An Automatic and Symbolic Parallelization System for Distributed Memory Parallel Computers

K. Ikudome, G. C. Fox

Caltech Concurrent Computation Program
California Institute of Technology, Pasadena, CA 91125

A. Kolawa, J. W. Flower

Parasoft Corporation
2500 E. Foothill Blvd., Suite 205, Pasadena, CA 91107

Abstract

This paper describes *ASPAR* (Automatic and Symbolic PARallelization) which consists of a source-to-source parallelizer and a set of interactive graphic tools. While the issues of data dependency have already been explored and used in many parallel computer systems such as vector and shared memory machines, distributed memory parallel computers require, in addition, explicit data decomposition. New symbolic analysis and data-dependency analysis methods are used to determine an explicit data decomposition scheme. Automatic parallelization models using high level communications are also described in this paper. The target applications are of the "regular-mesh" type typical of many scientific calculations.

The system has been implemented for the language C, and is designed for easy modification for other languages such as Fortran.

1. Introduction

Distributed memory parallel computers, while offering virtually unlimited, cost effective performance [12], suffer by comparison with other architectures in their perceived programming problems. Parallelization by individual users has shown that the architecture is extremely powerful and has led to the development of sophisticated runtime systems such as *Express* which support the communication, decomposition, I/O, et.c requirements of such programs. Despite these advances application developers continue to develop conventional sequential programs in which the natural or inherent parallelism is all too often obscured by programming "tricks". Since these sequential algorithms are often required to execute on parallel computers for performance reasons we must develop methods by which they can be easily converted.

There are several potential approaches to making programming for a parallel computer easy:

1) New or extended languages: OCCAM, Ada, Strand, Fortran/8x [23], C* [19], etc.
2) Intelligent runtime support and libraries: *Express* [20], Linda, Helios, etc.[5],[6]
3) Parallelizing translators or compilers: [3],[4],[10],[21]

ASPAR represents a system of type (3) collaborating with *Express*, a system of type (2).In a similar manner to the "vectorizing compiler" and "autotasking libraries" [17],[18] this system allows users who have little interest in the details of distributed memory parallel computers to use them.

Much work concerning parallelization and optimization for vectorizing and shared memory parallel computers has been done. [1],[11],[13],[14],[15],[22],[23] In addressing distributed memory machines some researchers have adopted the approach of extending techniques of other parallel architectures such as "A virtual shared memory machine on a distributed memory machine"[10] In contrast with such work, our approach is more application-oriented and more straightforward. We attempt to find an explicit global data decomposition strategy for a sequential code by symbolic analysis, and then figure out appropriate communication requirements. We find that high level communication is more efficient than a simple "point-to-point" interface in terms not only of ease of parallelization but also for enhancing the performance of the parallelized program.

It is naive to expect that all sequential programs can be automatically parallelized. One of our goals, therefore, is to delineate those applications and those software engineering practices which allow automatic parallelization. In order to help guide the user to methods which will allow for successful parallelization *ASPAR* provides interactive graphical tools which allow the user to "visualize" the parallelism of a sequential program and understand problems preventing its parallelization.

2. System Overview

Fig1 shows an overall picture of the system. The "bold" boxes represent the components of the automatic parallelizer. The complementary graphical analysis tools, "`mapv`" and "`ftool`" (described in section 6) are shown relative to the parallelizer at the

0-8186-2113-3/90/0000/1105$01.00 © 1990 IEEE

appropriate stages of the parallelization process. Support tools supplied by the basic *Express* system and which play an important role in the parallelization are shown underlined.

The "Pre-processor" is a standard C pre-processor use to remove '#' directives."Parser", the second phase of *ASPAR*, contains a C language parser and lexical analyzer and is used to break down a piece of C code to its "parse-tree" containing a significantly simplified representation of the original program. These two phases contain all the language dependencies for the programming language being parallelized.

The "Pre-analyzer" is an aggregation of techniques whose purposes are both to prepare the "parse-tree" for further complex manipulations and to improve the parallelism of original code. Its basic tasks are:

Link

Individual parse-trees for single source files must be combined to form a representation of the whole program. This is similar in concept to a conventional object module linker except that it operates on the "parse-trees" in some internal representation rather than machine code object files.

Loop normalization & "inhibitor" checking

Loop normalization is a technique commonly used in optimizing and vectorizing compilers. Pointer expressions," `union`" and `goto`'s which make symbolic analysis impossible are inhibitors for *ASPAR* preventing parallelization. Flow control statements ("`if`"), nested loops and procedure calls do not necessarily inhibit parallelization.

Other

Several other common techniques are effective in helping subsequent symbolic dependency tests. Forward substitution, induction variable recognition [22] and compound statement fissions are used. Note that no loop reconstruction [13],[22],[23] techniques are applied since these methods are not particularly useful *at the parallelization stage* for a distributed memory architecture. They can, however, be usefully employed in the final node compiler after parallelization has been completed.

The "Analyzer" module is used to extract the parallelism from the sequential program and its functions are described in detail in the next section.

The "Translator" is responsible for modifying the original sequential program by the addition of suitable calls to the runtime library. Note that this translation is "source-to-source" to enhance the portability of the parallelized code and also to facilitate later "hand-tuning" by the user.

To further enhance the performance and portability of the parallelized code we have adopted the *Express* runtime system for our work, shown in the bottommost box of Fig. 1. This system has the advantage of already providing many automated decomposition tools and a correspondingly matched communication, I/O and graphics system which can easily be used in performing the types of decomposition used by *ASPAR*. The availability of high-level tools such as the debugger and performance analysis systems is also an advantage in providing the user an easy transition from sequential to parallel programming.

