies can often be eliminated if the user can answer the questions posed by the analysis during testing of the dependence. Figure 6 shows an example of the *dependence graph browser* displaying a graph from the ARC3D code and the *why dependence window* displaying information on a selected dependence. The dependence can be eliminated if the user can answer any of the questions provided by CAPTools. Furthermore, this knowledge can be used to resolve other uncertain dependencies via an incremental analysis.

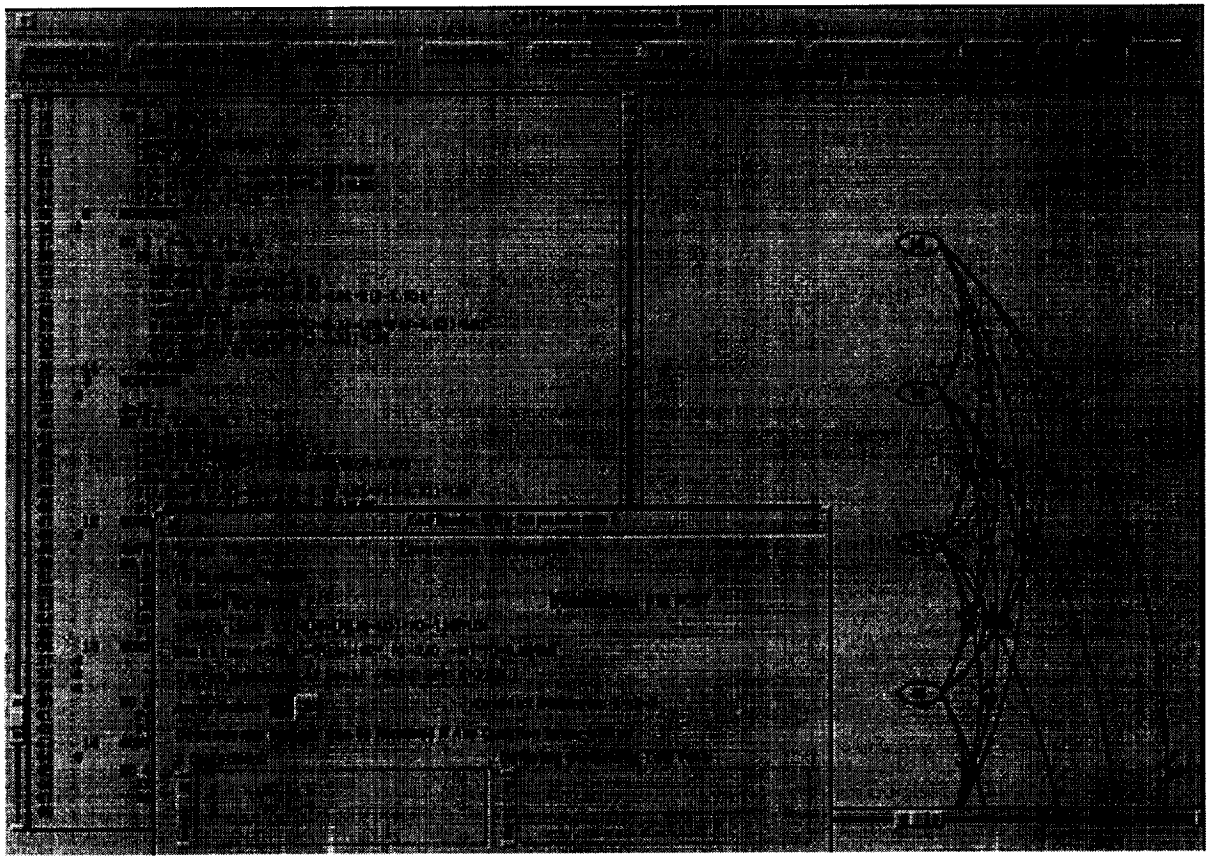

Figure 6. DependenceGraph Browser

3.4. Data Partitioning Techniques and Automatic Inheritance

For large application codes, the selection of the arrays to be partitioned across the processor topology can be a very significant task. It can involve a large number of arrays in every routine where the mapping of these arrays between routines is not always straightforward. Since the parallelizations are based upon domain decomposition, the partition determination should take a global view of the code. To make the parallelization tools accessible to the scientific and industrial communities, this process must be simplified and automated into as painless a task as possible.

3.4.1. Automatic Data Partition Inheritance

A basic idea of how the data should be distributed, given the applications and parallelization strategies considered here, is fairly simple to obtain. The automatic partition inheritance commences from a user selection of one array in one routine and one component of that array (for example, an index). From that selection, arrays related within single statements and transitively through dependencies to previously partitioned arrays can inherit the partition. Both positive and negative partition implications are inferred where an array should or should not be partitioned due to the relationships with other arrays. This process also proceeds interprocedurally so that a comprehensive array partition can be constructed throughout an application code [51].

3.4.2. Catering for Interprocedural Dimension Change

Interprocedural dimension change and the use of linearized array accesses as mentioned in section 3.3.3 also have implications when defining the data partitioning as part of the parallelization of a code. During the data partitioning process, arrays that are dimension-mapped across routines are accurately handled within the algorithm. This is achieved by internally specifying the data partitions in terms of symbolic **MODULUS** and **DIVISOR** expressions. For the code in Figure 7, the array **A** is first initialized as a one-dimensional array in routine **TIMESTEP** at S_1. Then, the array **A** is aliased to array **B** when used in routine **UPDATE** as a four-dimensional array.

```
      SUBROUTINE  TIMESTEP(A,NI,NJ,NK,NL)
      REAL A(NI*NJ*NK*NL)
      DO I=1,NI
         DO J=1,NJ
            DO K=1,NK
               DO L=1,NL
S1                A(I+(J-1)*NI+(K-1)*NI*NJ+
     &               (L-1)*NI*NJ*NK) = ...
      ENDDO
      CALL UPDATE(A,NI,NJ,NK,NL)
      DO I=1,NI*NJ*NK*NL
S2       A(I) = . . .
      ENDDO
      END
```

```
SUBROUTINE UPDATE(B,NI,NJ,NK,NL)
  REAL B(NI,NJ,NK,NL)
  DO L=1,NL
     DO K=1,NK
        DO J=1,NJ
           DO I=1,NI
         ... =B(I,J,K,L)
           ENDDO
        ENDDO
     ENDDO
  ENDDO
END
```

Figure 7. Simple Example of Interprocedural Dimension Mapping.

