Optimization and Parallelization of a Commodity Trade Model for the SP1, Using Parallel Programming Tools

Donna Bergmark and Marcia Pottle
Cornell Theory Center *

March 25, 1994

Revised: July 26, 1994

Abstract

We compare two different approaches to parallelization of Fortran programs. The first approach is to optimize the serial code so that it runs as fast as possible on a single processor, and then parallelize that. The second approach is to parallelize the program immediately, and then optimize the parallel version.

In this paper a variety of parallel programming tools is used to obtain an optimal, parallel version of an economic policy modeling application for the IBM SP1. We apply a new technique called Data Access Normalization; we use an extended ParaScope as our parallel programming environment; we use FORGE 90 as our parallelizer; and we use KAP as our optimizer. We make a number of observations about the effectiveness of these tools.

Both strategies obtain a working, parallel program, but use different tools to get there. On this occasion, both KAP and Data Access Normalization lead to the same critical transformation of inverting four of the twelve loop nests in the original program. The next most important optimization is parallel I/O, one of the few transformations that had to be done by hand. Speedups are obtained on the SP1 (using MPLp communication over the High Performance Switch).

Keywords: multiprocessors, program transformations, parallel programming tools, data access normalization, ParaScope, Lambda Toolkit, Fortran, HPF, FORGE, SP1, SPMD, KAP, parallel I/O, pedLambda, data parallel, loop distribution, loop fusion, trace analyzers

CTC94TR181

6/94 (revised 7/94)

*This work was partially supported by NSF New Technologies project ASC 9201498, entitled "The Evaluation and Deployment of ParaScope".

Contents

1 *The Commodity Trade Model*

Nagurney, Nicholson, and Bishop [13] describe a spatial market model that handles tariffs and asymmetric nonlinear supply and demand price functions. Given a set of supply markets and demand markets and supply and demand price functions, along with a set of fixed *ad valorem* tariffs and transportation costs, one can use the model to derive the number of units that will be shipped between individual markets. Supply prices and demand prices are assumed to vary with supply and demand. The methodology used for the formulation and computation of the model is the theory of variational inequalities. Figure 1 shows an implementation of the algorithm used to solve the model.

The solution to the problem is an iterative one, in which quantities shipped, supply prices, and demand prices are repeatedly adjusted until the system reaches a steady state. The computation can be time-consuming, and so has been parallelized for various platforms. We are interested in parallelizing it for the IBM 9076 Scalable POWERparallel System (SP1) [9] using various parallel programming tools.

In particular, we wish to compare two different approaches to parallelizing the code. The first approach is to optimize the serial code so that it runs as fast as possible on a single processor, and then parallelize that. The second approach is to parallelize the program immediately, and then optimize the parallel version. The two approaches both result in a working parallel program, but use different tools to get there. It is unknown whether one approach produces a better program than the other, or whether one approach is easier than the other.

2 *Problem, Program, and Data Analysis*

The alogrithm used to solve the model is described, motivated, and proved to converge in [13]. The original sequential version of the Fortran code (`mprojtarg.f`) is shown in Appendix A.

The algorithm works as follows. First, the data are read in. Then, starting with Q $\in \Re_{+}^{mn}$=0, compute for each market i the price of supply, $\pi_i(s)$, as a function of current supply s, and the demand price $\rho_j(d)$ as a function of current demand. This leads to a trial shipment structure $\bar{Q}$, which represents $\bar{s}$ and $\bar{d}$, the new supply and demand. That in turn sets the prices $\pi(\bar{s})$ and $\rho(\bar{d})$,

SOLVING FOR SHIPMENTS Q GIVEN TARIFFS τ

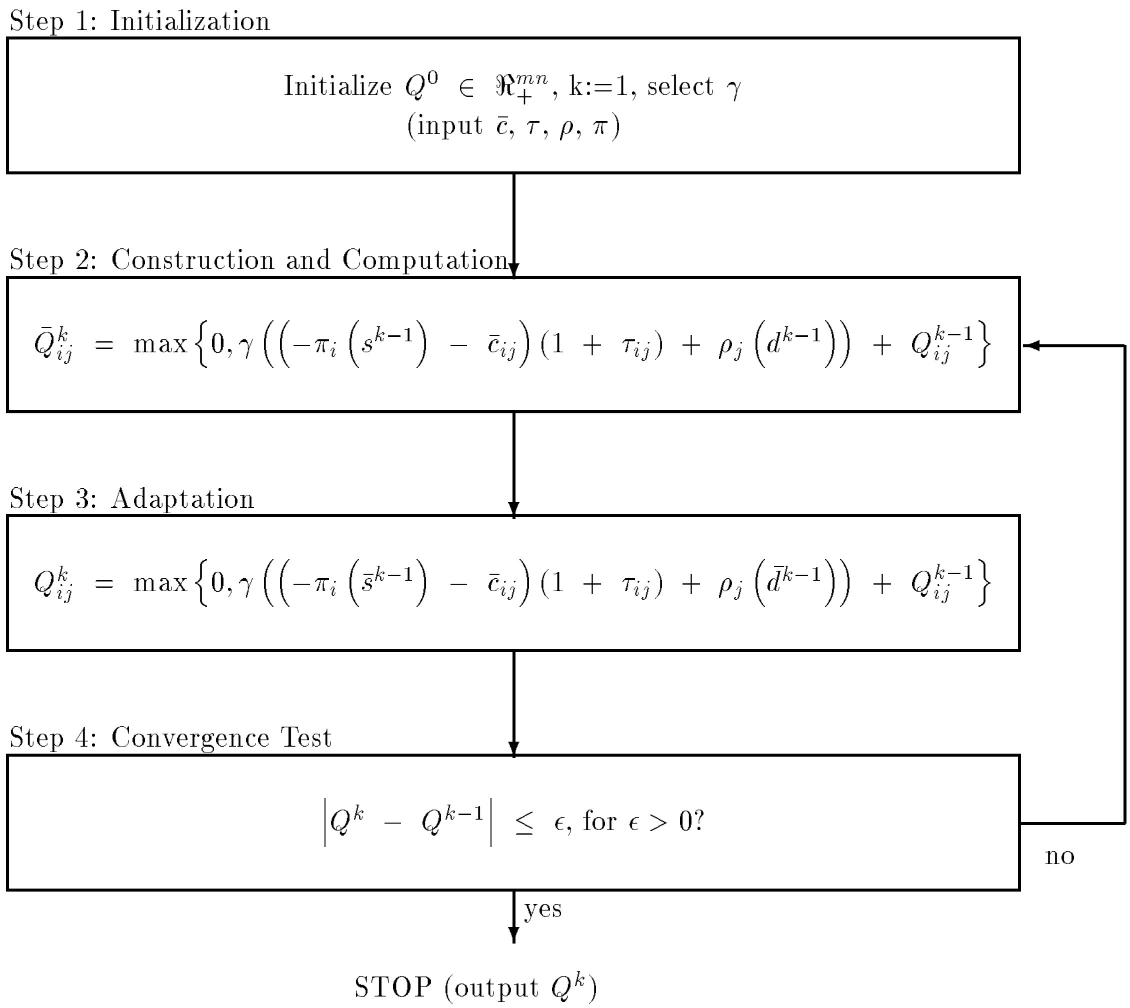

Figure 1: (See Table 1 for explanation of symbols.) Given $\bar{c}$, τ, ρ, and π, compute Q^k, the shipment from sources $i \in 1..m$ to demand markets $j \in 1..n$. For all supply and demand market pairs (i,j), each of $\bar{Q}$ and Q can be computed concurrently. d and s are computed from Q, e.g. d_j = sum of everything shipped to market j, and s_i = sum of everything shipped from market i. ρ and π are functions of d and s. The first step, construction and computation, is a trial $\bar{Q}^k$ based on current demand and supply prices as determined by s^{k-1} and d^{k-1}. This determines the new supply and demand, s and d. The second step is to recompute supply and demand prices and update the system to Q^k. That is the end of one time step.

Symbol	Definition	Program Representation
m	number of supply markets	NSUPM
n	number of demand markets	NDEMM
k	iteration number	
Q, $\bar{Q}$	units shipped from i to j	APATH(NSUPM,NDEMM)
Q^{k-1}	prev. shipment from i to j	APATHO(NSUPM,NDEMM)
τ_{ij}	tariff for shipments from i to j	TAR(NSUPM,NDEMM)
s_i	supply at market i	SUPPLY(NSUPM)
d_j	demand at market j	DEMAND(NDEMM)
π_i	supply price at supply market i	SUPP(NSUMP)
ρ_j	demand price at demand market j	DEMP(NDEMM)
$\bar{c}_{ij}$	cost of transport from i to j	THXLIN(NSUPM,NDEMM)
γ	parameter for Lipschitz continuity	0.0001
	equation components for π	SHXLIN(NSUPM), IND1(NSUPM,5)
	equation components for ρ	DHXLIN(NDEMM), IND2(NDEMM,5)

Table 1: Symbols used in the algorithm, their meaning, and their representation in program **mprojtarg**. See Figure 2 for how the variables are to be laid out in distributed memory.

which determine Q, the final shipment structure for this timestep.

From the implementation in Figure 1, it is apparent that for all demand and supply pairs (i, j) the elements of Q (and $\bar{Q}$) can be generated in parallel. In theory, one could use up to $m \times n$ processors to compute Q. However, since demand is the sum across j and supply is the sum across i, and since they are both needed to compute the prices that in turn are used to compute Q, the algorithm also requires global communication. The actual code uses indirect addressing to compute the supply and demand prices, so the global communication required is not necessarily between nearest neighbors.