3. Symbolic analysis

3.1 "Yet another" dependency analysis tech-

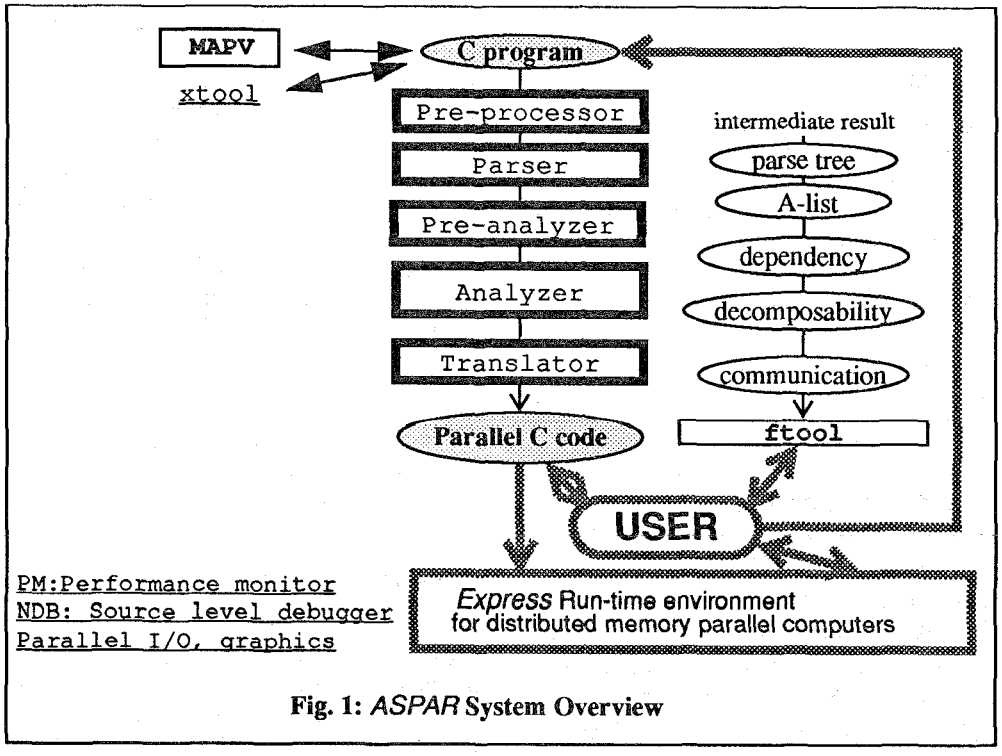

Fig. 1: *ASPAR* System Overview

C statement	A-list	
	Left-side	Right-side
`for(i=0;i<N;i++)`	`i`	`N`
`if(aa[i]<sh)`	`NULL`	`aa[i],i,sh`
`aa[c[i]]=10;`	`aa[c[i]]`	`c[i],i`
`cc=10;`	`c`	`NULL`
`sfunc(x,y);`	`NULL`	`sfunc(x,y),x,y`

Figure 2. C statements and the corresponding A-list entries

nique

We chose to exploit the loop level parallelism exhibited by the C language "`for`" statement as the basis for our parallelization. The parallelism implicit in such constructs is typically extracted by dependency analysis - if loop iterations show no inter-dependencies they can be executed in parallel.

The technique of "data dependency analysis" has been explored by many researchers [13],[14],[15],[22],[23] and is used as the vehicle of advanced compilers for various type of parallel computers such as CRAY and Alliant. A typical technique is to construct a so-called "dependency-tree" which represents every type of dependency implied by the statements of the original sequential code. In contrast with such an elegant but complex technique, the dependency analysis required for a distributed memory parallel computer can be simplified by assuming that no loop reconstruction (loop fusion, loop distribution, loop interchanging, etc.[22],[23]) is necessary. One advantage of this fact is that C language constructs which typically prevent vectorization of "`for`" loops will be allowed by *ASPAR*. "Loop carried dependencies", are the only conventionally recognized dependencies which prevent parallelization.

As a result the "A-list" (Atom list) method which represents only the flow of variables through each statement (including flow control nad loop headers) is quite convenient for performing the analysis - a much simpler technique than building the full dependency tree.

In cases where loops involve flow control statements dependencies are examined for each potential execution path by "stacking" the A-lists dynamically. In this way nested loops and procedure calls from within loops are reasonably simply dealt with.

Note that not all "loop carried dependencies" inhibit parallelization. The availability of such collective communication primitives as "`excombine`" allows loops with the "recurrence" dependency to be parallelized even when they would normally be forbidden.

3.2 Local and Global Decomposability analysis

The distinguishing feature of a distributed memory parallel computer is the availability of no shared memory. As a result the machines are cheap and simple to build and can be scaled to very large numbers of processors. Unfortunately their programming requires interprocessor communication which, if done carelessly, can result in communication overheads dominating the amount of time spent by the CPU's in useful work. A good strategy for such architectures is that of "domain decomposition" and it is this which *ASPAR* attempts to implement in translating sequential programs.

A major component of *ASPAR*, therefore is devoted to analysis of the possible global decomposition strategies. Once this is done interprocessor communication becomes well-defined and can be tackled separately.