Using the initial partition described in Section 3.4.1, the user selects to partition array **B** in routine **UPDATE** in the third index (i.e. **K**). The equivalent definition of the partition for array **A** in routine **TIMESTEP** is represented in terms of symbolic **MODULUS** and **DIVISOR** expressions given by:

Modulus expression: NI*NJ*NK

Division expression: NI*NJ

The partitioned component for array **A** can then be extracted using symbolic factorization. Consider the reference to array **A** at S_1:

```
A(I+(J-1)*NI+(K-1)*NI*NJ+(L-1)*NI*NJ*NK)
```

Firstly, adjusting the expression to be relative to zero and applying the Modulus expression to the one-dimensional index expression removes terms that have a higher order than the partitioned component:

```
MOD(I+(J-1)*NI+(K-1)*NI*NJ+(L-1)*NI*NJ*NK-1 , NI*NJ*NK) = I+(J-1)*NI+(K-1)*NI*NJ-1
```

This is then followed by the application of the integer division expression to remove lower order terms:

```
DIV(I+(J-1)*NI+(K-1)*NI*NJ-1 , NI*NJ) = K-1
```

indicating a partition based upon the expression **K-1**.

In the case of the reference to array **A** in statement S_2, since the reference consists of a single variable, no partitioned component can be extracted. The code generation will then use loop unrolling to separate the partitioned component or, if that is not possible, use the **MODULUS** and **DIVISOR** expressions explicitly in the output code.

3.4.3. User Interaction in Data Partitioning

The goal of the data partitioning algorithm is to identify as many arrays as possible that are suitable for distribution using the SPMD paradigm. On completion of the data partitioning stage, the user can inspect the data partitioning defined for each array and if necessary, the user can enforce a coherent partitioning alignment for different arrays. It is also important for the user to inspect any arrays that were not partitioned. Although it may not be possible for the user to gauge the impact of arrays that are not partitioned on the quality of the generated parallel code at this stage, the user can inspect (through the *Partition Browser*) any reasons arrays were not partitioned, enabling a more informed decision to be made.

3.5. Parallel Execution Control Determination

The exploitation of parallelism from the sequential code is achieved during execution control mask addition. A maximum coverage of masks is essential if high efficiency is to be obtained from the resultant parallel code since any unmasked statements must be executed on every processor. The basic algorithm for the addition of execution control masks firstly uses the "owner computes" rule for assignments to partitioned arrays. This is followed by the addition of masks to statements related to previously masked statements through dependence relations and also to statements that use partitioned data [51]. This allows, for example, parallelism to be exploited in computations that involve arrays not yet partitioned. The determination of which masks to use on a statement when several options exist is based upon the loop nesting depth of the dependencies that inferred those masks, where the most deeply nested should be honored to avoid the re-

quirement of deeply nested communications. A key to the success of the execution control masking algorithm is the effective application in an interprocedural environment. In particular, it is very common for routine calls to be within a loop, where that loop should be executed in parallel. Figure 8 shows two cases within such a loop that employ different techniques.

```
   DO J = 1,NJ                          FUNCTION FACTOR(I,N,VAL)
      . . .                                FACTOR=(VAL*I)/N
S1    FAC = FACTOR(J,NJ,VAL)             END
      . . .
S2    CALL UPDATE(A(1,J),FAC,NI)         SUBROUTINE UPDATE(X,FAC,N)
      . . .                              . . .
   ENDDO                                    DO I=1,N
                                     S3        X(I)=X(I)+FAC
                                            ENDDO
```

Figure 8. Interprocedural execution control mask addition example.

Array **A** is partitioned in index 2, also forcing array **X** in routine **UPDATE** to be partitioned. The use of the modulus and divisor expressions to extract the partitioned component of the references to array **X** return the partitioned component as **1** (the default FORTRAN declaration limit). The execution control mask algorithm therefore places an owner computes mask based on the partitioned component **1** on statement S_3. The dependence relations then enforce that mask onto all the executable statements in routine **UPDATE**. Since the entire routine is masked, this mask can then be inherited by the call at S_2. It will then be adjusted to take account of the indices of **A** passed into routine **UPDATE**, creating a mask based on the expression **J**. The use of variable **FAC** in S_3 also infers a mask onto statement S_1, where that mask is again adjusted to use the expression **J**. The addition of a mask inherited via dependencies onto the statement S_1 is legal since the call to routine **FACTOR** contained in that statement accesses no partitioned arrays. Since both S_1 and S_2 have been masked, and assuming that all other statements in the loop also have masks based on **J**, it is possible to allow this mask to be inherited to the surrounding **J** loop, subsequently allowing it to operate in parallel.

In the case where routine **UPDATE** contained several statements with differing masks, a mask constructed from the union of the contained masks (i.e. combined using the **.OR.** operator) could be inherited by the call at S_2. The indices in array **A** in the call at S_1 requires code generation to alter the values of the partition range variables on entry to routine **UPDATE** i.e.

```
CALL UPDATE(A(1,J),FAC,NI,CAP_L-J+1,CAP_H-J+1)
SUBROUTINE UPDATE(X,FAC,N,CAP_L,CAP_H)
```

This allows the mask based on the expression **1** in routine **UPDATE** to be equivalent to a mask based on the expression **J** in the caller routine.

The CAPTools *Mask Browser* window allows the user to investigate the result of the automatic masking algorithm. The *Mask Browser* can show why a statement was masked and through a context sensitive link to the *WhyDependence* window and *Partition Browser* window, allowing the user to examine the dependencies and data partitions involved in the creation of that mask.

3.6. Communication Identification, Migration and Merger

The automatic generation of communication statements is essential for a parallelization tool aimed at distributed memory architectures. To achieve the high parallel efficiency, the number of communications generated in the parallel code must be minimal, avoiding duplicated or unnecessary communications. Additionally, placement of communications is crucial, in particular avoiding the placement within loops whenever possible. In doing so, this would reduce the frequency of communication calls, which in turn reduces the cost of the significant communication startup latency currently inherent with all parallel systems.

The essential features of the communication calculation algorithm in CAPTools include the ability to perform communication splitting, communication migration and communication merger.

3.6.1. Communication Splitting

Much effort is invested in attempting to move communication requests outside of the loop nest they originated from. As an illustration we consider a code extract that describes the partial use of the upwind differencing scheme in a computational mechanics code (Figure 9).

```
   DO J=2,NJ-1
      DO I=2,NI-1
         ...
S3       NSUP = IFIX(SIGN(1.0,CONVECW)
         ...
         KLOC=J-NSUP
S2       JLOC=KLOC-(1+NSUP)/2
         ...
S1       USOURCE(I-1,J)=USOURCE(I-1,J)-COEF*UVEL(I,JLOC)
        ENDDO
   ENDDO
```

Figure 9. Example of Upwind Differencing Taken from a Computational Mechanics Code.