The behavior of the original sequential version of `mprojtarg.f` on several datasets is tabulated in Table 2. The size of each dataset is given by NDEMM and NSUPM, the number of demand and supply markets. Note that the number of iterations to convergence, ITCNT, is data dependent, not size dependent. The length of each iteration, however, is a function of dataset size, growing as the product of NDEMM and NSUPM.

A loop analysis and a hot spot analysis conclude our initial inspection of the program. The loop analysis is useful because many of the parallel programming tools at the Cornell Theory Center

PROGRAM & DATA CHARACTERISTICS

	NSUPM	NDEMM	ITCNT	ICT	ITC	PERC	serial time on the SP1
Dataset 1	100	100	4611	10,000	194	1.94	1–3 minutes
Dataset 2	200	200	2951	40,000	566	1.415	7 minutes
Dataset 3	300	300	2796	90,000	153	.1700	14–15 minutes
Dataset 4	400	400	4140	160,000	330	.0206	52 minutes
Dataset 5	500	500	2825	250,000	7	.0028	52–53 minutes

ITCNT = `1 more than the number of iterations before convergence`
ICT = `number of elements that have converged`
ITC = `the number of non-zero elements in` Q `at the end of the run`
PERC = `itc / (nsupm*ndemm) = % of elements that are non-zero`

Table 2: Characteristics of the five datasets available for the **mprojtarg** code. Note that the number of iterations to convergence is data dependent. The exact time of the serial run varies depending on system load and compile time options.

are Fortran DO loop based. (For a recent discussion of these tools, see Tornè [19].) The original **mprojtarg** program contains 26 simple DO loops and one IF loop, as determined by running PAT [1] on the Cornell Theory Center's AFS cluster:

```
pat -l mprojtarg.f
```

The 26 loops consist of twelve double loops and two single loops (Table 3). The IF loop corresponds to the single loop shown in Figure 1; it branches back to statement 1188 at each time step until convergence is reached.

The time consumed by each loop nest can be derived from a timing report generated by FORGE 90 Baseline [15], using the following commands on the cluster (in this example, the package name in FORGE is "`mproj`"):

```
forge90          <=== (import mprojtarg.f,instrument it,
                         and dump it into file mproj_clk.f)

xlf  -o mproj_clk  -O2  mproj_clk.f  -lforgetim  <=== (compile and link)

mproj_clk > mproj.tim  <=== (generate a timing report)
```

Loop	depth		description		inclusive
25	1		Performs I/O (input)		
35	1		Performs I/O (input)		
45	2		Performs I/O (input)		1.1 %
78	2		Performs I/O (input)		0.7 %
65	2	*	do j=1,nsupm	apath, apath0, supply := 0	
			(1188 ... IF target)		
1175	2		do j=1,nsupm	compute supp(j)	2.4 %
1185	2		do j=1,ndemm	compute demp(j)	2.5 %
2225	2	*	do j=1,nsupm	compute supply(j), apath(j,k)	34.6 %
2240	2		do k=1,ndemm	compute demand(k)	
11175	2		do j=1,nsupm	compute supp(j)	1.9 %
11185	2		do j=1,ndemm	compute demp(j)	2.2 %
12225	2	*	do j=1,nsupm	compute supply(j), apath(j,k)	54.3 %
12240	2		do k=1,ndemm	compute demand(k)	
			(IF ... GO TO 1188)		
245	2	*	do j=1,nsupm	compute itc	

Table 3: Initial classification of loops in **mprojtarg**. Refer to Appendix A for a listing of the original program.

Looking at the last column of Table 3, conventional wisdom would say that the `do 2225` and the `do 12225` loops should be optimized and/or parallelized first, since together they account for about 90% of the run time.

3 *The "Optimize First" Approach*

In one approach, we spent considerable effort on tuning the original code before launching into serious parallelization work. This optimization work was done primarily with the KAP preprocessor, but some hand optimizations were performed. Finally, the modified serial program was analyzed and parallelized using FORGE.

3.1 *The KAP Preprocessor*

The chief tool we used for optimization of the serial code was the KAP preprocessor from Kuck and Associates [10, 14]. KAP was distributed as an integral part of the IBM AIX XL Fortran Compiler/6000 Version 2.3, and was accessed through the use of the compiler option -Pk. Now it must be purchased separately from IBM, beginning with XLF Version 3.1. Based on the results which we describe below, we highly recommend the use of KAP for code optimizations.

What can KAP do for us? There are three general optimization options: (1) `-optimize` applies loop interchanging techniques; (2) `-roundoff` trades certain roundoff errors for increased efficiency; and (3) `-scalaropt` allows a whole range of scalar optimizations. Allowing small roundoff errors enhances the effect of the other options. For example, interchanging loops around arithmetic reductions may cause slight changes in the resulting sums and will not be performed if `-roundoff=0`. Scalar optimizations include loop unrolling, dead code elimination, and memory management. See [6] and [7] for further details.

3.1.1 An Example Transformation in KAP

The following loop will serve as an illustration of the kinds of transformations that KAP can do. The `do 2225` loop from Appendix A is shown here:

```
                 DO 2225 J=1,NSUPM
                 supply(j)=0.
                 DO 2230 K=1,NDEMM
                 TEST=(-SUPP(J)-thxlin(J,K))*(1.+tar(j,k))+DEMP(K)
                 apath(j,k)=apath(j,k)+(rho*(test))
                 if(apath(j,k).lt.0.)apath(j,k)=0.
                 supply(j)=supply(j)+apath(j,k)
2230             CONTINUE
2225             CONTINUE
```

In the code below, generated by KAP, we see several changes: (1) *loop distribution*, separating out the initialization of SUPPLY (see also Figure 4); (2) reordering of the loop nest by interchanging two loops; and (3) the use of floating point registers to allow faster operand references. KAP suboptions used to produce this code were `-optimize=5,-scalaropt=3,-roundoff=2,-unroll=1`, where the latter signifies *no* loop unrolling.

```
do   J=1,NSUPM
 SUPPLY(J) = 0.
 end do
 do   K=1,NDEMM
 _KRR2 = DEMP(K)
 do   J=1,NSUPM
 TEST = _KRR2 + (-(SUPP(J) + THXLIN(J,K))) * (1. + TAR(J,K))
 _KRR15 = APATH(J,K)
```

Dataset	Time	Compiler options
100x100	2:12	`-O2`
100x100	1:04	`-O2 -Pk`
100x100	0:50	`-O2 -Pk -Wp,-optimize=5,-scalaropt=3,-roundoff=3`
100x100	0:52	`-O2 -Pk -Wp,-o=5,-so=3,-r=3,-unroll=1`
100x100	0:58	`-O2 -Pk -Wp,-o=5,-so=3,-r=2,-unroll=1`
500x500	52:29	`-O2`
500x500	26:50	`-O2 -Pk`
500x500	12:34	`-O2 -Pk -Wp,-optimize=5,-scalaropt=3,-roundoff=3`
500x500	13:22	`-O2 -Pk -Wp,-o=5,-so=3,-r=3,-unroll=1`
500x500	14:17	`-O2 -Pk -Wp,-o=5,-so=3,-r=2,-unroll=1`

Table 4: Run times in elapsed min:sec for KAP optimizations in **mprojtarg**. The `-Wp` command line flag gives the listed options to the preprocessor.

```
_KRR16 = _KRR15 + (.0001 * TEST)
if (_KRR16 .gt. 0.) then
_KRR15 = _KRR16
else
_KRR15 = 0.
end if
SUPPLY(J) = SUPPLY(J) + _KRR15
APATH(J,K) = _KRR15
end do
end do
```

3.1.2 Results of KAP Treatment

Use of the KAP preprocessor reduced the run time of the serial code by a factor of two for the 100x100 dataset, and by more than a factor of four for the 500x500 case. The main reason for the speedups is that KAP performed loop nest reordering to achieve stride minimization thus making better use of the cache. As it happens, the loop reordering performed by KAP for serial optimization was the *same* as the loop interchange performed during parallel optimization on some of the loops (but KAP retains the meaning of the indices). Elapsed times in minutes and seconds are tabulated in Table 4, together with the dataset and compiler options used in each case. Note that the option `-Pk` is equivalent to `-Pk -Wp,-optimize=5,-scalaropt=3,-roundoff=0`.

The best performance figures were obtained with full loop unrolling and maximum scalar optimization. However, the generated code was 867 lines of Fortran. Since it was no longer the compact program that we started with, we backed off in the interest of having a manageable (and recognizable) program to analyze with FORGE. We used the code generated with the final options in Table 4 for further work: there is no loop unrolling, nor is there loop blocking. We call this version `noroll.f`; it is 226 lines long, as compared with 164 lines of Fortran for the original code.

3.2 Further Optimizations of the Serial Program

Additional optimizations were undertaken to clean up the code and remove unnecessary statements. The fact that this is a template for actual research codes accounts for some of the extraneous statements and arrays that can profitably be removed.

For example, we noticed that the array APATH was being used only to collect sums for DEMAND and SUPPLY. This could more efficiently be done using a privatized scalar, so we replaced the array with the scalar APTEST. As a result of this change, we were then able to perform some *loop fusions*.[1]

In simplified form, these two loop nests:

```
                DO 2225 J=1,NSUPM
                supply(j)=0.
                DO 2230 K=1,NDEMM
                TEST=(-SUPP(J)-thxlin(J,K))*(1.+tar(j,k))+DEMP(K)
                apath(j,k)=apath(j,k)+(rho*(test))
                if(apath(j,k).lt.0.)apath(j,k)=0.
                supply(j)=supply(j)+apath(j,k)
2230            CONTINUE
2225            CONTINUE

                do 2240 k=1,ndemm
                demand(k)=0.
                do 2245 j=1,nsupm
                demand(k)=demand(k)+apath(j,k)
2245            continue
2240            continue
```

became one loop nest:

```
                DO 2230 K=1,NDEMM
                demand(k)=0.
                DO 2230 J=1,NSUPM
                TEST=(-SUPP(J)-thxlin(J,K))*(1.+tar(j,k))+DEMP(K)
                aptest=apatho(j,k)+rho*test
```

[1] Loop fusion is the opposite of loop distribution, and simply joins two loops with the same bounds. It is illustrated in Figure 4 on page 16 except now the original code is on the right, and the transformed code is on the left.