To solidify the issues surrounding the decomposability analysis consider a normalized "`for`" loop.The range of the loop can be defined by integer constants or variables, and the loop increment is +1. Assume that this loop has no "loop carried dependencies" so that parallelization is possible. Further assume that the loop contains reference to an array "`AR`", indexed by some function "`f()`". Symbolically this specification takes the form:

```
for(e1(i); e2(i); e3(i)) {
    Body(i, AR[...][f(i)][...]);
}
```

where

`i`	Loop index
`Body(i, AR)`	One or more source statements involving the loop index and array "`AR`".
`f(i)`	Array indexing function

Now we proceed to consider each index of array "`AR`" independently. Denote by "F_1" the set of array indexing functions used to address elements of `AR` throughout in this loop

$$F_1 = \{f_1(i), f_2(i), \ldots, f_n(i)\}$$

We define this index of array `AR` to be "locally decomposable", (LDC) if and only if each indexing function can be expressed in the form, where `a` and `b` are interger constants.

$$f_k(i) = a_k * i + b_k$$

and all a_k are equal $1 <= k <= n$.

Furthermore define this index of array `AR` to be "globally decomposable", (GDC) if and only if it is locally decomposable in every loop, `1`, each of these loops is parallelizable, and the values of a_k are identical in each case.

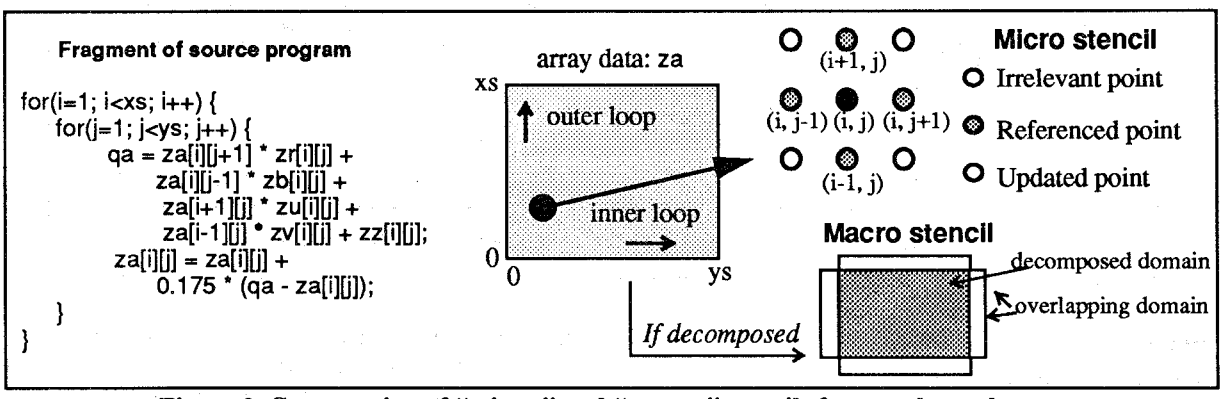

Figure 3. Construction of "micro-" and "macro-" stencils from update scheme

3.3 Loop range variation

Having made these decisions we need to further check that the range of array indices used by the program is consistent in each instance. All loops which involve the globally decomposable array "AR" should have the same range of indices. In the case where loop indices are constants this is easy to verify. Where loop ranges are indicated by variables it is impossible to statically determine whether or not the ranges are equal. One option would be to implement a dynamic load balancing strategy which would be able to take care of potential changes in array usage. This solution is, however, extremely costly to implement both in terms of human effort and also its impact on algorithmic performance. Instead *ASPAR* makes a simpler assumption that array ranges specified by variables will remain constant. This assumption is valid in the vast majority of the regular applications at which *ASPAR* is directed.

3.4 Communication analysis, "stencils"

A "stencil" is a range of distances from a particular point in the mesh from which information is required to update and maintain the integrity of the data in a distributed memory architecture. In the trivial case each grid point is independent and no interprocessor communication is required when parallelizing the algorithm. In more common cases, such as the one shown in Fig. 3, a stencil can be constructed of finite size to direct the communication required by the parallel algorithm.

Fig. 3 shows a typical stencil associated with to a nested "FOR" loop. We can distinguish two types of stencil: micro and macro-stencils. The "micro-stencil" describes the update scheme for a single point in the grid whereas the macro-stencil describes the area where the various decomposed domains overlap and communication is required. In the example of Fig. 3 values from nearest neighbors are required to update the point at `(i,j)`. This means that the "micro-stencil" for the first array index is (-1:1). Similarly the stencil for the second index is seen to be (-1:1). In principle we could use these "micro-stencils" to implement a strategy in which individual grid points were communicated whenever necessary.

Using the "micro-stencils", however, it is possible to construct a "macro-stencil" which describes the areas where the entire decomposed domain "overlaps" with neighboring domains. Having done this we can simplify and optimize the inter-node communication by using the collective communication ability of *Express* to transmit all of the boundary messages to the appropriate nodes *before* entering the nested loops. The benefit is that the number of communication calls is reduced dramatically and a significant improvement in performance is obtained.

4. Parallelization models

In order to automatically parallelize a FOR loop, *ASPAR* uses only 4 types of parallelization model, each of which uses a different high-level communication function from the *Express* library [12],[20]. This in itself is an interesting result since it shows the importance of the high-level "collective" communication routines over the simple "point-to-point" communication schemes.

Each of the four strategies is briefly described with an example of the original source code and the parallelized version. For simplicity the arguments to the *Express* functions have been simplified.

4.1 Independent cycles, no communication

If a "`for`" loop has no loop carried dependency and every updated array is globally decomposable, it can be parallelized without any communication. A typical example is kernel #12 of the Livermore loop benchmark, "first difference", shown in Fig.4. Parallelizing such a loop on a distributed memory computer is equivalent to simply dividing the loop range by the number of processors available, being careful to treat the remainder correctly!. This operation is performed by the function "`AS_set_ranges`" which calculates variables "`AS_cnt`" to indicate the range of loop iterations in each node.