A communication request is initially made on statement S_1 because the array element **UVEL(I,JLOC)** may be owned by another processor. The migration of the communication request for **UVEL** is bariered by the assignment of the index **JLOC**. CAPTools uses logical substitution to determine the assignment(s) of variables, therefore, the assignment of **JLOC** (S_2) is dependent on the variable **NSUP** (S_3). In general, the assignment of **NSUP** is a barrier to the

communication request from statement S_1 leaving the communication of single values of the **UVEL** array within the **I** and **J** loops. In this case however, CAPTools can further resolve the intrinsic **SIGN** function and therefore the knowledge is added that **NSUP** is assigned to be -1 or +1 (and hence **JLOC** can only ever be J-2 or J+1). The single request is then replaced by two requests for the array values of **UVEL(I,J-2)** and **UVEL(I,J+1)** and the assignment S_3 is no longer a barrier to the commuication . Consequently, these requests are migrated outside both the **I** and **J** loops allowing further migration through the code.

3.6.2. Interprocedural Migration of Communications

As well as moving communication requests outside loop nests it is essential to be able to migrate the requests out from one routine and into another. This can also greatly reduce the number of communications generated. The communication requests are migrated through the application code traversing the pre-dominator tree of the control flow graph [58], until they are prevented from moving up any further by a barrier such as an assignment to the data that is to be communicated.

Additionally, as real applications usually include a very large number of routines, interprocedural migration of these communication requests is essential. An obvious reason for this is to avoid of repeated communication in different routines when a single communication prior to both routines would be more efficient. The migration of such requests to the same point in the application provides the possibility of them being merged into a single communication. Consider the communication in the CAPTools communications browser in Figure 10.

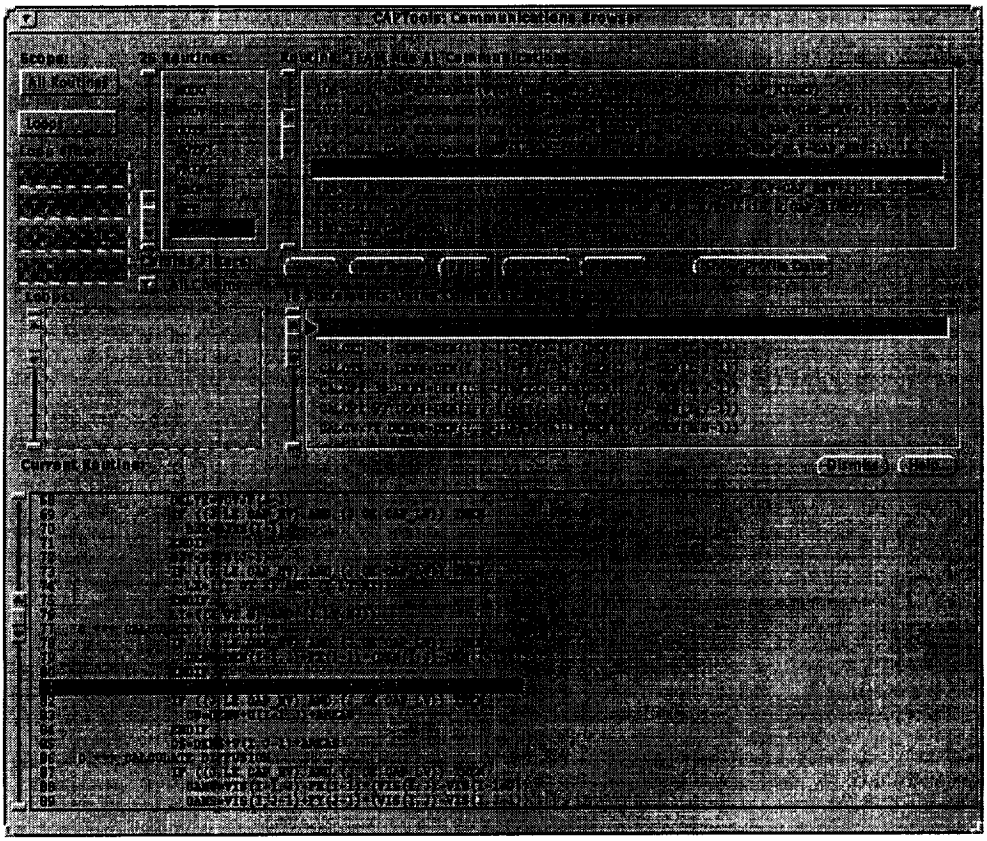

Figure 10. Communications browser highlighting interprocedural communication migration.

The browser shows a highlighted exchange communication of array **DEN** (density) in the main program of the application. For this communication, there are 16 statements that requested (and would therefore use) the received data values (as shown in the lower list). These statements appear in seven different routines, clearly illustrating the importance of the interprocedural aspect of the migration and merger algorithms. Without such migration, at least six extra communications would be required, at least one in each of the original routines.

For structured mesh codes a single communication request is issued and migrated through the code, eventually barriered, for example, by the assignment of the data to be communicated. The generation of communications for an unstructured mesh code uses the inspector-executor approach [60] so a separate request for an inspector loop and a communication (executor) are issued. The inspector request is needed for the construction of the list of all items that are to be communicated. It is constructed from code sections from any number of routines as required to evaluate the set of array indices required in the requesting computation. The code to be included in the inspector loop is determined to allow migration of inspector loops out of loop nests, where dependencies of any statement to be included in the inspector loop represent a barrier to migration. The executor request, in a similar way to a structured mesh request, is for the

communication statement, indicating, for example, the array to be communicated. Since only the executor request is barriered by the assignment of the data to be communicated, the inspector loop will migrate past the communication.

3.6.3. Communication Merging

Communication requests that have migrated to the same point in the code are then merged where possible by comparing the data space they cover. This may involve either the deletion of a communication request that is proven to be a subset of another request, or the merger of two requests whose data spaces intersect. The unnecessary replication of communications can adversely affect the quality of the generated code so it is essential to merge any communications whenever possible and in doing so, generate code that does follow the best manual practice. The example in Figure 10 shows that through the use of the interprocedural capability of CAPTools the communication requests were migrated out from 7 different routines and were all barriered by the same statement. Without the merger of these communications into a single communications this would lead to an unnecessary number of communications being generated.

Merger of unstructured mesh communications uses the same technique as employed in structured mesh parallelizations, except that the merger of the associated inspector loops is a pre-requisite for the merger of communications. The inspector loop merger algorithm uses an induction proof technique [61] to allow merger without consideration of textual and statement ordering differences. The primary goal of this merger is the subsequent merger of communications, however, a secondary pass of the algorithm is also made to eliminate duplicated inspector loops in order to reduce runtime overheads in terms of execution time, memory requirements and generated code volume.