Program	Dataset	Time	Compiler options
noroll.f	100x100	0:58	`-O2`
optim.f	100x100	0:55	`-O2`
optim.f	100x100	0:49	`-O2 -Pk -Wp,-o=5,-so=3,-r=3`
noroll.f	500x500	14:17	`-O2`
optim.f	500x500	13:26	`-O2`
optim.f	500x500	11:40	`-O2 -Pk -Wp,-o=5,-so=3,-r=3`

Table 5: Comparison of noroll.f and optim.f

```
                if(aptest.lt.0.)aptest=0.
                supply(j)=supply(j)+aptest
                demand(k)=demand(k)+aptest
2230            CONTINUE
```

This set of serial optimizations led to an improved code `optim.f`. Performance figures for `noroll.f` and for `optim.f` are tabulated in Table 5. Thus concludes our serial optimization effort, with the best elapsed time of 11 minutes 40 seconds for the 500x500 dataset, a substantial reduction over the initial time of 52 minutes 29 seconds.

3.3 Parallelization using FORGE

The analysis of `optim.f` using FORGE and optimization of the resulting parallel program was similar to the procedure followed in Section 4. These steps produced `final.f`, a parallel version of `mprojtarg`. Elapsed run times for the final code are shown in Table 6.

4 The "Parallelize First" Approach

The other approach to parallelization is to parallelize the serial program directly, and then improve upon the resulting parallel program. It is generally the case that in order to parallelize a code, one must first reorganize the serial program to expose parallelism and remove data dependences, or at least insert directives to a parallelizing compiler. We chose to use data access normalization, a new parallel optimization method for distributed memory machines [11].

	Run times on the SP1 in elapsed min:sec				
	serial	1-way	2-way	3-way	4-way
Dataset 1	0:52	1:19	1:29	2:27	2:43
Dataset 2	2:10	2:32	2:06	3:18	3:32
Dataset 3	4:35	6:28	3:58	4:18	4:59
Dataset 4	12:02	13:01	9:07	7:46	12:36
Dataset 5	12:46	13:40	8:13	7:07	7:14

Table 6: Run times on the SP1 for `final.f`, the result of parallelizing `optim.f` using FORGE DMP. These runs were performed using MPL rather than MPLp, the lower latency protocol for the SP1 switch. A subsequent 4-way MPLp run on Dataset 5 took only 5:02 minutes.

4.1 Data Access Normalization

This method, developed at Cornell as one application of the Lambda Toolkit [12], has been incorporated into the ParaScope programming environment [2] as an enhancement to ParaScope's program editor, PEDX. The enhanced editor is called PEDLAMBDA [3, 4].

The basic idea of data access normalization is that given some decomposition[2] of problem data across processor memories, one transforms loop nests so that access to data is made as local as possible. Transforming a loop nest requires not only rewriting loop bounds, but also all expressions within the loop nest that refer to loop index variables. Where expressions are subscripts, the goal is to have them correspond to the parallel, non-dependence-carrying, outermost loop. Among all such expressions, the ones that occur most frequently in partitioned dimensions are favored; that is, after rewriting the loop nest, the outermost loop index and very frequent, distributed, subscript expressions will be identical.

Figure 2 illustrates one possible decomposition of the program variables listed in Table 1. Shaded areas represent data that are basically read-only. The large two-dimensional arrays (APATHO, APATH[3]) keep track of the emerging interaction between suppliers and consumers. We decided to decompose these large arrays by column, thus aligning them with "demand." Only the processor in which the columns are stored would assign values to them. (If supply exceeded demand, we

[2] perhaps expressed as HPF directives

[3] APATH turns out to be an unnecessary array, as noted in Section 3.2 on page 11.

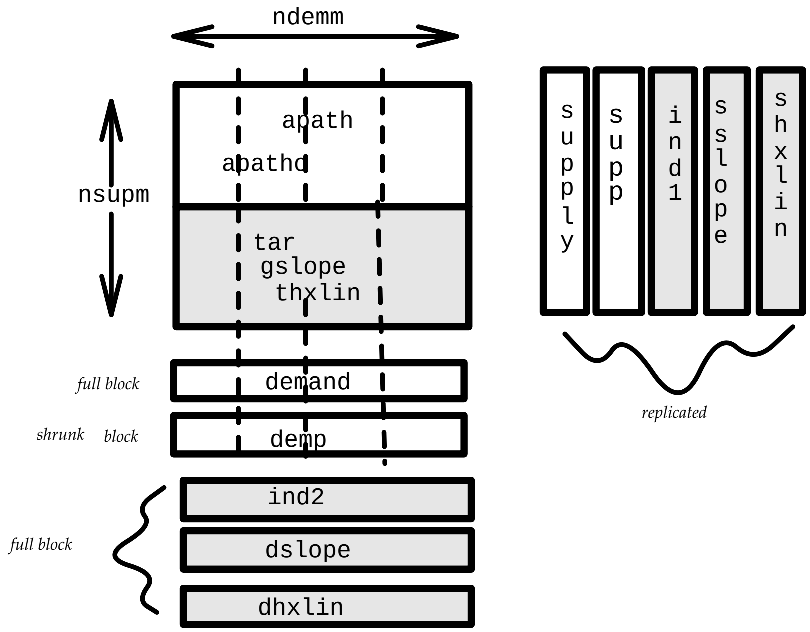

Figure 2: This is one possible partitioning of **mprojtarg** data. "Full block" means that storage is set aside in each processor for the entire array, even though it is assumed the processor assigns to only that part of the array it owns. "Shrunk block" sets aside only as much memory as needed to hold the part of an array that is owned by the processor. "Replicated" means that the array is replicated in full across all processors. The shaded areas are read-only arrays, input from disk files. Here, we decompose the "demand" variables and replicate the "supply" variables. The dashed lines indicate typical processor boundaries.

---

would probably have partitioned problem data the other way — by rows — in order to maximize the amount of work that could be done in parallel).

Given that APATH and APATHO are partitioned by columns, it is prudent to leave SUPPLY and friends replicated in each processor so that the computation of each column of APATH would have immediately available all data needed, including the current price data in SUPP. The computation of SUPPLY is an efficient sum-reduction across APATH rows. DEMAND is full block rather than shrunk only because the computation of DEMP in each processor indirectly accesses DEMAND. (Recall that the demand price is an irregular function of the demand volume.) Input-only arrays are fully blocked (not shrunk) so that we could read data into them in parallel[4] if we had to. Our goal is to find a parallelization that works best for this data decomposition, using the data access normalization technique embodied in PEDLAMBDA. Section 6 comments on some other possible decomposition strategies.

[4] "read data into them in parallel" means that each instance of an SPMD version of this application could potentially read *all* the data into its own memory.

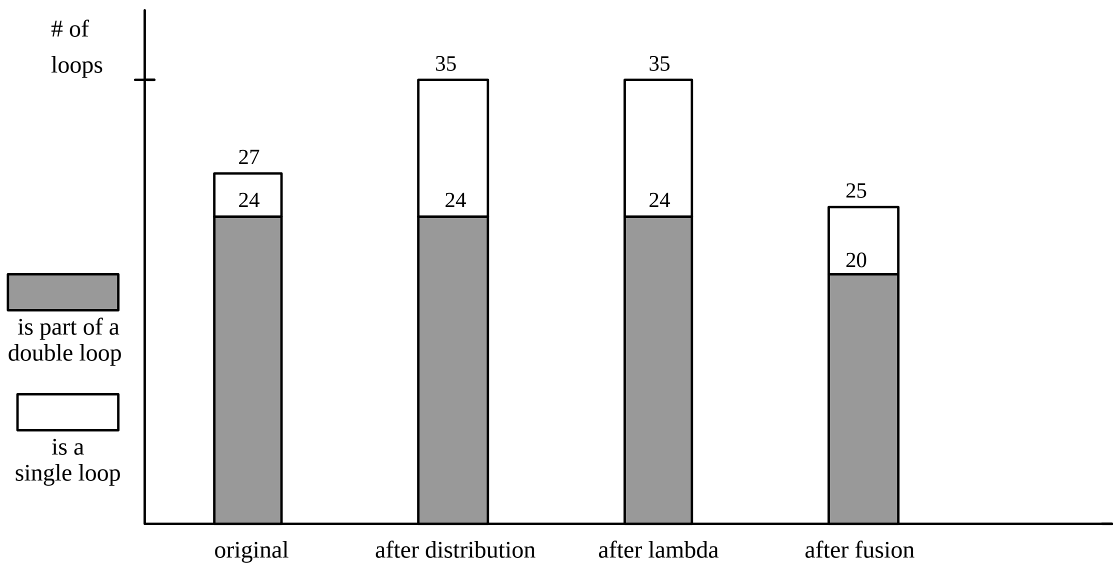

Figure 3: Diagram of **mprojtarg** loop population during the course of several loop transformation steps.

The goal is to transform as many loop nests as possible. As shown in the leftmost columns of Figure 3, there are 12 double loop nests in the program. However, two of the 12 double loops contain I/O and can not profitably be rearranged. That leaves the 10 loops in the bottom portion of Table 3, but none of them can be transformed, because they are not perfect nests and PEDLAMBDA currently works only on tight loop nests.