4.2 "Combine" type

The typical example of this case is a "reduction" loop which has only one kind of loop carried dependency known as "recurrence". Typically the operation on the data values is some simple binary operator such as ad-

dition or subtraction. The example shown in Fig. 4 is the standard scalar product taken from kernel #3 of the Livermore Loops.

To parallelize such a loop a minimal type of algorithmic modification is required since the order of operation is changed in going to the parallelized version of the code. The *Express* `excombine` function is used to recalculate the global quantity after the parallel loop operations have been completed.

4.3 "Concatenation" type

In isolation this type of loop has the same appearance as that described in section 4.1 which involved no communication: there is no loop carried dependency. If, however, some of the data in the loop is *not* decomposed elsewhere in the program it must be accumulated in every processor. This operation involves broadcasting each node's portion of the decomposed data to all others while simultaneously receiving contributions from all other nodes. It is handled by a simple call to the *Express* `exconcat` function.

A good example of the use of this technique is the conjugate gradient matrix solver described in section 5.2.

Sequential Code	Parallel Code

Type 1: No Communication
`for(k=1; k<=n; k++) {`
`  x[k] = y[k+1] - y[k];`
`}`
	`AS_set_ranges(0, n, 1, 0);`
	`for(k=1; k<=AS_cnt[0]; k++) {`
	`  x[k] = y[k+1] - y[k];`
	`}`

Type 2: "excombine"
`Q = 0.0;`
`for(k=0; k<N; k++) {`
`  Q += (Z[k] * X[k]);`
`}`
	`Q = 0.0;`
	`AS_set_ranges(0, N-1, 0, 1);`
	`for(k=0; k<AS_cnt[0]; k++) {`
	`  Q += (Z[k] * X[k]);`
	`}`
	`excombine(&Q, D_PLUS, ALLNODES);`

Type 3: "exconcat"
`for (i=0;i<elm;i++) {`
`  X[i]= 0.0;`
`  R[i] = bb[i];`
`  P[i] = bb[i];`
`}`
	`AS_set_ranges(0, elm-1, 0, 1);`
	`for (i=0;i<AS_cnt[0];i++) {`
	`  X[i]= 0.0;`
	`  R[i] = bb[i];`
	`  P[i+AS_ofst[0]] = bb[i];`
	`}`
	`AS_size[0] = sizeof(double) * AS_cnt[0];`
	`exconcat(&P[AS_ofst[0]], P, AS_size[0]);`

Type 4: "exchange"
`for(j=2; j<=6; j++) {`
`  for(k=2;k<=n;k++) {`
`  qa = za[k][j+1] * zr[k][j] +`
`  za[k][j-1] * zb[k][j] +`
`  za[k+1][j] * zu[k][j] +`
`  za[k-1][j] * zv[k][j] + zz[k][j];`
`  za[k][j] = za[k][j] +`
`  0.175 * (qa - za[k][j]);`
`  }`
`}`
	`exvchange(&za[1][2], AS_num[1], LEFT,`
	`  &za[AS_cnt[0]][2] ,AS_num[1], RIGHT);`
	`exvchange(&za[AS_cnt[0]+1][2], AS_num[1],`
	`  RIGHT,&za[2][2], AS_num[1], LEFT);`
	`exvchange(&za[2][1], AS_num[0], DOWN,`
	`  &za[2][AS_cnt[1]], AS_num[0], UP);`
	`exvchange(&za[2][AS_cnt[1]+1], AS_num[0], UP,`
	`  &za[2][2], AS_num[0], DOWN);`
	`AS_set_ranges(1, 6, 2, 0);`
	`for(j=2; j<=AS_cnt[1]; j++) {`
	`  AS_set_ranges(0, n, 2, 0);`
	`  for(k=2;k<=AS_cnt[0];k++) {`
	`  qa = za[k][j+1] * zr[k][j] +`
	`  za[k][j-1] * zb[k][j] +`
	`  za[k+1][j] * zu[k][j] +`
	`  za[k-1][j] * zv[k][j] + zz[k][j];`
	`  za[k][j] = za[k][j] + 0.175 * (qa - za[k][j]);`
	`  }`
	`}`

Figure 4. Different types of loop parallelization

4.4 "Exchange" type

The "stencil" operations described in section 3.4 require a rather different type of communication. Before any loop involving updates to decomposed data which possess a "macro-stencil" we must arrange for the regions of overlap between processors to be communicated so that the updates can occur with valid data. Typical examples of this kind are partial differential equation solvers, image processing algorithms, etc. The particular example shown in Fig. 4 is from kernel #23 of the Livermore Loops - "Implicit hydrodynamics".

Note that this loop carries dependencies: from `za[k][j-1]` to `za[k][j]` and from `za[k-1][j]` to `za[k][j]`. Normal dependency analysis would prohibit parallelizing this loop but *ASPAR* provides a special switch to enable the user to allow such parallelizations although the parallel algorithm is now subtly different from the sequential one. Alternate strategies which will be implemented in the future involve the "red-black" update scheme and the "hyperplane" technique which both allow parallelization without algorithmic modification.

Having made the decision that both loops are parallelizable and that the arrays are decomposable a standard mapping to a two-dimensional grid is invoked and appropriate boundary information is updated with the *Express* `exvchange` function.

5. Experimental results

To evaluate our methods we have processed several different types of application with *ASPAR*

5.1 Livermore loop kernels

The Livermore kernels are 24 loops from actual production codes that have been widely used to evaluate the performance of various computer systems. [8] Written originally in Fortran, the benchmarks were rewritten in C for this test. Table 1 shows the type of the application, its algorithmic complexity and the result of applying *ASPAR*. The complexity here is defined as the number of computations required to complete the main procedure on an ideal parallel computer as a function of input size [8]. The complexity of a vector sum, for example loop #11, is O(N) on a sequential machine. It will, however, be O(logN) if it can be parallelized in a binary tree fashion. If the iterations of a loop can be executed completely independently its complexity is O(1).