3.7. Essential and Desirable User Interaction

On completion of the communication statements generated, the user can inspect these communications using the communications browser. For moderately complex codes, the user can examine the number of communications generated on a routine by routine basis or even for the entire code to get an initial impression of the quality of the parallelization of the code. For complex industrial application codes however, this may not provide any intuitive feel. Due to the complexity of the code, there may be a large number of communications generated, but these may only execute under certain conditions. For example, the setting up of complex boundary conditions may lead to complex execution control masks which have in turn been inherited by the communication (of data required to correctly compute the boundary data). Another possible scenario could begin with the user inquiring why a communication was required. On examination of the execution control mask associated with the communication (through the *Mask Browser*) it is discovered that the data being used is not partitioned (through the *Partition Browser*). The user can inquire the reasons why the data was not partitioned and can take one of a number of actions as a result of

the information presented. For example, the user can decide to partition the array, or it may be necessary to repartition from scratch selecting a different starting point, or the user may be satisfied with the partitioning and hence the side-effect of additional data communication.

The CAPTools browsers allow the user to explore and extract information about why certain actions were taken and this enables the user to make more informed decisions about the parallelization.

"Value profiling" [46] can be used to calculate the number of times each communication call is executed. Using this information, the user can focus on those communications which are called most frequently, allowing for manual optimizations or even revisiting an earlier stage in the parallelization process. CAPTools can also generate instrumented versions of the parallel code for use with the NASA's Automated Instrumentation and Monitoring System, AIMS [62]. CAPTools currently has a rudimentary ability to interact with AIMS' Trace Visualizer. This enables the user to further investigate the quality of the parallel code through the performance traces generated. Selecting a communication in the trace file allows the user to explore (through the CAPTools *Communication Browser*) the reasons why the communication was needed and may provide the user with some insight into how the communication may be optimized, if at all possible. This might entail merging key communications, or the resolution of related uncertain dependencies.

4. Examples and Evaluation

4.1. NAS Parallel Benchmarks

The performance of the code CAPTools produced was first compared against Portland Group's HPF compiler (pgHPF), and the usage of compiler directives available with SGI's FORTAN77 compiler (SGI-pfa) using the NAS Parallel Benchmarks (NPB's) [63-65]. NPB consists of five kernels and three simulated CFD applications derived from important classes of aerophysics applications. These five kernels mimic the computational core of five numerical methods used by CFD applications. The simulated CFD applications reproduce much of the data movement and computation found in full CFD codes. They were designed to compare the performance of parallel computers and are widely recognized as a standard indicator of computer performance. The latest release, NPB2.3, contains an MPI-based message-passing version as well as a serial version . For this case study, we used four of the NPB's: LU, SP, BT, and FT.

- ***LU*** uses symmetric successive over-relaxation (SSOR) to solve a block lower triangular-block upper triangular system of equations resulting from an unfactored implicit finite-difference discretization of the Navier-Stokes equations in 3D.
- ***SP*** uses an implicit algorithm to solve the 3D compressible Navier-Stokes equations. The finite differences solution to the problem is based on a Beam-Warming approximate fac-

torization that decouples the x, y and z dimensions. The resulting system has scalar pentadiagonal bands of linear equations that are solved using Gaussian elimination without pivoting. Within the algorithm, this solution is performed by the ADI (alternating direction implicit) solver which solves three systems of equations sequentially along each dimension.

- ***BT*** solves systems of equations resulting from approximately factored implicit finite-difference discretization of the Navier-Stokes equations in three dimensions. Block-tridiagonal systems of 5×5 are solved in a similar fashion as SP.
- ***FT*** contains the computational kernel of a three-dimensional (3-D) Fast Fourier Transform (FFT)-based spectral method. FT performs three one-dimensional (1-D) FFT's, one for each dimension.

Two computing platforms were used in our experiments: an SGI Origin 2000 and an IBM SP2. Figure 11 compares the performance of the three automated approaches (CAPTools, SGI-pfa and PGHPF) to the performance of the hand coded benchmark (NPB-2.3) on the SGI Origin 2000. A similar comparison is shown on Figure 12 for the IBM SP2.

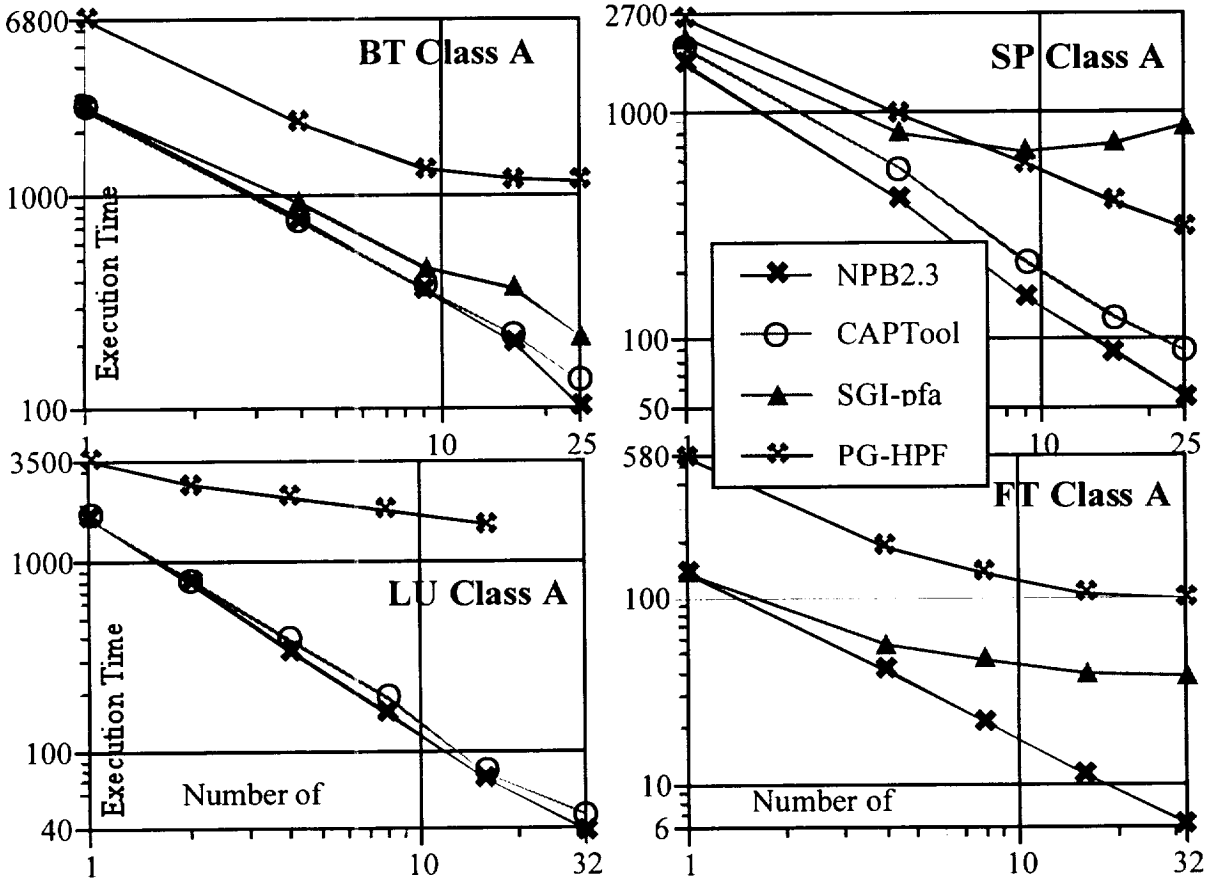

Figure 11. Performance Comparison for 4 Benchmarks

The following observations can be noted:

- CAPTools generated code have similar performance for the execution on one node serial code indicating that the serial code is close to NPB 2.3. In most cases, they also scale similarly to the hand-written NPB 2.3 code. CAPTools consistently produces code with the best performance of the three automated approaches. However, the efficient parallelization of FT requires a data transpose, a feature not currently available on CAPTools.
- The HPF code has the worst performance for almost all cases. Its performance on a single processor is still over twice as slow as the other approaches, signifying that part of the reason for its relatively poor performance is the compiler. Furthermore, because it must use data transposition for LU, BT and SP instead of parallel pipelined computation and communication, the performance is worse than the other versions. (For more details see [17, 66])
- The Origin compiled code has good performance in some cases. The use of automated tools[9] combined with tuning improves the performance of generated code. User intervention includes: forcing parallelization of certain loops, loop-transformations and co-locating certain computation with its data. (For more details see [68])

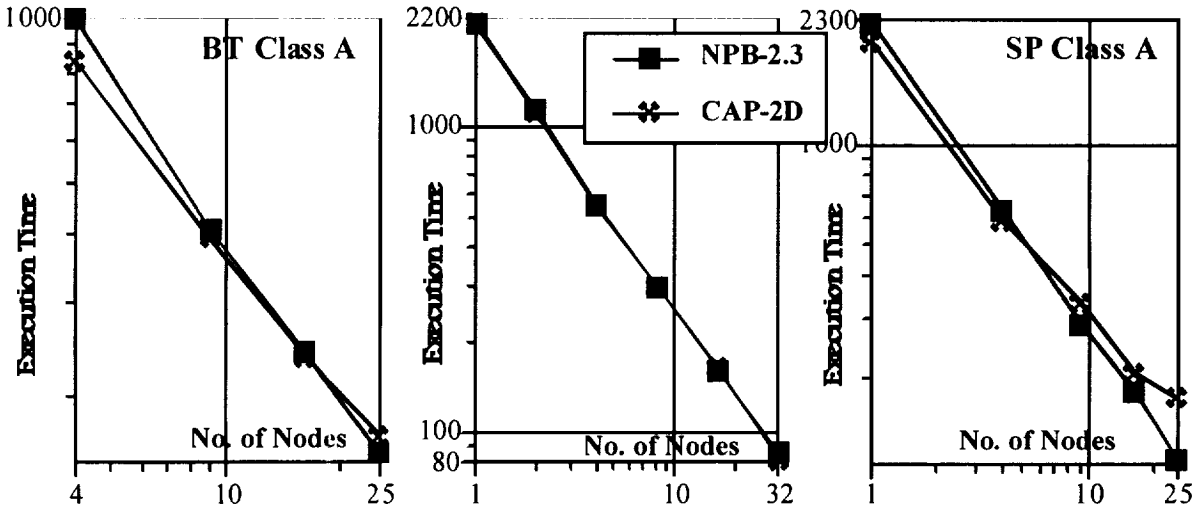

Figure 12. Execution time of LU, BT, SP Class A on IBM SP-2

It should further be noted that while NPB 2.3 took man-months to be parallelized, CAPTools took hours to generate the first parallel versions and days to tune. The directive-based versions of NPB took days to be produced and HPF versions took much longer because the benchmarks have to be re-written [17].

---

[9] Power Fortran Accelerator (PFA) [67] can automatically insert parallelization directives in sequential code and transform the loops to enhance their performance. Parallel Analyzer View (PAV) can annotate the results of dependence analysis of PFA and present them graphically. Using these tools in conjunction with the MipsPro Fortran77 compiler allow incremental tunining of the parallelized code.

4.2. ARC3D

ARC3D [69] is a CFD application developed at NASA Ames Research Center. Parallelization of ARC3D is significant as it forms the basis of an application being used in the design of aircraft today. ARC3D solves Euler and Navier-Stokes equations in three dimensions using a single rectilinear grid. Unlike the NAS benchmarks, ARC3D contains a turbulent model and a more realistic boundary condition. The code was automatically parallelized using CAPTools in approximately 1 hour. In this case, the CAPTools routine duplication transformation was required to allow efficient code to be generated in two routines. An essential feature of the generated code is the use of software pipelines to handle the implicit nature of the solvers. This can carry a high cost if the communication startup latency is high on the parallel platform and/or there is a relatively low computational load within each stage of the pipeline. Table 2 shows the performance of the generated parallel code using a 2-D block partition on the Cray T3D/E and IBM SP2.

IBM SP2		Cray T3D		Cray T3E	
Processors	Speed Up	Processors	Speed Up	Processors	Speed Up
2(1×2)	1.99	4(2×2)	3.68	4(2×2)	3.78
4(2×2)	3.79	16(4×4)	11.79	16(4×4)	12.31
16(4×4)	8.15	32(4×8)	19.87	32(4×8)	21.09
		64(8×8)	31.22	64(8×8)	33.52

Table 2. Results for ARC3D on a 64×64×64 grid on the Cray T3D/E and IBM SP2 using 2-D partitioning.

	SGI Origin 2000	Cray T3E
Processors	Speedup	Speedup
4	3.91	4.005
16	13.58	13.18
25	19.40	21.66
32	21.84	25.11
36	23.43	29.23

Table 3. CAPTools generated parallel code on SGI Origin and Cray T3E for ARC3D.

Due to the recognition of the CAPTools generated code by the user, it was possible to carry out minor optimizations to the algorithm in the code, whereby the number of pipeline startup and shutdown stages were reduced. These results are shown in Table 3 for the SGI Origin and the Cray T3E.

4.3. Finite Element Stress/Strain Analysis Code

A code that uses finite element techniques to solve for displacement, stress and strain over a number of time steps, in either a 2- or 3-D mesh was tested with CAPTools. As with many such codes, the algorithm constructs stiffness matrices for each finite element in turn and incorporates the components of these matrices into a full system matrix and boundary condition vector. The resulting linear system is then solved by a Conjugate Gradient solution procedure to determine displacements. These displacements are then used to evaluate stress and strain values throughout the problem domain.