4.1.1 Loop Distribution – Preparing for Data Access Normalization

In Section 3, KAP performed loop distribution in order to interchange the remaining loops. Here we perform loop distribution in order to apply Lambda-based transformations. In ParaScope, one interactively selects the first statement of a loop nest and then picks the loop distribution menu item. The code is changed as shown in Figure 4.

Loop distribution yields a tight loop nest at the expense of one additional loop, and so we are left with 35 loops (second column in Figure 3). Of the twelve double loops, four are amenable to Lambda transformations: 65, 2225, 12225 and 245. These loops have asterisks in Table 3. The thing to notice about these four loop nests is that their outer loop index iterates over NSUPM whereas the data are decomposed across the NDEMM index. To normalize data access - i.e. minimize communication

```
                                        do j = 1, nsupm
                                        supply(j) = 0.
                                        end do

      do 2225 j = 1, nsupm
      supply(j) = 0.                    do 2225 j = 1, nsupm
      do 2230 k = 1, ndemm              do 2230 k = 1, ndemm
          :                                 :
      apath(j,k) = ...                  apath(j,k) = ...
          :                                 :
2230   continue                  2230   continue
2225   continue                  2225   continue

         (a)                             (b)
```

Figure 4: Loop Distribution replaces one loop nest with several by pulling statements contained in one loop but not the other out into a separate loop. Here, (a) is the original loop nest. By moving `supply(j)=0` into its own loop (b), we have "distributed" the loop. Now the `do 12225` loop is a tight loop nest, suitable for further analysis and transformation.

- PEDLAMBDA interchanges these four nests so that the NDEMM index is in the outermost loop. As we shall see later, this interchange greatly reduces parallel run time. In general, the Lambda transformation is more complicated than that necessary for **mprojtarg**; it performs, in one step, whatever interchange, scaling, reversal, and skewing is necessary to obtain normalized data access.

4.1.2 An Example Transformation in PEDLAMBDA

The loop shown on page 9 and outlined in Figure 4a is one of the more time-consuming ones. It computes APATH and also updates the SUPPLY vector. Since APATH is partitioned by columns with a block of columns allocated to each processor, after breaking out the `SUPPLY(J)=0.`, PEDLAMBDA's data access normalization interchanges the two loops, putting the second index of APATH in the outer loop. With a touch of the PEDLAMBDA button, ParaScope correctly interchanges the loops and all the subscripts (see Figure 5). (Notice also that ParaScope gratuitously reformats the code, deletes statement numbers – which were re-inserted by hand, and switches the axes over which each index variable iterates.)

```
      do 2225 j = 1, ndemm
        do 2230 k = 1, nsupm
          test = (-supp(k) - thxlin(k,j)) * (1. + tar(k,j)) + demp(j)
          apath(k, j) = apath(k, j) + (rho * (test))
          if (apath(k, j) .lt. 0.) apath(k, j) = 0.
          supply(k) = supply(k) + apath(k, j)
2230    continue
2225  continue
```

Figure 5: A loop that has been data access normalized. With the interchanged loop, columns of the decomposed data, if they need to be communicated, can be communicated in blocks consisting of columns. If they are only stored locally, they are more likely to be held in cache during execution of the inner loop.

4.1.3 Loop Fusion – Cleaning Up After Data Access Normalization

After the PEDLAMBDA transformations have been applied to every loop nest, it is possible that some opportunities for loop fusion arise, due to the loop distribution plus the interchanging of some double loops. For example, the new `do 2225` loop of Figure 5 was originally followed by this loop nest:

```
              do 2240 k=1,ndemm
              demand(k)=0.
              do 2245 j=1,nsupm
              demand(k)=demand(k)+apath(j,k)
2245          continue
2240          continue
```

Since the transformed `do 2225` loop runs from 1 to NDEMM, it can be fused with the `do 2240` loop. After applying ParaScope's loop fusion button, we wind up with the loop shown in Figure 6. ParaScope generates new names for loop index variables, drops statement numbers, rewrites subscript expressions, and replaces `continue` with `enddo`.

Note that the fusion of parallelized code is quite different from that done for the serial optimization in Section 3.2.

At this point the reader may be wondering why we are working on *all* the loop nests in the program rather than just the two "hot spots" pointed out by our timing run (see Table 3). The

```
         do j$1 = 1, ndemm
           do 2230 k = 1, nsupm
             test = (-supp(k) - thxlin(k,j$1)) * (1. + tar(k,j$1)) + demp(j$1)
             apath(k, j$1) = apath(k, j$1) + (rho * (test))
             if (apath(k, j$1) .lt. 0.) apath(k, j$1) = 0.
             supply(k) = supply(k) + apath(k, j$1)
 2230      continue
           demand(j$1) = 0.
           do 2245 j = 1, nsupm
             demand(j$1) = demand(j$1) + apath(j, j$1)
 2245      continue
 2240      continue
         enddo
```

Figure 6: Two loops, `do 2225` and `do 2240`, fused into a single loop by ParaScope.

main reason for this is that to minimize communication of global data, you must be consistent about which processor handles which data, which typically means parallelizing even small loops.

For example, consider the `do 245` loop in Figure 3 which computes ITC by counting how many elements of APATH are non-zero. If this loop were replicated rather than parallelized, each node would be forced to communicate its particular columns of APATH to every other node, just so that each could then compute ITC. If instead we parallelize that seemingly inconsequential loop, ITC can be computed by an efficient sum reduction.

4.2 FORGE 90 DMP and xHPF

The previous section outlined one strategy for preparing programs for distributed memory parallelization: (1) distribute the loop nests; (2) apply the data access normalization Lambda transformation; (3) fuse loops. Using the available tools, you are told which individual edits are legal.

The final step toward parallelization is to feed the reorganized code to a parallelizer, such as the FORGE Distributed Memory Parallelizer (DMP) [17] or High Performance Fortran preprocessor (xHPF) [16]. DMP is an interactive tool, whereas xHPF is a batch tool. In either case, the output is a SPMD parallel program with message-passing. This program can be compiled and linked with MPL or MPLp (IBM's two message passing libraries for the SP1 [8]) or with other message passing

	Run times on the SP1 in elapsed min:sec				
	serial	1-way	2-way	3-way	4-way
Dataset 1	1:02	1:32	14:42	24:26	40:35
Dataset 5	15:25	19:16	185:44	(incomplete)	

Table 7: After data access normalization but before any other optimizations. Elapsed time (`min:sec`) was determined for each run by `date;`command`;date` in a standalone batch queue. Serial time is the same program with no FORGE calls. 1-way is with only one processor, so there are no communications, but there is some parallel overhead. Data access normalization has reduced the runtimes for dataset 1 from many, many hours to minutes, but severe "slowdown" problems are evident. We did not pursue runs with the larger datasets, as they used too much machine time.

libraries such as PVM (public domain software from Oak Ridge National Laboratory [18]). The result can be run on the SP1.

It is also possible to use DMP and xHPF in concert with each other. If you insert HPF directives into the Fortran program, xHPF will parallelize it, and optionally produce a database that can be analyzed by DMP. For example, you might determine from interactive inspection of the program that some communications are unnecessary. (FORGE is conservative and will generate communications that might be needed to keep data consistent). You can also examine variable uses and find variables that are not referenced.

During the PEDLAMBDA step, we used FORGE in one other way: to check our parallelization. After doing a few transformations, we would run the new program through FORGE and then on the SP1 just to make sure we were still getting the same answers as the serial program.

4.3 Optimization

At this point we have a parallel program that has been data access normalized, and which obtains correct results when run in parallel. However, the program takes far too long when run in parallel. Table 7 shows how badly the program slows down as extra processors are added. This section describes how we optimized the parallel program produced by FORGE.

4.3.1 *Optimizing Communications*

Using the VT trace visualizer for the SP1, we found unnecessary communications before and after each loop. FORGE xHPF parallelizes too many of the loops, and apparently does not do a deep enough analysis to recognize when data computed by one processor are not needed by other processors later in the program. Given our decomposition, FORGE generated communications such as the following:

```
---> Distribute the loop on DEMP<1~1>
---> Preloop  communication of  IND2<1~1, 1:ncr>
---> Preloop  communication of  DEMAND<@>
---> Preloop  communication of  DHXLIN<1~1>
---> Preloop  communication of  DSLOPE<1~1, 1:ncr>
---> Postloop communication of  DEMP<1~1>
                   DO 1185 J=1,NDEMM
                   DEMP(J)=0.
                   DO 1187 K=1,NCR
                   DEMP(J)=DEMP(J)+DSLOPE(J,K)*(DEMAND(IND2(J,K)))
      1187        CONTINUE
                     DEMP(J)=DEMP(J)+DHXLIN(J)
      1185         CONTINUE
```

FORGE annotation is documented in [16]. Here the annotation says that prior to computing this processor's elements of DEMP, the processor must import its portion of IND2, DEMAND, DHXLIN, DSLOPE, and afterwards export DEMP. (The portion of an array "belonging" to this processor is denoted by `1~1`.) However, these data are already in local memory (refer to Figure 2): IND2, DHXLIN, and DSLOPE are read-only arrays; DEMAND is completely up-to-date and identical in each processor because of an inter-processor exchange of DEMAND values slightly earlier in the run. It is possible that the indirect addressing of DEMAND obscured the true simplicity of the dependence situation.

Fortunately, we can use FORGE DMP interactively to remove all four preloop communications and the postloop communication from this loop nest. The effect of deleting useless communications (called *ignoring* in the parlance of FORGE DMP) can be seen in the following, where two partial runs are compared, first with extraneous communications present and then without. (The output of the program is the same in either case.) This experiment indicated that FORGE is not able by itself to optimize away all unnecessary communications, either at compile time or at run time.