These tests determined that 14 of the loops were parallelized by *ASPAR*. Examination of the failed cases showed that some were not parallelizable, even by hand. In each successful case the parallel algorithm correctly matched the expected complexity for the ideal machine.

5.2 A Conjugate Gradient linear equation solver

While the Livermore loops provide an indication of the basic capabilities of *ASPAR* it is important to realize that "real" programs present significantly more complex problems, not merely because they use more complex algorithms but because a large application has more "baggage" surrounding it which can inhibit parallelization in many ways.

Table 1. Results of applying *ASPAR* to the Livermore loop kernels
(A "Yes" in the final column indicates that automatic parallelization was possible)

Loop number	Type of Application	Complexity	Parallelized?
1	Hydrodynamics	O(1)	Yes
2	Incomplete Cholesky	O(logN)	No
3	Inner product	O(logN)	Yes
4	Banded linear equations	O(logN)	No
5	Tridiagonal elimination	O(N)	No
6	Linear recurrence relations	O(N)	No
7	Equation of state	O(1)	Yes
8	A.D.I	O(1)	Yes
9	Numerical Integration	O(1)	Yes
10	Numerical Differentiation	O(logN)	Yes
11	Finite sum	O(logN)	No
12	Finite difference	O(1)	Yes
13	2-D "particle in a cell"	O(N)	No
14	1-D "particle in a cell"	O(1)	Yes
15	Prime example	O(1)	Yes
16	Search loop from Monte Carlo	O(1)	Yes
17	Implicit conditional computation	O(N)	No
18	2-D explicit hydrodynamics	O(1)	Yes
19	Linear recurrence relation	O(logN)	No
20	Discrete ordinates transport	O(N)	No
21	Simple matrix calculations	O(logN)	Yes
22	Planck distribution	O(1)	Yes
23	2-D implicit hydrodynamics	$O(N^2)$	Yes
24	Minimization	O(logN)	No

To look at the potential problems we tested *ASPAR* on a complete algorithm - a conjugate gradient matrix solver, the abbreviated source code for which is shown in Fig. 5. It is important to note that this is a "banded" solver rather than one for full matrices, which present considerably fewer problems.

The parallelized code is shown in Fig. 6 The "`/*$...$*/`" strings are comments inserted by *ASPAR*. Modifications are usually indicated by variables/functions whose names start with "`AS_`". The inserted routines whose names begin with "`ex`" are calls to the *Express* runtime library inserted automatically by *ASPAR*. The variable "`P1`" indicates a decomposed array index.

Note that all arrays are decomposed except "`P`". This array is an argument to the "`band_multi`" function and is used with too complex an index for *ASPAR* to understand.

The array "`ar`" could potentially be decomposed in two dimensions. due to the banded structure of the matrix it is better on grounds of global efficiency, however, to decompose only along the first dimension as shown in Fig. 7. As a result the two inner loops of "`band_multi`" although the outer loop is parallelized despite the complex flow of control and nested

```
/****** Solver for linear equations by CG method ******/
#include "stdio.h"
#define SZ 400
#define BND 100
#define ep (double)1.0e-14
double eps;
double ar[SZ][BND],bb[SZ],xx[SZ],P[SZ],X[SZ];
double R[SZ],newR[SZ],newP[SZ],KP[SZ],newX[SZ];
void band_multi(A,x,b,elm,hbnd)
{
    int elm,hbnd;
    double A[SZ][BND],x[],b[];
{
    int i,j,band;
    double sum;
    band=hbnd*2-1;
    for(i=0;i<elm;i++){
        sum=0.0;
        if(i<hbnd)
            for(j=0;j<band;j++)sum+=A[i][j]*x[j];
        else
            for(j=0;j<band;j++)sum+=A[i][j]*x[(j+1+(i-hbnd))];
        b[i]=sum;
    }
}
main()
{
    int i,j,k,elm,hbnd,param[2];
    double sum1,sum2,rd,alpha,beta;
    read_data(ar,xx,bb,param);
    elm=param[0],hbnd=param[1];
    for(i=0;i<elm;i++){
        X[i]=0.0;
        P[i]= bb[i];
        R[i]=bb[i];
    }
    sum1=0.0;
    for(i=0;i<elm;i++)
        sum1+=bb[i]*bb[i];
    eps=ep*sum1;
    k=1,rd=(double)100000.0;
    while(rd>eps){
        band_multi(ar,P,KP,elm,hbnd);
        sum1=sum2=0.0;
        for(i=0;i<elm;i++)sum1+=R[i]*R[i];
        for(i=0;i<elm;i++)sum2+=P[i]*KP[i];
            alpha=sum1/sum2;
        for(i=0;i<elm;i++)newX[i]=X[i]+alpha*P[i];
        for(i=0;i<elm;i++)newR[i]=R[i]-alpha*KP[i];
        sum2=0.0;
        for(i=0;i<elm;i++)sum2+=newR[i]*newR[i];
        beta=sum2/sum1;
        for(i=0;i<elm;i++)
        newP[i]=newR[i]+beta*P[i];
        rd=sum1;
        for(i=0;i<elm;i++){
            P[i]=newP[i];
            R[i]=newR[i];
            X[i]=newX[i];
        }
        k++;
    }
    print_result(X);
}
```