The code was parallelized in one day using CAPTools and the results are presented for a small 3000 element simulation taken from the NAFEMS benchmark set [70] (Figure 13). A loop-split transformation was carried out using CAPTools for a loop in which the system matrix set up was defined. This enabled a more efficient placement of a communication in the generated code. The CAPTools generated code was re-compiled for the different platforms without any machine specific optimizations being made. These results show that the generated code provides high efficiency even on this small mesh. The decrease in efficiency as the number of processor increase is due to the small amount of computation that is independent of the communication that cannot completely overlap the communication time.

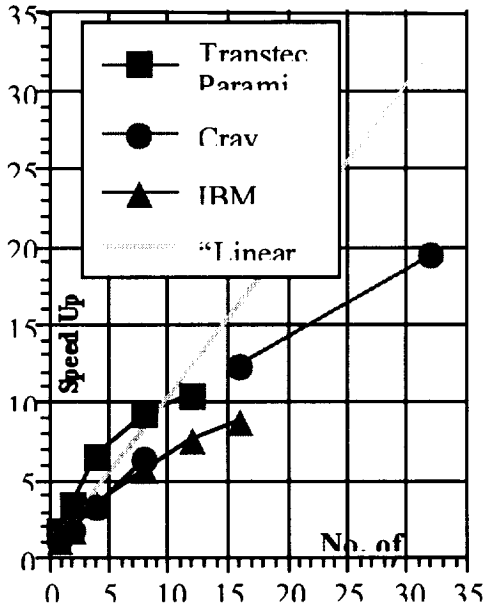

Figure 13. Results for stress code on different parallel platforms

4.4. UIFS – Unstructured Incompressible Flow and Stress

UIFS uses an unstructured mesh control volume discretization of the domain to solve for the incompressible Navier-Stokes equations. The code employs the SIMPLE solution procedure and solves coupled equations for momentum, continuity, heat, solidification, displacement, stress and

strain. The flow components are solved using classical iterative schemes and the stress computations are solved using the conjugate gradient algorithm with diagonal preconditioning. Examination of the code generated by CAPTools reveals that inspector loops were placed appropriately for different calculation modules. For example, inspector loops were generated specifically for the flow computations and another set generated specifically for the stress computations. This distinction was correct since the overlap data requirements for the two computations were not the same. In general, the computation required in the flow module are based on "fluid" cells in the overlap areas, and for computation in the solid mechanics modules only "solid" cells are required in the overlap areas. The time taken to parallelize this code manually was about six months of effort, whereas the time taken to parallelize the same code using CAPTools was a few days. The results for the parallel execution of CAPTools generated code for a thin plate problem using 30,000 nodes are shown in Figure 14. Although, the speed up figures are not as high as that obtained for a fully optimized manual parallelization [50] they do follow similar trends. The CAPTools generated code has a similar appearance to the manually generated parallel code because the strategies used by CAPTools were extracted from the best manual parallelization. As such, the generated code can be optimized to achieve the same performance.

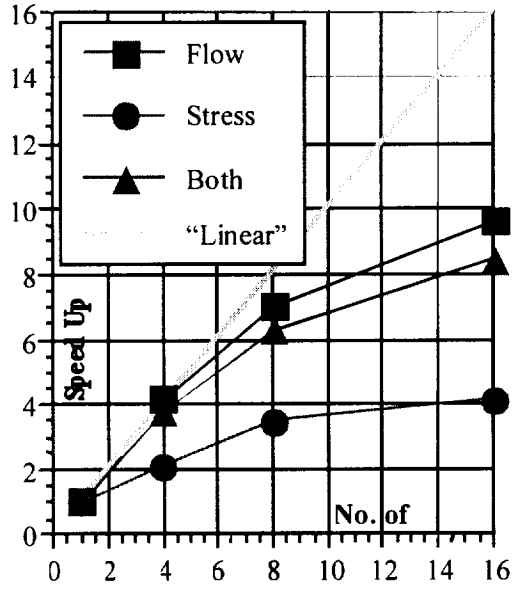

Figure 14. Performance of CAPTools Generated Code for a Fluid-solid Interaction Application

5. Conclusions and Future Research

Parallel programming is a difficult process; there are many challenges to generating efficient parallel code. Currently, there are four major approaches besides hand coding for porting applications to parallel architectures. These include the use of parallelizing compilers, user supplied "directives", and semi-custom library. All these approaches are unattractive because they require

the user to analyze the entire code and re-write part of it. The use of a parallelization toolkit has two major advantages:

1. The user operates at a higher level; leaving the mundane, and error-prone tasks to the tool; and
2. A tool helps the user to quickly focus on a very small portion of the code critical to parallel performance.

In light of these considerations, CAPTools was constructed specifically to cater for the parallelization of large-scale scientific applications, many of which are "legacy codes" that have been developed and used for many years. CAPTools was designed to ensure its

- **applicability**: accepts a wide range of FORTRAN standards; recognizes common programming practices;
- **performance**: allows the user to control the amount of resources devoted to program analysis to ensure the quality of the result; makes use of user knowledge to reduce analysis time; employs parallelization algorithms mimicking best manual practice, interacts with the user at all stages of the parallelization to ensure the quality of the resultant code;
- **maintainability**: generates recognizable code following well-understood parallelization techniques; enables further development of parallel code if needed; supports efficient incremental analysis should user modify a small section of the application; and
- **portability**: generates parallel code based on widely supported standards such as MPI and PVM. An "OpenMP" version is being developed at NASA.

The performance of the generated code has been evaluated using the NAS Parallel Benchmarks as well as larger applications. Results to-date indicate that CAPTools generated code performs very closely to message-passing code written by hand. Furthermore, this performance is maintained across at least three architectures (Cray T3E, IBM SP2 and SGI Origin 2000). Furthermore, parallelization can be obtained in hours or days, as opposed to weeks or months for manual parallelization.

6. References

[1] "The National Partnership for Advanced Computational Infrastructure", NPACI, *http://www.npaci.edu/Research*

[2] "Partnerships for Advanced Computational Infrastructure", PACI, *http://www.cise.nsf.gov/acir/paci.html*

[3] "CF90 Commands and Directives Reference Manual," Cray Research, Inc. SR-3901.10, 1993.

[4] "Forge Explorer User's Guide", Advanced Parallel Research, Inc., *http://www.qpsf.edu.au/workshop/forge/forge.html*

[5] "KAP/Pro Toolset for OpenMP", KAI, Inc., *http://www.kai.com:80*

[6] "MIPSpro(TM) Compilers", SGI, Inc., *http://www.sgi.com/origin/products/compilers.html*

[7] "OpenMP: A Proposed Standard API for Shared Memory Programming", *http://www.openmp.org/*

[8] C. H. Koelbel, D. B. Loverman, R. Shreiber, J. G.L. Steele, and M. E. Zosel, *The High Performance FORTRAN Handbook*: MIT Press, 1994.