	Run times on the SP1 in elapsed min:sec				
	serial	1-way	2-way	3-way	4-way
Dataset 1	1:04	1:19	1:58	3:31	4:33
Dataset 2	2:41	3:07	5:03	5:32	6:52
Dataset 3	5:38	6:20	13:04	20:20	19:38
Dataset 4	17:54	15:56	31:38	30:31	30:24
Dataset 5	15:25	16:57	15:15	18:23	20:34

Table 8: After data access normalization and ignoring unnecessary data communications. Elapsed time (`min:sec`) was determined for each run by `date`;command;`date` in a standalone batch queue. Serial time is the same program without any calls to FORGE runtime routines, so there is no parallel overhead. 1-way is with only one processor, so there are no communications, but there is some parallel overhead. See Appendix B for handling of communications.

```
500x500 dataset      100 iterations        Same, but with
 n (for n-way)     Elapsed Time (min)   Many Comms Ignored
      1                  :30                    :21
      2                 2:13                   1:03
      3                 3:37                   1:58
      4                 4:46                   1:51
```

The savings in elapsed wall clock time by removing unneeded communications can be proportional to the number of processors. Appendix B contains an "annotated listing" of the final version of the parallelized program, showing which communications we ignored. The run times of the normalized minimal-communication program are shown in Table 8. While the times are much improved over Table 7, there is still no speedup.

4.3.2 Parallel I/O

When FORGE parallelizes a program, it produces an SPMD program that performs I/O only on node 0 (the processor from which the program is launched). This is a properly conservative approach, but it is very expensive in communications since node 0 must then broadcast the input data to all other nodes.

FORGE contains a parallel profiler, called **polytime**, which breaks down the time spent in each loop into communication costs and computational costs. A run through this profiler confirms that a significant cost in a parallel run is the reading in of initial input and its distribution to the other

Run times in elapsed min:sec for major stages in parallelization, based on 4-way runs.

Dataset	Time	Stage of Parallelization	Source
100x100	2:12	Original program (`-O2`)	Table 4
100x100	4:33	After data access normalization	Table 8, 4-way
100x100	2:37	After parallel I/O	Batch run 5/26/94
100x100	2:30	After other optimizations	Batch run 6/3/94
100x100	2:28	MPLp	Lightly loaded interactive 5/10/94
500x500	52:29	Original program (`-O2`)	Table 4
500x500	20:34	After data access normalization	Table 8, 4-way
500x500	9:05	After parallel I/O	Batch run 4/16/94
500x500	7:37	After other optimizations	Batch run 4/28
500x500	6:51	MPLp	Lightly loaded interactive 5/10/94

Table 9: This table shows the effect of each successive step of "parallelize first, then optimize". The first line in each case is simply the original program run serially. The first parallel run is based on data access normalization and minimized inter-processor communications, but with all data being read by node 0 and then distributed to the other processors. The middle line in each case removes this serial bottleneck by having each processor read in all the data directly. These results show a major improvement due to parallel I/O. The last line in each case shows what happens when the same program is run over the switch using MPLp, the optimized communications library. (Elapsed time (min:sec) for each run is determined by the date command, e.g. `date;mproj;date`.)

processors.[5] The more processors there are, the more time this communication will take.

Fortunately, a simple solution is at hand. It is possible for all node programs simultaneously to read the same input file into their own local memory, with no need for further broadcasting of data. Arranging for parallel input proved to be simple:

1. Declare all input arrays to be FULLBLOCK rather than SHRUNKBLOCK so that there is sufficient space allocated in each processor's memory to hold all the data to be read in.

2. Move all the input statements into one input routine.

3. Put all the data areas into named COMMON blocks so that they are available both to the main program and the input routine.

4. Call this routine from the main program. Upon return, the program instance running in each processor will have all the needed data on hand (and probably more than what's needed, since it is easiest to have each node read *all* the data rather than only the part it "owns".

5. When parallelizing this program with FORGE, declare the input routine to be "benign", by using the BenignUnknown flag on the **xhpf** command or interactively if using DMP. (This is, of course, a bald-faced lie since the input routine is nothing *but* side effect!)

6. Generate the _pf.f or _mpf.f program and compile[6] and link it with the input routine. Example:

```
pref77 -p mproj     <===(creates an mproj_pf.f)
xlf -c -O2  getdata.f   <=== subroutine for reading data
mpxlf -o mproj mproj_pf.f getdata.o -ip -O2 \
   -L$APRMPLLIB -ldd_n
```

The results of optimizing file I/O are shown in Table 9.

4.3.3 Other Optimizations

In working with FORGE to bring run times into the realm of reason, we were shown that many small optimizations of the program were possible. For example, the variables declared but never

[5]This is in stark contrast to the profile of the serial program which showed that the input loops consumed relatively little of the elapsed job time (Table 3). The serial and parallel programs have much the same I/O time, but the parallel program in addition has communication time.)

[6]You have to use the -ip switch rather than -lsp due to a current bug in FORGE runtime library for MPL.

used were marked as such in FORGE's screen displays. We removed these from the program. XHPF's parallelization report alerted us to the useless assignment to TEST in the final loop of the program. We commented this out. GSLOPE is read in but never used, so we replaced this by a scalar. It should be noted that xHPF, like `xlf`, uses the KAP preprocessor to analyze the input program and to translate F90 constructs into Fortran 77.

All of these changes were easily made to the message passing program output by FORGE. For convenience, however, we copied these changes into the input program as well.

We also studied compiler options a bit, just to determine what optimization level to use. The FORGE-generated message passing program is very long with complicated indexing expressions. We found that `-O2` produces slightly shorter run times than `-O1` (which cannot promote the subscript computations enough) and `-O3` (for whatever reason). None of these optimizations have much effect on the runtime, however.

The final optimization was to run the program over the lower-latency protocol for the SP1 switch, supported by the MPLp runtime library. To link our program with MPLp,

```
xlf -o mproj -bimport:/usr/lpp/euih/eui/eui.exp \
   mproj_pf.f getdata.o -O2 -L$APRMPLPLIB -ldd_n
```

This last improvement resulted in speedup for the program on 8 processors.[7] The lower latency of MPLp was enough to overcome the overhead of exchanging several messages during each iteration.

5 *Comments on Optimization and Parallelization*

There are two different paths to an optimal parallel code, and two points of view. One camp says to get the fastest possible serial program and then parallelize that one; the other camp says that one should go parallel right away.

For legitimate speedup figures, it is necessary to find the optimal serial program anyway, so one might as well go on and parallelize that. On the other hand, it is by no means intuitively obvious that starting with the best serial program is the way to wind up with the best parallel program.

For example, loop unrolling is extremely effective on super-scalar architectures such as the RS/6K but can cause problems when parallelizing codes because it only makes data dependence

[7] See [4] for details; 4-way on Dataset 5 was 6:51 minutes and 8-way was 4:23 minutes.

analysis more difficult. By the same token, we could have run KAP *after* parallelization on the message-passing program to unroll loops, but it probably would not have been effective. The parallel code has too many calls to the FORGE runtime library.

Likewise, splitting out the input into a separate routine was a crucial optimization of the parallel program, but would never be considered when optimizing a serial program. A final example is the scalarization of APATH. The fact that this is a temporary and need not be replicated is a nice serial optimization but (depending on the parallel architecture) may need to be replicated for parallel implementations.

Thus in our work we tried both approaches. We noticed in the course of this work that with the "optimize first" strategy you tend to work on the hot spot loops only. With the "parallelize first" approach, you work with *all* the loops to guard against globalizing locally-held data.

It can be difficult to categorize program transformations. For example, one might wonder whether data access normalization is an optimization or a parallelization. Both strategies discussed in the paper are, after all, applied to serial codes, not parallel codes. The difference is what do you have in mind: maximal serial speed? or maximal concurrency (which is often the same as maximal computation to communication ratio)? If the former, you will use optimizations such as cache reuse techniques and loop unrolling. If the latter, then you are using a parallelization technique; data access normalization is of that class.

6 *Suitability of the Application for this Study*

This particular application is very interesting for several reasons. First, it is only four pages long, yet computationally intense. The large loop (if...goto 1188) iterates over time steps and therefore cannot be parallelized. But the two inner loops that account for 90% of each time step *can* be parallelized.

Indirect indexing is used on the SUPPLY and DEMAND vectors, and so these vectors need to be updated to have the same values everywhere whenever they are re-computed. However, the index pattern is fixed, so the problem could (perhaps) be solved by pre-sorting the data, or it could be amenable to runtime optimization of irregular communications (see Saltz [5]). No dynamic realigning of data structures is necessary.

There is scope here for parallel I/O, since IND1, IND2, SSLOPE, DSLOPE, THXLIN, DXLIN, and TAR can all be read into each processor's memory in parallel. It would be better to be able to read only partial information into partitioned shrunk arrays, but this is beyond the capability of our existing tools.

There are interesting tradeoffs to be made between replicating computation versus communicating between processors. For example, if we partitioned SUPPLY and friends (in addition to DEMAND and friends), then SUPP could be computed in parallel but it would then need to be communicated to all the other processors. We decided that given the minimal computation involved, it would be faster for each processor to compute all of SUPP by sum reduction across the column-partitioned APATH0 arrays. (xHPF, given a free hand, decides to compute SUPP in parallel. It parallelizes every loop it can. There is an APR directive to prevent loop parallelization.)

To make this program run really fast, however, the total number of iterations will need to be reduced. This would require an algorithmic redesign, and might affect the convergence properties of the code.