Figure 5 A sequential algorithm to perform Conjugate Gradient iteration

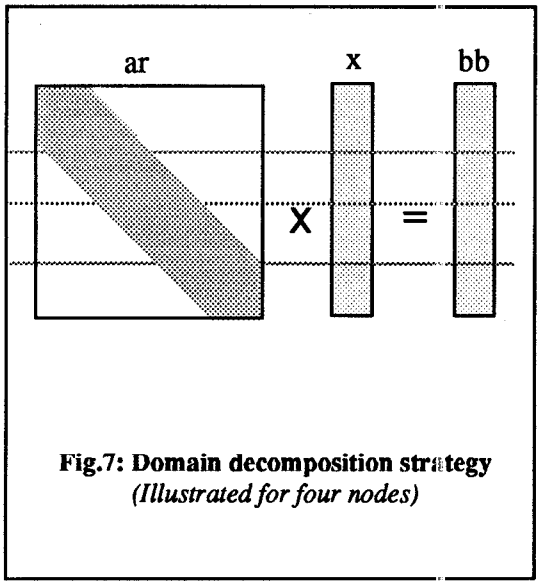

Fig.7: Domain decomposition strategy
(Illustrated for four nodes)

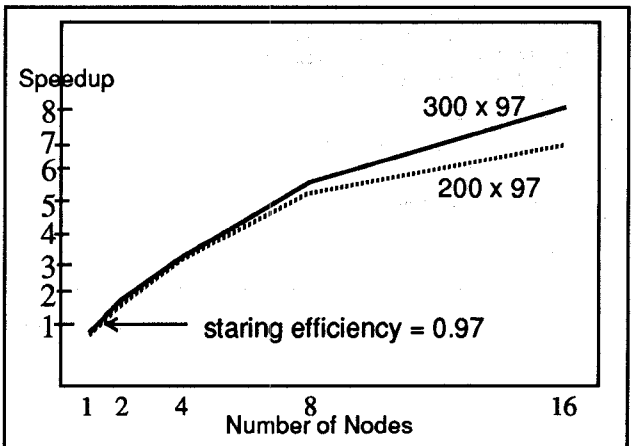

Fig. 8: Speedup of automatically parallelized code.
The performance of the Conjugate Gradient (Fig. 6) is shown for matrices of order 200 and 300 with bandwidth 97. Data collected from an NCUBE/10

```
*$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$*/
/*$  AUTOMATICALLY PARALLELIZED PROGRAM     $*/
/*$  Decompose this as 1-D problem          $*/
/*$  P1: proc.number of 1st dimension       $*/
/*$  AS_procs:Number of processors          $*/
/*$  AS_lgc:Logical processor location (logical no)  $*/
/*$  AS_lst:Table of logical-physical proc.numb      $*/
/*$  AS_ofst:Offset for undecomposed data   $*/
/*$  AS_cnt:No.of iteration after parallelized $*/
/*$$$$$$$$$$----EXPRESS header----$$$$$$$$$$$$$$$$$*/
#defineP1(1)    /*-You may change it--*/
#include <express.h>
struct nodenv env;
int AS_lgc[3],AS_procs[3],AS_lst[100],
    AS_type=123,AS_ofst[3],AS_cnt[3],AS_size[3];
/*******Solver for liner equations by C G method *******/
#include "stdio.h"
#define SZ 400
#define BND 100
#defineep(double)1.0e-14
double eps;
double ar[SZ/P1][BND],bb[SZ/P1],xx[SZ/P1],P[SZ],X[SZ/P1];
double R[SZ/P1],newR[SZ/P1],newP[SZ/P1],KP[SZ/P1],
    newX[SZ/P1];
void band_multi(A,x,b,elm,hbnd)
    int elm,hbnd;
    double A[SZ/P1][BND],x[],b[];
{
    int i,j,band;
    double sum;
    band=hbnd*2-1;
/*$ PARALLEL $*/
    AS_set_range(0,elm-1,0,1);
    for(i=0;i<AS_cnt[0];i++){
        sum=0.0;
    if(i+AS_ofst[0]<hbnd)
/*$ Sequential: Data-Decomp Strategy2 $*/
    for(j=0;j<band;j++)
        sum+=A[i][j]*x[j];
    else
/*$ Sequential: Data-Decomp Strategy2 $*/
        for(j=0;j<band;j++)
            sum+=A[i][j]*x[(j+1+(i+AS_ofst[0]-hbnd))];
        b[i]=sum;
    }
}
main()
{
    int i,j,k,elm,hbnd,param[2];
    double sum1,sum2,rd,alpha,beta;
/*$$$$$$$$ Initialization for EXPRESS $$$$$$$$$*/
    exparam(&env);
    AS_procs[0]=env.nprocs;
    exgridinit(1,AS_procs);
    exgridcoord(env.procnum,AS_lgc);
    exconcat(AS_lgc,4,AS_lst,4,NULLPTR,ALLNODES,
            NULLPTR,&AS_type);
/*$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$*/
    read_data(ar,xx,bb,param);
    elm=param[0],hbnd=param[1];
/*$ PARALLEL $*/
    AS_set_range(0,elm-1,0,1);
    for(i=0;i<AS_cnt[0];i++){
        X[i]=0.0;
        P[i+AS_ofst[0]]=bb[i];
        R[i]=bb[i];
    }
/*$ CONCATENATEP $*/
    AS_size[0]=sizeof(*P)*AS_num[0];
    exconcat(&P[AS_ofst[0]],AS_size[0],P,AS_size[0],
            NULLPTR,env.nprocs,AS_lst,&AS_type);
    sum1=0.0;
/*$ PARALLEL $*/
    AS_set_range(0,elm-1,0,1);
    for(i=0;i<AS_cnt[0];i++)
        sum1+=bb[i]*bb[i];
/*$ COMBINE sum1 by PLUS $*/
    excombine(&sum1,f_add,sizeof(sum1),1,ALLNODES,
            NULLPTR,&AS_type);
    eps=ep*sum1;
    k=1,rd=(double)100000.0;
    while(rd>eps){
        band_multi(ar,P,KP,elm,hbnd);
        sum1=sum2=0.0;
/*$ PARALLEL $*/
        AS_set_range(0,elm-1,0,1);
        for(i=0;i<AS_cnt[0];i++)sum1+=R[i]*R[i];
/*$ COMBINE sum1 by PLUS $*/
        excombine(&sum1,f_add,sizeof(sum1),1,ALLNODES,
                NULLPTR,&AS_type);
/*$ PARALLEL $*/
        AS_set_range(0,elm-1,0,1);
        for(i=0;i<AS_cnt[0];i++)
            sum2+=P[i+AS_ofst[0]]*KP[i];
/*$ COMBINE sum2 by PLUS $*/
        excombine(&sum2,f_add,sizeof(sum2),1,ALLNODES,
                NULLPTR,&AS_type);
        alpha=sum1/sum2;
/*$ PARALLEL $*/
        AS_set_range(0,elm-1,0,1);
        for(i=0;i<AS_cnt[0];i++)
            newX[i]=X[i]+alpha*P[i+AS_ofst[0]];
/*$ PARALLEL $*/
        AS_set_range(0,elm-1,0,1);
        for(i=0;i<AS_cnt[0];i++)newR[i]=R[i]-alpha*KP[i];
        sum2=0.0;
/*$ PARALLEL $*/
        AS_set_range(0,elm-1,0,1);
        for(i=0;i<AS_cnt[0];i++)sum2+=newR[i]*newR[i];
/*$ COMBINE sum2 by PLUS $*/
        excombine(&sum2,f_add,sizeof(sum2),1,ALLNODES,
                NULLPTR,&AS_type);
        beta=sum2/sum1;
/*$ PARALLEL $*/
        AS_set_range(0,elm-1,0,1);
        for(i=0;i<AS_cnt[0];i++)
            newP[i]=newR[i]+beta*P[i+AS_ofst[0]];
        rd=sum1;
/*$ PARALLEL $*/
        AS_set_range(0,elm-1,0,1);
        for(i=0;i<AS_cnt[0];i++){
            P[i+AS_ofst[0]]=newP[i];
            R[i]=newR[i];
            X[i]=newX[i];
        }
/*$ CONCATENATE P $*/
        AS_size[0]=sizeof(*P)*AS_num[0];
        exconcat(&P[AS_ofst[0]],AS_size[0],P,AS_size[0],
                NULLPTR,env.nprocs,AS_lst,&AS_type);
        k++;
    }
    print_result(X);
}
```