[9] D. Gannon, P. Beckman, E. Johnson, T. Green, and M. Levine, "HPC++ and HPC++ Lib Toolkit," Indiana University, Bloomington, IN 1997.

[10] S. Benkner, "Vienna FORTRAN 90- An Advanced Data Parallel Language," presented at International Conference on Parallel Computing Technologies (PACT-95), St. Petersburg, Russia, 1995.

[11] V. Kumar, A. Grama, A. Gupta, and G. Karypis, "Introduction to parallel computing," : Benjamin/Cummings, 1994, pp. 450-459.

[12] C. Clémençon, K. M. Decker, V. R. Deshpande, A. Endo, J. Fritscher, P. A. R. Lorenzo, N. Masuda, A. M-uller, R. R-uhl, W. Sawyer, B. J. N. Wylie, and F. Zimmerman, "Tools-Supported HPF and MPI Parallelization of the NAS Parallel Benchmarks," Swiss Center for Scientific Computing, Manno, Switzerland TR-96-02, March, 1996 1996.

[13] "CM Fortran Reference Manual," Thinking Machines Corporation,, Cambridge, MA Version 5.2, 1989.

[14] D. Gannon and e. al., "Implementing a parallel C++ runtime system for scalable parallel systems," presented at Supercomputing '93, Portland, OR, 1993.

[15] G. C. Fox, S. Hiranandani, K. Kennedy, C. Koelbel, U. Kremer, C. W. Tseng, and M. Wu, "The FORTAN D Language Specification," CRPC, Rice University CRPC-TR90079, December, 1990 1990.

[16] A. S. Grimshaw, "Easy to use object-oriented parallel programming with Mentat," in *IEEE Computer*, 1993, pp. 39-51.

[17] M. Frumkin, H. Jin, and J. Yan, "HPF Implementation of NPB2.3," presented at ISCA 11th International Conference on Parallel and Distributed Computing Systems, Chicago, IL, 1998.

[18] S. Hiranandani, K. Kennedy, and C. Tseng, "Preliminary Experiences with the Fortran D Compiler," presented at Supercomputing '93, Portland, OR, 1993.

[19] E. Ayguade, J. Garcia, and U. Kremer, "Tools and Techniques for Automatic Data Layout: A Case Study," *Parallel Computing*, 1998.

[20] U. Kremer, "Automatic Data Layout With Read-Only Replication and Memory Constraints," in *Language and Compilers for Parallel Computing, Lecture Notes in Computer Science*. Berlin: Springer-Verlag, 1998.

[21] T. Gross, D. O'Hallaron, and J. Subhlok, "Task Parallelism in a High Performance Fortran Framework," in *IEEE Parallel & Distributed Technology*, vol. 2, 1994, pp. 16-26.

[22] I. T. Foster and K. M. Chandy, "Fortran M: A Language for Modular Programming," Argonne National Laboratory MCS-P327-0992, June, 1992 1992.

[23] B. Chapman, P. Mehrotra, J. V. Rosendale, and H. Zima, "A software architecture for multi-disciplinary applications: Integrating task and data parallelism," ICASE, NASA Langley Research Center,, Hampton, VA, 94-18, March 1994 1994.

[24] D. C. Cann, "The Optimizing SISAL Compiler: Version 12.0," Lawrence Livermore National Laboratory,, Livermore, CA 94550 UCRLMA-110080, April, 1992 1992.

[25] Culler and e. al., "Parallel Programming in Split-C," presented at Supercomputing '93, Portland, OR, 1993.

[26] L. S. Blackford and e. al., "ScaLAPACK: a Portable Linear Algebra Library for Distributed Memory Computers - Design Issues and Performance," presented at Supercomputing '96, Pittsburg, PA, 1996.

[27] S. Balay, W. D. Gropp, L. C. McInnes, and B. F. Smith, "Efficient Management of Parallelism in Object-Oriented Numerical Software Libraries," in *Modern Software Tools in Scientific Computing*, E. Arge, A. M. Bruaset, and H. P. Langtangen, Eds.: Birkhauser Press, 1997.

[28] R. F. V. d. Wijngaart, "Charon toolkit for parallel, implicit structured-grid computations: Literature survey and conceptual design," NASA Ames Research Center, NAS Division, Moffett Field NAS-97-018, 1997.

[29] S. Hiranandani, K. Kennedy, C. Tseng, and S. Warren, "Design and implementation of the D editor," presented at Second SIAM workshop on environments and tools for parallel scientific computing, Townsend, Tenessee, 1994.

[30] K. Kennedy, K. S. McKinley, and C. Tseng, "Interactive parallel programming using the ParaScope editor," *IEEE Trans on parallel and distributed systems*, vol. 2, pp. 329-341, 1991.

[31] V. S. Adve, J. Mellor-Crummey, M. Anderson, K. Kennedy, J.-C. Wang, and D. Reed, "An Integrated Compilation and Performance Analysis Environment for Data Parallel Programs," presented at Supercomputing '95, San Diego, CA, 1995.

[32] S. Benkner, S. Andel, R. Blasko, P. Brezany, A. Celic, B. M. Chapman, M. Egg, T. Fahringer, J. Hulman, Y. Hou, E. Kelc, E. Mehofer, H. Moritsch, M. Paul, K. Sanjari, V. Sipkova, B. Velkov, B. Wender, and H. P. Zima, "Vienna FORTRAN Compilation System- Version 1.2- User's Guide," University of Vienna, Vienna, Austria Oct. 1995 1995.

[33] S. Benkner, "HPF+: High Performance Fortran for Advanced Industrial Applications," presented at HPCN '98, Amsterdam, Nederlands, 1998.

[34] "SUIF Explorer: A Programming Assistant for Parallel Machines", *http://www-suif.stanford.edu/suifconf/suifconf2/*

[35] R. P. Wilson, R. S. French, C. S. Wilson, S. P. Amarasinghe, J. M. Anderson, S. W. K. Tjiang, S.-W. Liao, C.-W. Tseng, M. W. Hall, M. Lam, and J. Hennessy, "SUIF: An Infrastructure for Research on Parallelizing and Optimizing Compilers," Computer Systems Laboratory, Stanford University, Stanford, CA.

[36] B. Blume, R. Eigenmann, K. Faigin, J. Grout, J. Lee, T. Lawrence, J. Hoeflinger, D. Padua, Y. Paek, P. Petersen, L. Rauchwerger, P. Tu, and S. Weatherford, "Restructuring Programs for High-Speed Computers with Polaris," presented at ICPP Workshop on Challenges for Parallel Processing, 1996.