The most interesting finding was that the serial optimizer and the parallel optimizer reached the same conclusion: interchange four of the loop nests. But the conclusion was reached for different reasons. The KAP preprocessor did it to maximize cache hits, while data access normalization did it to regularize data communications in parallel loops. Given a different data decomposition, the two approaches would have been in conflict. KAP, being a serial code optimizer for RS/6000 workstations, would of course not take data partitioning into account.

7 *What We Learned About Tools By Doing This Project*

This section contains a number of observations and strategies for tool-based program parallelization.

7.1 *General*

1. The quickest way to get an overview of a program's loop structure is to run `pat -l <program>`. This produces a very useful list of loop nests, their depth, their location in the source, and an indication of whether or not they are trivially parallelizable.

2. We enjoyed the code manipulation abilities of both ParaScope and FORGE. The original **mprojtarg** code was read and processed by all our tools with no problem, indicating that the sample code uses standard Fortran 77 syntax. However, as code migrates to Fortran 90 over the next couple of years, older tools such as ParaScope will probably no longer work.

3. It would have been very handy to have a code mover that would let you change `nesta;nestb;nestc` into `nesta;nestc;nestb`. You might, for example, want to do this to set things up for a loop fusion step. It turns out that interchanging loop nests can be just as useful as interchanging loops within a loop nest.

4. Do bulk editing with your favorite editor, not with ParaScope's (which is slow) or with FORGE's (which is quirky). Both ParaScope and FORGE can be run in the background, leaving the foreground for file editing.

5. When using ParaScope, reinsert the `do` statement numbers that ParaScope deletes, so that it is possible for you to analyze the same program within FORGE. This is a prime example of two tools not quite working together.

7.2 About ParaScope and KAP

1. Since ParaScope is experimental software, we were paranoid enough to check the output of each step (distribution, pedLambda, fusion) by running the resulting code on the SP1. ParaScope never made an incorrect transformation, though at one point it lost the crucial 1188 statement number, and occasionally dumped core. The lesson: you can trust ParaScope's results, but save often!

2. ParaScope is perfectly willing to fuse loops which contain I/O statements. This can certainly mess up your relationship with your data. On the other hand, this is a more flexible approach than KAP's, in which all loop nests containing I/O are ineligible for further analysis.

3. It is easier to fuse loops by hand except when many indices and subscripts need to be changed. Then ParaScope's name manipulation is very handy.

4. The minor inconvenience of having to edit the new index names (e.g. `j$1`) back into something reasonable and to reinsert lost statement numbers is outweighed by the major advantages of ParaScope – with mouse clicks you can distribute, rerarrange, and fuse loops. The main benefit of KAP is that you don't need to use the mouse at all, nor decide in what order to make the transformations. KAP also retains the meaning of the indices.

5. A big benefit of embedding the Lambda Toolkit into a larger programming environment like ParaScope is that operations such as loop distribution and fusion can be done alongside Lambda transformations.

6. Our experience is that ParaScope, though conservative in its assumptions, can make some rather extensive changes to your code on your behalf. At each step of the way, the program is checked for accuracy, protecting you from clerical errors.

7.3 About FORGE

1. The annotated listing from FORGE DMP is very useful. It shows where in the program it is likely that interprocessor communications will occur. These can be correlated with message passing that shows up in trace visualizations. This is the sort of thing we will expect with HPF compilers from third-party vendors. A FORGE annotated listing is gotten by:

```
          select a package
          analyze
          select the entire code tree
          parallelize for distributed memory
          show parallel code
     (point cursor at the parallel code window)
          spacebar
          save filename
```

2. It really is necessary to examine the annotated listing for unnecessary communications.

3. Many programs, like this one, have read-only data that is read in at the beginning of the run. The parallel I/O solution given in this paper is general and effective.

4. DMP and xHPF complement each other very well.

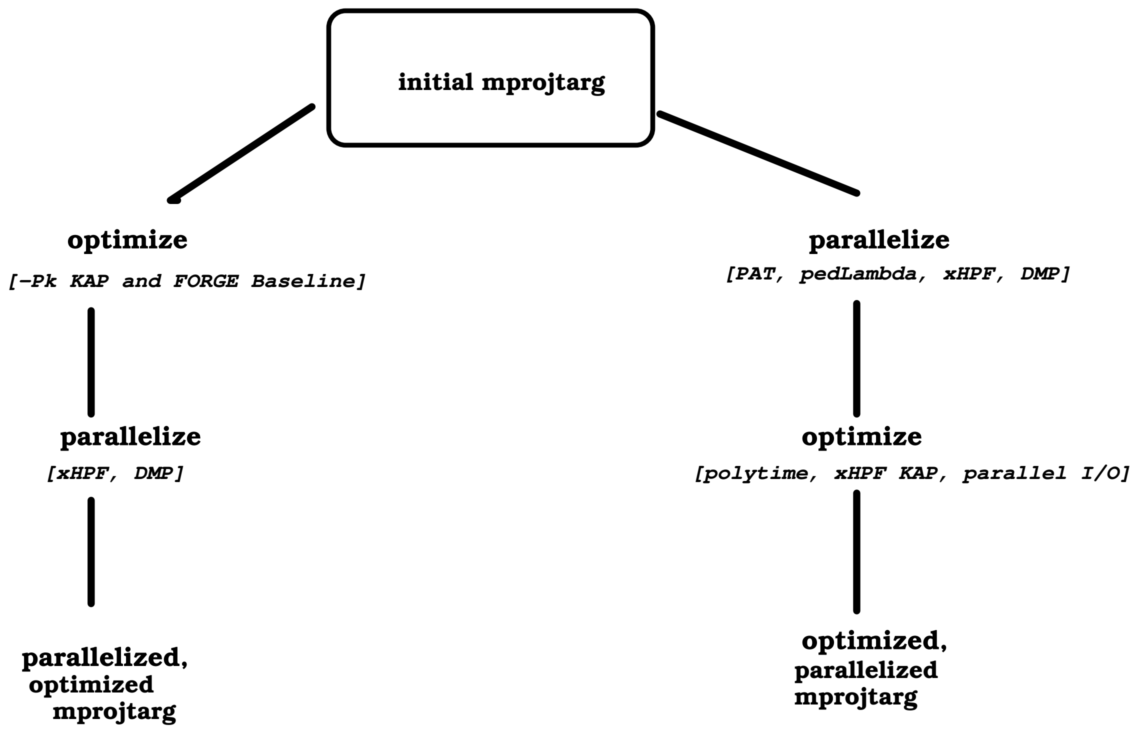

Figure 7: The two strategies compared, along with typical tools for moving from one stage to the next.

5. FORGE is robust. We encountered 3 bugs in ParaScope but only 1 in FORGE's pref77, which was quickly fixed by APR, the vendor.

8 Conclusions

In practice, we mix and match optimization and parallelization strategies. After all, in this study the "optimize first" plan did apply further optimization (like parallel I/O) to the parallel code, and some serial optimizations (like loop unrolling) were backed out. We considered the extremes here in order to contrast the two approaches.

An interesting finding was that the two methods rely on somewhat different sets of tools (see Figure 7). Also, if you start with parallelization, you may never bother with some detailed optimizations to the node program, having gotten more effective speedup from parallelization.

In either case, this study gave us a chance to evaluate some parallel programming tools and also experiment with the data access normalization technique.

We conclude that data access normalization works. Parallelizing an application for the SP1 using parallel tools works. We confirmed the validity of the general overall approach to parallelization consisting of these steps:

1. Reduce your problem to a partial run, just long enough to test the validity of each transformation as you do it. In this case, we simply ran 3 iterations.
2. Run the serial code and save the answers for reference purposes while doing these validations (the equivalent of Table 2).
3. "Prep" the code by removing or protecting data dependences. For this, PAT and ParaScope are best. Keep in mind that reductions are usually handled automatically by parallelizers, including FORGE. You do not have to remove or protect the data dependences implied by reductions.
4. Use DMP and xHPF to get a parallel code.
5. Run it on 1, 2, and n processors to see if the answer is correct. You can do this on an interactive workstation cluster using PVM or on the SP1 itself.
6. When the answers are correct, go back to the full code, link it with the SP1's communications library (currently MPL) and run the code in a single-threaded parallel batch queue.
7. If there is no speedup, find where the communications are.
   (a) Turn on tracing and see where communications happen.
   (b) Inspect the annotated listing for potential communications "hotspots."
   (c) Use FORGE's variable static tracing features to determine which communications might safely be ignored. Turn these off.
   (d) Go to parallel I/O. Avoid the "node zero bottleneck."

9 *Implementation Notes*

All work was done in an AFS cluster of RS/6K workstations running AIX 3.2.2 and 3.2.5 and on an IBM 9076 Scalable POWERparallel System running AIX 3.2.4.

We did all our work in Fortran, using the April 1993 Release of ParaScope, Version 8.5 and 8.9 of FORGE, the 1994 version of PEDLAMBDA, and xlf 2.4 with KAP. Other tools used were PAT (1992) and VT (version 1).

Timings were mostly `date;"run-job";date` in singly-threaded LoadLeveler batch queues on the SP1, using `ip` over the High Performance Switch. CPU times, where reported, were obtained from `mclock()` which has a resolution of .01 second for system and user time of the caller.

10 Acknowledgements

This research was conducted using the resources of the Cornell Theory Center, which receives major funding from the National Science Foundation, and New York State. Additional funding comes from the Advanced Research Projects Agency, the National Institutes of Health, IBM Corporation and other members of the center's Corporate Research Institute.

Prof. Anna Nagurney (and Arundhati Dhagat, graduate student) are to be thanked for providing us with a most interesting application. We thank Professors Wei Li and Keshav Pingali for the Lambda Toolkit. The CRPC at Rice University provided ParaScope software and consulting. Georgia Institute of Technology provided PAT. FORGE is a commercial product from Applied Parallel Research.