Figure 6. Parallel Conjugate Gradient Algorithm as generated by *ASPAR*

loop constructs.

The current version of *ASPAR* was unable to correctly parallelize the bounds on the I/O constructs and so these had to be modified by hand in order for the algorithm to function correctly. The compilation/linking process involved no special precautions and some sample performance data for the parallel algorithm are shown in Fig. 8. We feel that these results are extremely interesting. The problems solved are relatively small and, considering the banded nature of the matrix, the performance gains are quite strong. With relatively minor improvements in the interface between *ASPAR* and *Cubix*, the parallel I/O system of *Express* it should be possible to completely automate the parallelization of this algorithm.

6. Support Tools

As well as the automatic parallelizer *ASPAR* offers several supporting utilities which aid users whose programs are not immediately parallelized.

"`ftool`" is a utility which allows the user to interactively visualize the flow of data/control on a window and the relevant source code on another window. It also presents information about the loop carried dependencies and the possibilities of data decomposition corresponding each FOR loop. An example is shown in Fig.9.

"`mapv`" is a set of tools which allow the user to interactively visualize the patterns in which the application accesses memory with blue colored reference and red colored update. This type of information is central to achieving good domain decomposition strategies. The operation of this utility is a two-phase process.

1) Auto-profiling
   A profiling utility is used to "instrument" the sequential program and record references to particular data structures indicated by the user.
2) Visualization
   Once the sequential program has been executed a data file is created which contains memory access pattern data. This can be visualized with the "`vtool`" program which allows the user to "play back" the history of memory accesses made by the sequential program.

7. Conclusions

ASPAR is a powerful tool which is able to automatically parallelize a significant sequential program for a distributed memory parallel computer. It is able to not only parallelize the basic sequential code but also modify its algorithm for parallel execution. Its basic abilities lie in sophisticated symbolic analysis coupled to a knowledge base regarding "domain-decomposition" parallelization. The *Express* runtime environment both simplifies the task of parallelization and also allows the user the flexibility to run the parallelized programs on a wide variety of parallel computer systems and network based workstations.

Generally speaking, *ASPAR* works extremely well on applications which use regular meshes. Even when failing to parallelize a problem *ASPAR* generally issues adequate diagnostics to correct the problem or modify the coding to allow parallelization. Particularly important in this area are the graphical display tools which allow the user to visualize problem areas and interactively modify them.

Although *ASPAR* is unable to parallelize all C programs its current abilities, especially in regard to the efficiency of the parallelized algorithms, are very encouraging. One area in which work remains to be done is the interaction with the I/O system. Once this is accomplished *ASPAR* should be capable of completely parallelizing quite sophisticated C codes.

8. Acknowledgments

K. Ikudome would like to thank the Caltech Concurrent Computation Project for enabling him to work as a visiting research fellow on leave from Nippon Steel Co.