[37] V. Adve, G. Jin, J. Mellor-Crummey, and Q. Yi, "High Performance Fortran Compilation Techniques for Parallelizing Scientific Codes," presented at Supercomputing '98, Orlando, FL, 1998.

[38] G. Robinson, "Parallel Computational Fluid Dynamics On Unstructured Meshes Using Algebraic Multigrid," presented at Parallel CFD '92, 1992.

[39] C. Bergman, P. Wahlund, and J. B. Vos, "Implementation of a 2D Multi Block Euler Solver on the Connection Machine," presented at Parallel CFD '91, 1991.

[40] Ortner, G. Sieder, and D. Hanel, "Solution of the Navier-Stokes Equations on a Massively Parallel Transputer System," presented at Parallel CFD '91, 1991.

[41] S. P. Johnson and M. Cross, "Mapping Structured Grid Three-Dimensional CFD Codes Onto Parallel Architectures," *Applied Mathematics Modelling*, vol. 15, 1991.

[42] M. Cross, S. P. Johnson, and P. Chow, "Mapping Enthalpy Based Solidification Algorithms onto Vector and Parallel Architectures," *Applied Mathematics Modelling*, vol. 13, 1989.

[43] S. P. Johnson, F. Ali, and M. Cross, "Parallelising of the FAMCALC FEA Code," University of Greenwich 1992 1992.

[44] C. S. Ierotheou, C. Forsey, and U. Block, "Parallelisation of Novel 3D Hybrid Structured-Unstructured Grid CFD Production Code," presented at HPCN '95, 1995.

[45] P. Leggett, "CAPTools Communications Library," University of Greenwich, London, London 98/IM/37, 1998.

[46] S. P. Johnson, P. F. Leggett, C. S. Ierotheou, A. J. Marsh, E. W. Evans, and M. Cross, "Computer Aided Parallelisation Tools (CAPTools) User Manual," University of Greenwich, London 1996.

[47] C. H. Walshaw, M. Cross, and M. G. Everett, "A Localised Algorithm for Optimising Unstructured Meshes," *Int. J. of Supercomputing Applications*, vol. 9, pp. 280-295, 1995.

[48] G. Karypis and V. Kumar, "A Fast and High Quality Multilevel Scheme for Partitioning Irregular Graphs," Computer Science Department, University of Minnesota, Minneapolis, MN 55455 TR95-035, 1995 1995.

[49] R. Das and J. Saltz, "A Manual for PARTI runtime primitives - Revision 2," University of Maryland 1992 1992.

[50] K. McManus, "A Strategy For Mapping Unstructured Mesh Computational Mechanics Programs Onto Distributed Memory Parallel Architectures," in *Department of Mathematics and Computing*. London: University Of Greenwich, 1995

[51] S. P. Johnson, C. S. Ierotheou, and M. Cross, "Automatic Parallel Code Generation For Message Passing on Distributed Memory Systems," *Parallel Computing*, vol. 22, pp. 227-258, 1996.

[52] U. Banerjee, *Dependence Analysis For Supercomputing*: Kluwer Academic Publishers, 1988.

[53] J. R. Allen and K. Kennedy, "Automatic Translation of Fortran Programs to Vector Form," *ACM Trans. Programming Languages Systems*, vol. 9, pp. 491-542, 1987.

[54] S. P. Johnson, M. Cross, and M. Everett, "Exploitation of Symbolic Information in Interprocedural Dependence Analysis," *Parallel Computing*, vol. 22, pp. 197-226, 1996.

[55] C. S. Ierotheou, S. P. Johnson, and M. Cross, "An Extension of the Standard Omega Test for the Parallelisation of Computational Mechanics Codes Containing Arrays with Mapped Indices," 98/IM/34, 1998.

[56] P. Havlak and K. Kennedy, "An Implementation Of Interprocedural Bounded Regular Section Analysis," *IEEE Transactions On Parallel And Distributed Systems*, vol. 2, 1991.

[57] Z. Li and P.-C. Yew, "Program Parallelisation with Interprocedural Analysis," *Journal Of Supercomputing*, vol. 2, pp. 225-244, 1988.

[58] S. P. Johnson, "Mapping Numerical Software Onto Distributed Memory Parallel Systems," in *Department of Computer Science and Mathematics*. London: University of Greenwich, 1992.

[59] W. Pugh, "The Omega test: a fast and practical integer programming algorithm for dependence analysis"," in *Communications of the ACM*, 1992, pp. 102-114.

[60] J. H. Saltz, R. Mirchandaney, and K. Crowley, "Run-time Parallelization and Scheduling of Loops," *IEEE Tansactios on Computers*, vol. 40, pp. 5, 1991.

[61] S. P. Johnson, K. McManus, C. S. ierotheou, P. F. Leggett, and M. Cross, "Inspector Loop Determination To Reduce Communication Overheads in Unstructured Mesh Code Parallelisation.," University of Greenwich, London PPRG-98-003, Jauary 1998 1998.

[62] J. C. Yan, S. R. Sarukkai, and P. Mehra, "Performance Measurement, Visualization and Modeling of Parallel and Distributed Programs using the AIMS Toolkit," *Software Practice and Experience*, vol. 25, pp. 429-461, 1995.

[63] D. Bailey, J. Barton, T. Lasinksi, and H. Simon, "The NAS Parallel Benchmarks," NASA Ames Research Center, Moffett Field, CA RNR-91-002, January 1991.

[64] D. Bailey, T. Harris, W. Saphir, R. V. d. Wijngaart, A. Woo, and M. Yarrow, "The NAS Parallel Benchmarks 2.0," NASA Ames Research Center, Moffett Field, CA RNR-95-020, 1995.

[65] "NAS-Parallel Benchmarks 2.3", NAS Division, NASA Ames Research Center, *http://science.nas.nasa.gov/Software/NPB*

[66] M. Frumkin, "HPF Implementation of LU NAS Parallel Benchmark," NASA Ames Research Center, Moffett Field, CA In preparation, 1998 1998.

[67] "MIPSpro Fortran77 Programmer's Guide", Silicon Graphics, Inc., *http://techpubs.sgi.com/library/dynaweb_bin/0640/bin/nph-dynaweb.cgi/dynaweb/SGI_Developer/MproF77_PG/@Generic__BookView*

[68] A. Waheed and J. Yan, "Parallelization of NAS Benchmarks for Shared Memory Multiprocessors," presented at HPCN '98, Amsterdam, 1998.

[69] T. H. Pulliam, "Solution Methods In Computational Fluid Dynamics," , , Belgium, 1986," in *Notes for the von Kármán Institute For Fluid Dynamics Lecture Series*. Rhode-St-Genese, Belgium, 1986.

[70] G. A. O. Davies, R. T. Fenner, and R. W. Lewis, "NAFEMS - "Background to benchmarks"," : Nafems, 1993.