David Presberg reviewed this report and made many helpful comments on its contents.

References

[1] Bill Appelbe, Kevin Smith, and Charlie McDowell. Start/pat: A parallel-programming toolkit. *IEEE Software*, pages 29–38, July 1987.

[2] V. Balasundaram, K. Kennedy, U. Kremer, K. S. McKinley, and J. Subhlok. The ParaScope Editor: An interactive parallel programming tool. In *Proceedings of Supercomputing '89*, November 1989.

[3] D. Bergmark and D. Presberg. Initial experiments in the integration of ParaScope and Lambda. Technical Report CTC93TR136, Cornell Theory Center, June 1993.

[4] D. Bergmark and D. Presberg. The integration of ParaScope and Lambda. Technical Report CTC94TR180, Cornell Theory Center, June 1994.

[5] R. Das, R. Ponnusamy, J. Saltz, and D. Mavriplis. *Distributed Memory Compiler Methods for Irregular Problems – Data Copy Reuse and Runtime Partitioning*, volume 3 of *Advances in Parallel Computing*, pages 185–218. Elsevier Science Publishers, 1992.

[6] IBM. *AIX XL FORTRAN compiler/6000 User's Guide, Version 2.3*, 1992.

[7] IBM. *Optimization and Tuning Guide for the XL FORTRAN and XL C Compilers*, 1992.

[8] IBM. *AIX Parallel Environment Parallel Programming Reference*, September 1993.

[9] IBM. *IBM Scalable POWERparallel Systems Reference Guide*, 1993.

[10] Kuck & Associates, Inc., Champaign, Ill. *KAP User's Guide*, 1988.

[11] W. Li and K. Pingali. Access normalization: Loop restructuring for NUMA compilers. *ASPLOS '92*, 1992.

[12] W. Li and K. Pingali. A singular loop transformation framework based on non-singular matrices. In *Proc. 5th Annual Workshop on Languages and Compilers for Parallelism*, August 1992. Also published as Cornell Theory Center Tech Report CTC92TR98, July 1992.

[13] A. Nagurney, C. Nicholson, and P. Bishop. Spatial price equilibrium models with discriminatory *ad valorem* tariffs: Formulation and comparative computation using variational inequalities. An invited paper for a Festchrift volume in honor of Takayama to be published in Holland, December 1993.

[14] P. Petersen and D. Padua. *Machine-Independent Evaluation of Parallelizing Compilers.* January 1992. Also available as CSRD Report No. 1173.

[15] Applied Parallel Research. *FORGE 90 Baseline System User's Guide.* 550 Main Street, Suite I, Placerville, CA 95667, June 1993.

[16] Applied Parallel Research. *FORGE High Performance Fortran xHPF User's Guide.* 550 Main Street, Suite I, Placerville, CA 95667, December 1993.

[17] Applied Parallel Research. *FORGE 90 Distributed Memory Parallelizer User's Guide.* 550 Main Street, Suite I, Placerville, CA 95667, April 1994.

[18] V. S. Sunderam. PVM: A framework for parallel distributed computing. *Concurrency: Practice & Experience*, pages 315–339, December 1990.

[19] J. Torné. How parallel programming tools are used. Technical Report CTC94TR166, Cornell Theory Center, February 1994.

Appendix A: The Original MPROJTARG Program

```
      program mprojtarg
C       THIS PROGRAM SOLVES THE LINEAR ASYMMETRIC SPATIAL PRICE EQUILIBRIUM
C       PROBLEM with tariffs using the modified projection method
C       Details of the algorithm and model, along with numerical results
C       can be found in the paper, "Spatial price equilibrium models with
C       discriminatory ad valorem tariffs: formulation and comparative
C       computation using variational inequalities," Nagurney,
C       Nicholson, and Bishop.
C       this code is a serial version implemented for the es/9000. A
C       massively parallel version has been implemented in CM Fortran
C       for the CM-2.
      real*8 ircodea, ircodeb, accuma, accumb, tm
      common /a/ nsupm, ndemm, ncomm, ncr
      common /a1/ sslope(500, 5), shxlin(500), supply(500), supp(500)
      common /a2/ dslope(500, 5), dhxlin(500), demand(500), demp(500)
      common /a3/ gslope(500, 500), thxlin(500, 500), apath(500, 500),
     * transc(500, 500), tfix(500, 500)
      common /a4/ ind1(500, 500), ind2(500, 500)
      common /a5/ apatho(500, 500), tar(500, 500)
C       INPUT DATA
      open (unit = 5)
      open (unit = 4)
      open (unit = 6)
C       read in number of supply markets, number of demand markets,
C       number of commodities, and number of cross-terms in the price functions
      read (5, 10) nsupm, ndemm, ncomm, ncr
10      format (4i5)
      write (6, 10) nsupm, ndemm, ncomm, ncr
      npath = nsupm * ndemm
      itcnt = 1
C       read in supply price data
      do 25 j = 1, nsupm
        read (5, 467) ind1(j, 1), sslope(j, 1), ind1(j, 2), sslope(j,
     *2), ind1(j, 3), sslope(j, 3), ind1(j, 4), sslope(j, 4), ind1(j, 5)
     *, sslope(j, 5)
467       format (i5,f9.2,i5,f9.2,i5,f9.2,i5,f9.2,i5,f9.2)
        read (5, 22) shxlin(j)
22        format (f10.3)
25      continue
C       read in transportation cost data
      do 45 j = 1, nsupm
        do 50 k = 1, ndemm
          read (5, 17) gslope(j, k), thxlin(j, k)
17          format (2f10.3)
          thxlin(j, k) = .01 * thxlin(j, k)
50        continue
45      continue
C       read in demand price data
      do 35 i = 1, ndemm
        read (5, 467) ind2(i, 1), dslope(i, 1), ind2(i, 2), dslope(i,
     *2), ind2(i, 3), dslope(i, 3), ind2(i, 4), dslope(i, 4), ind2(i, 5)
     *, dslope(i, 5)
        read (5, 22) dhxlin(i)
35      continue
C       read in tariff data
```

```
      do 78 i = 1, nsupm
        do 88 j = 1, ndemm
          read (4, 98) tar(i, j)
98         format (f10.3)
88       continue
78     continue
      rho = .0001
C      CONSTRUCT AN INITIAL FEASIBLE FLOW ASSIGNMENT
C      MOST LIKELY CAN BE PARALLELIZED
      do 65 j = 1, nsupm
        do 70 l = 1, ndemm
          apath(j, l) = 0.
          apatho(j, l) = apath(j, l)
          demand(l) = 0.
70       continue
        supply(j) = 0.
65     continue
C      CONSTRUCT NEW SUPPLY PRICE FUNCTIONS
C      call CPUTIME(T1,NN)
C      COMPUTE SUPPLY PRICES, DEMAND PRICES, AND TRANSPORTATION
C      COSTS
1188    do 1175 j = 1, nsupm
        supp(j) = 0.
        do 1176 k = 1, ncr
          supp(j) = supp(j) + sslope(j, k) * supply(ind1(j, k))
1176      continue
        supp(j) = supp(j) + shxlin(j)
1175    continue
      do 1185 j = 1, ndemm
        demp(j) = 0.
        do 1187 k = 1, ncr
          demp(j) = demp(j) + dslope(j, k) * demand(ind2(j, k))
1187      continue
        demp(j) = demp(j) + dhxlin(j)
1185    continue
C      this is one of the two main iterative steps and can be parallelized
      do 2225 j = 1, nsupm
        supply(j) = 0.
        do 2230 k = 1, ndemm
          test = (-supp(j) - thxlin(j, k)) * (1. + tar(j, k)) + demp(k
     *)
          apath(j, k) = apath(j, k) + (rho * (test))
          if (apath(j, k) .lt. 0.) apath(j, k) = 0.
          supply(j) = supply(j) + apath(j, k)
2230      continue
2225    continue
      do 2240 k = 1, ndemm
        demand(k) = 0.
        do 2245 j = 1, nsupm
          demand(k) = demand(k) + apath(j, k)
2245      continue
2240    continue
      do 11175 j = 1, nsupm
        supp(j) = 0.
        do 11176 k = 1, ncr
          supp(j) = supp(j) + sslope(j, k) * supply(ind1(j, k))
```

```
11176     continue
        supp(j) = supp(j) + shxlin(j)
11175   continue
      do 11185 j = 1, ndemm
        demp(j) = 0.
        do 11187 k = 1, ncr
          demp(j) = demp(j) + dslope(j, k) * demand(ind2(j, k))
11187     continue
        demp(j) = demp(j) + dhxlin(j)
11185   continue
      ict = 0
C       this iterative step can also be parallelized
      do 12225 j = 1, nsupm
        supply(j) = 0.
        do 12230 k = 1, ndemm
          test = (-supp(j) - thxlin(j, k)) * (1. + tar(j, k)) + demp(k)
          apath(j, k) = apatho(j, k) + rho * test
          if (apath(j, k) .lt. 0.) apath(j, k) = 0.
          if (abs(apath(j, k) - apatho(j, k)) .le. .01) ict = ict + 1
          apatho(j, k) = apath(j, k)
          supply(j) = supply(j) + apath(j, k)
12230     continue
12225   continue
      do 12240 k = 1, ndemm
        demand(k) = 0.
        do 12245 j = 1, nsupm
          demand(k) = demand(k) + apath(j, k)
12245     continue
12240   continue
      itcnt = itcnt + 1
      if (ict .lt. npath) goto 1188
C       call CPUTIME(T2,NN)
131     t3 = (t2 - t1) / 1000000.
      write (6, 890) t3, itcnt
890     format (' CPU TIME IN SECONDS=',f18.4,'itcnt=',i5)
C       OUTPUT SECTION
      itc = 0
      do 245 j = 1, nsupm
        do 250 k = 1, ndemm
          test = (supp(j) + thxlin(j, k)) * (1. + tar(j, k)) - demp(k)
          if (apath(j, k) .gt. 0.) itc = itc + 1
C           WRITE(6,66)  J,K,TEST,APATH(J,K)
C           66      FORMAT('S=',I4,'D=',I4,
C           1'COST=',F10.3,'FLOW=',f7.2)
250       continue
245     continue
      perc = (float(itc) / float(npath)) * 100.
      write (6, 138) perc
138     format ('perc=',f10.4)
      close (unit = 6)
      close (unit = 4)
      close (unit = 5)
      stop
    end
```