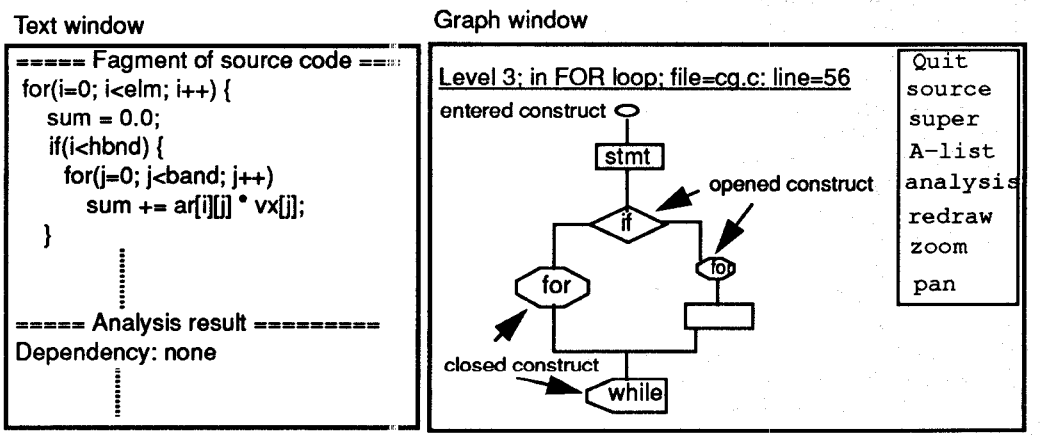

Fig.9: A typical image drawn by `ftool`

9. References

[1] T.Brandes, "Determination of dependencies in a knowledge-based parallelization tool", *Parallel Computer*, **vol. 8**, 1988, 111-119.

[2] M.Kumar, "Measuring Parallelism in Computation-Intensive Scientific/Engineering Applications", *IEEE Trans. Computer*, Sep 1988,1088-1098.

[3] U.Kremer, H.J. Bast, M.Gerndt, H.P. Zima, "Advanced tools and techniques for automatic parallelization", *Parallel Computing*, **vol. 7**, 1988,387-393.

[4] H.P.Zima, H.J.Bast, M.Gerndt, "SUPERB: A tool for semi-automatic MIMD/SIMD parallelization", *Parallel Computing*, **vol. 6**, 1988, 1-18.

[5] H.Muhlenbein, O.Kramer, F.Limburger, M.Mevenkamp, S.Streitz, "MUPPET: A programming environment for message-based multiprocessors", *Parallel Computer*, **vol. 8**, 1988, 201-221.

[6] R.Mirchandaney, J.H.Saltz, R.M.Smith, D.M.Nichol, K.Crowley, "Principles of Runtime Support for Parallel Processors", *Proceedings of the ACM*, 1988.

[7] Th.Ruppelt, G.Wirtz, "From Mathematical Specification to parallel program on a message-based system", *Proceedings of the ACM*, 1988.

[8] J.T.Feo, "An analysis of the computational and parallel complexity of the Livermore Loops", *Parallel Computing*, **vol. 7**, 1988, 163-185.

[9] J.Dongarra, D.C.Sorensen, K.Connelly, J.Patterson, "Programming methodology and performance issues for advanced computer architectures", *Parallel Computing*, **vol. 8**, 1988, 41-58.

[10] D.Callahan, K.Kennedy, "Compiling programs for distributed-memory multiprocessors", *Rice University Computer Science Report*, COMP TR88-74, Aug. 1988.

[11] W.R.Cowell, C.P.Thomson, "Transforming Fortran DO loops to improve performance on vector architectures", *ACM transactions on Mathematical software*, **vol. 12** no.4, Dec. 1986, 324-353.

[12] G.C.Fox, M.Johnson, G.Lyzenga, S.W.Otto, J.Salmon, D.Walker, "Solving problems on concurrent processors", published Prentice Hall, 1988.

[13] D.J.Kuck, R.H.Kuhn, D.A.Padua, B.Leasure, and M.Wolfe, "Dependence graphs and compiler optimizations", in Proc. 8th ACM Symp. Principles Programming Languages, Jan. 1981, pp. 207-218.

[14] C.Polychronopoulos, "Compiler optimizations for enhancing parallelism and their impact on architecture design", *IEEE trans. on Computers*, **vol 37**, no 8 1988, 991-1004

[15] C.Polychronopoulos, D.Kuck, D.Padua, "Utilizing multidimensional loop parallelism on large-scale parallel processor systems", *IEEE trans. on Computers*, **vol.38**, no.9 1989, 1285-1296

[16] O.Brewer, J Dongarra, D.Sorensen, "Tools to aid in the analysis of memory access patterns for Fortran programs", *Parallel computing*, **vol 9**, 1988/89 25-35

[17] Cray Research Inc. "UNICOS Autotasking user's guide", SN-2088 CFT77 3.1

[18] Cray Research Inc. "Cray X-MP user's manual"

[19] Thinking Machines Co. "Using the Connection Machine System", **vol 1 & 2**, 1989

[20] *Parasoft* Co. "*Express* user's manual" 1989

[21] A.E.Terrano, S.M.Dunn, J.E.Peters, "Using an architectural knowledge base to generate code for parallel computers", Communications of the ACM, **vol.32**, no 9, 1989, 1065-1072

[22] D.A.Padua, M.J.Wolfe, "Advanced compiler optimizations for supercomputers", Communications of the ACM, **vol.29**, no 12, 1986, 1184-1201

[23] R.Allen, Ken Kennedy, "Automatic translation of Fortran programs to vector form", ACM trans. on Programming Languages and Systems, **vol.9**, no.4, Oct. 1987, 491-542