Appendix B: Annotated Listing of Final Program

Note: `--->` is a communication inserted by FORGE

`I-->` is a communication that we deleted, knowing that it was unnecessary.

```
Viewing Parallelization of MPROJTARG

           program mprojtarg
             parameter (niter = 100)
     C       THIS PROGRAM SOLVES THE LINEAR ASYMMETRIC SPATIAL PRICE EQUILIBRIUM
     C       PROBLEM with tariffs using the modified projection method
     C       Details of the algorithm and model, along with numerical results
     C       can be found in the paper, "Spatial price equilibrium models with
     C       discriminatory ad valorem tariffs: formulation and comparative
     C       computation using variational inequalities," Nagurney,
     C       Nicholson, and Bishop.
     C       this code is a serial version implemented for the es/9000. A
     C       massively parallel version has been implemented in CM Fortran
     C       for the CM-2.

             real*8 ircodea, ircodeb, accuma, accumb, tm
             common /a/ nsupm, ndemm, ncomm, ncr
             common /a1/ sslope(500, 5), shxlin(500), supply(500), supp(500)
             common /a2/ dslope(500, 5), dhxlin(500), demand(500), demp(500)
             common /a3/ gslope, thxlin(500, 500)
             common /a4/ ind1(500, 500), ind2(500, 500)
             common /a5/ apatho(500, 500), tar(500, 500)
     C       INPUT DATA
     C       read in number of supply markets, number of demand markets,
     C       number of commodities, and number of cross-terms in the price functions
   ---> partition APATHO (*,SHRUNKBLOCK[1:ndemm])
   ---> partition THXLIN (*,FULLBLOCK[1:ndemm])
   ---> partition DHXLIN (FULLBLOCK[1:ndemm])
   ---> partition IND2 (FULLBLOCK[1:ndemm],*)
   ---> partition DSLOPE (FULLBLOCK[1:ndemm],*)
   ---> partition SHXLIN (FULLREPLICATE[1:nsupm])
   ---> partition SUPP (FULLREPLICATE[1:nsupm])
   ---> partition SUPPLY (FULLREPLICATE[1:nsupm])
   ---> partition IND1 (FULLREPLICATE[1:],*)
   ---> partition TAR (*,FULLREPLICATE[1:ndemm])
   ---> partition SSLOPE (*,FULLREPLICATE[1:5])
   ---> partition DEMAND (FULLREPLICATE[1:ndemm])
   ---> partition DEMP (SHRUNKBLOCK[1:ndemm])
             call getdata
*            open (unit=6)
     10      format (4i5)
*            write (6, 10) nsupm, ndemm, ncomm, ncr
             npath = nsupm * ndemm
```

```
        itcnt = 1
        t1 = mclock()
        rho = .0001
C       CONSTRUCT AN INITIAL FEASIBLE FLOW ASSIGNMENT
C       MOST LIKELY CAN BE PARALLELIZED (parallelize on j, block on j)
 ---> Distribute the loop on APATHO<:, 1~1>
I --> Use of  DEMAND<1~1>
I --> Postloop communication of  APATHO<1:nsupm, 1~1>
I --> Postloop communication of  DEMAND<1~1>  replicated
        do 65 j = 1, ndemm
          do 70 l = 1, nsupm
            apatho(l, j) = 0.
            demand(j) = 0.
70        continue
65      continue
C       CONSTRUCT NEW SUPPLY PRICE FUNCTIONS
C       COMPUTE SUPPLY PRICES, DEMAND PRICES, AND TRANSPORTATION COSTS
1188    do 1175 j = 1, nsupm
          supp(j) = shxlin(j)
          do 1176 k = 1, ncr
            supp(j) = supp(j) + sslope(j, k) * supply(ind1(j, k))
1176      continue
1175    continue
 ---> Distribute the loop on DEMP<1~1>
I --> Preloop  communication of  IND2<1~1, 1:ncr>
I --> Use of  DEMAND<@>
I --> Preloop  communication of  DHXLIN<1~1>
I --> Preloop  communication of  DSLOPE<1~1, 1:ncr>
I --> Postloop communication of  DEMP<1~1>
        do 1185 j = 1, ndemm
          demp(j) = dhxlin(j)
          do 1187 k = 1, ncr
            demp(j) = demp(j) + dslope(j, k) * demand(ind2(j, k))
1187      continue
1185    continue
        do j = 1, nsupm
          supply(j) = 0.
        enddo
C       this is one of the two main iterative steps and can be parallelized
C       Moving the demand=0 statement to the top of the loop allows fusion of th
C       e two inner loops
 ---> Distribute the loop on THXLIN<:, 1~1>
 ---> Reduction function ADD on  SUPPLY<1:nsupm>
I --> Use of  SUPP<1:nsupm>
I --> Preloop  communication of  THXLIN<1:nsupm, 1~1>
I --> Preloop  communication of  DEMP<1~1>
I --> Use of  TAR<1:nsupm, 1~1>
I --> Preloop  communication of  APATHO<1:nsupm, 1~1>
 ---> Postloop communication of  DEMAND<1~1>  replicated
```

```
        do 2225 j = 1, ndemm
          demand(j) = 0.
          do 2245, k = 1, nsupm
            test = (-supp(k) - thxlin(k,j)) * (1. + tar(k,j)) + demp(j)
            apath = apatho(k,j) + (rho * (test))
            if (apath .lt. 0.) apath = 0.
            supply(k) = supply(k) + apath
            demand(j) = demand(j) + apath
2245      continue
2240      continue
2225    continue
        do 11175 j = 1, nsupm
          supp(j) = shxlin(j)
          do 11176 k = 1, ncr
            supp(j) = supp(j) + sslope(j, k) * supply(ind1(j, k))
11176     continue
11175   continue
 ---> Distribute the loop on DEMP<1~1>
I --> Preloop  communication of  IND2<1~1, 1:ncr>
I --> Use of  DEMAND<@>
I --> Preloop  communication of  DHXLIN<1~1>
I --> Preloop  communication of  DSLOPE<1~1, 1:ncr>
I --> Postloop communication of  DEMP<1~1>
        do 11185 j = 1, ndemm
          demp(j) = dhxlin(j)
          do 11187 k = 1, ncr
            demp(j) = demp(j) + dslope(j, k) * demand(ind2(j, k))
11187     continue
11185   continue
        do j = 1, nsupm
          supply(j) = 0.
        enddo
C       this iterative step can also be parallelized
        iict = 0
 ---> Distribute the loop on APATHO<:, 1~1>
 ---> Reduction function ADD on  IICT
 ---> Reduction function ADD on  SUPPLY<1:nsupm>
I --> Use of  SUPP<1:nsupm>
I --> Preloop  communication of  THXLIN<1:nsupm, 1~1>
I --> Preloop  communication of  DEMP<1~1>
I --> Use of  TAR<1:nsupm, 1~1>
I --> Preloop  communication of  APATHO<1:nsupm, 1~1>
I --> Postloop communication of  APATHO<1:nsupm, 1~1>
 ---> Postloop communication of  DEMAND<1~1>  replicated
        do 12225 j = 1, ndemm
          demand(j) = 0.
          do 12230 k = 1, nsupm
            test = (-supp(k) - thxlin(k, j)) * (1. + tar(k, j)) + demp(j)
            apath = apatho(k, j) + rho * test
```

```
   140:             if (apath .lt. 0.) apath = 0.
   141:             if (abs(apath- apatho(k,j)) .le. .01) iict = iict+1
   142:             apatho(k, j) = apath
   143:             supply(k) = supply(k) + apath
   144:             demand(j) = demand(j) + apath
   145: 12230     continue
   146: 12225   continue
   147:         itcnt = itcnt + 1
   148:         if (iict .lt. npath) goto 1188
   149: c       if (itcnt .le. NITER) go to 1188
   150:         t2 = mclock()
   151: 131     t3 = (t2 - t1) / 100.
*  152:         write (6, 890) t3, itcnt, iict
   153: 890     format (' CPU TIME IN SECONDS=',f18.4,/' itcnt=',i5,' iict=',i8)
   154:
   155: C       OUTPUT SECTION.
   156:
   157:         iitc = 0
   ---> Distribute the loop on APATHO<:, 1~1>
   ---> Reduction function ADD on  IITC
  I --> Preloop  communication of  APATHO<1:nsupm, 1~1>
   161:         do j = 1, ndemm
   162:           do 250 k = 1, nsupm
   163:             if (apatho(k, j) .gt. 0.) iitc = iitc + 1
   164: 250       continue
   165:         enddo
   166:
   167:         perc = (float(iitc) / float(npath)) * 100.
*  168:         write (6, 138) perc
   169: 138     format ('perc=',f10.4)
*  170:         close (unit = 6)
*  171:         close (unit = 4)
*  172:         close (unit = 5)
   173:         stop
   174:       end
```