An Experimental APL Compiler for a Distributed Memory Parallel Machine

by Wai-Mee Ching and Alex Katz (ching, akatz@watson.ibm.com)

Dept. of Software Technology

T. J. Watson Research Center, Yorktown Heights, NY 10598

Abstract We developed an experimental APL compiler for the IBM SP1 distributed memory parallel machine. It accepts classical APL programs, without additional directives, and generates parallelized C code for execution on the SP1 machine. The compiler exploits data parallelism in APL programs based on parallel high level primitives. Program variables are either replicated or partitioned. We also present performance data for five moderate size programs running on the SP1.

1. Introduction

As distributed memory machines become the mainstay of highly parallel machines in the market, the computing community is busily developing languages and compilers for such machines, with HPF as the most noted effort. HPF is based on Fortran90 ([15]) which is an array extension of FORTRAN77. HPF adds array distribution and alignment directives to Fortran90, to allow programmers to specify how program data should be distributed on a distributed memory machine. We agree with [2] that it is more natural to get parallel code using a programming language that can express the application's parallelism naturally. Instead of Fortran90, we differ from [2] by using APL. APL-style programs fully utilize array primitives without predilection to sequential execution; hence they are good specifications for parallel execution. We take classical APL programs to generate automatically parallelized low-level C programs. The goal of our research is not to compete with HPF on efficiency (for research in this direction see Kennedy [14]). Rather, our goal is to demonstrate the feasibility of automatically parallelizing array-oriented programs on distributed memory machines, and to provide such a tool for less sophisticated or impatient users.

We have earlier implemented a parallelizing compiler for APL on an experimental shared memory parallel MIMD machine ([5]). The work we report here is targeted for a distributed memory MIMD machine, the IBM SP1. As in [5], we only exploit the data parallelism implicit in the array operations of APL. This is similar to that of [2], i.e. exploiting what is evidently parallel. The effectiveness of this approach greatly depends on the programming style of the programs, i.e. programs must be array-oriented. Our compiler is simply not designed to reconstruct and exploit implicit parallelism contained in program loops with dependencies as in [12]. In essence, we differ from the traditional approach in that we do not try to parallelize loops but instead parallelize high level primitives in an array language. We describe in detail the array-primitive-based approach we take to implement a parallelizing compiler on a distributed memory machine. In contrast to other papers in this area which discuss various techniques on how to distribute and align data, we concentrate on which data to partition. The effectiveness of our rather simple approach is supported by the performance data we present. The overall architecture of our compiler is illustrated in Figure 1. Our results should be equally applicable to array-oriented programs written in other languages such as Fortran90.

We first describe the IBM SP1 parallel machine and the APL compiler developed earlier at the T. J. Watson Research Center. We then discuss the extensions to the compiler to produce parallelized C code. We outline how we decide which program variables are to be replicated and which are to be partitioned. Distributed implementation of several APL primitives is described. In sect. 4 we discuss compiler analysis for deciding pre-emptive replication to reduce communication cost. We present our initial results of compiling five sample programs for the IBM SP1 machine. After discussing related

1063-9535/94 $4.00 © 1994 IEEE

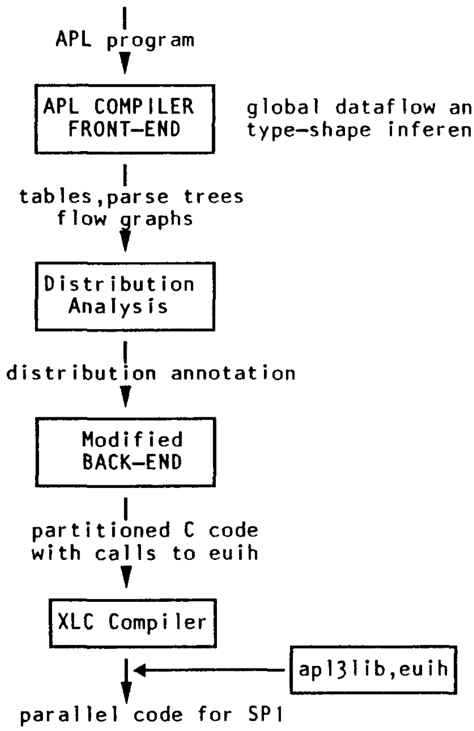


Figure 1. Parallelizing APL Compiler on SP1

work, we conclude with some suggestions for possible future work.

2. SP1 Parallel System and APL Compiler

The IBM 9076 SP1 parallel computer is representative of commercially available distributed memory MIMD parallel machines. The SP1 system is a scalable parallel machine based on the IBM RISC/6000 architecture, i.e. each SP1 processor is a complete RISC/6000 processor running full AIX (a version of Unix) operating system. Each node processor has a peak performance of 125 MFLOPS with 128MB of memory. In comparison with the CM5 parallel machine, which has control computers and server-nodes, the SP1 is a homogeneous machine (see the survey paper [1] by Bell). The network switch it uses makes the SP1 machine different from mesh connected or hypercube connected machines, as it provides any-to-any connectivity. In comparison with the homogeneous parallel Intel machine Paragon, which is connected by a 2-D mesh, the switch in the SP1 is 'flat' in the sense that the time to send a byte of message from one processor to another is almost identical for any pair of processors. Thus we can treat the processor group as an abstract linear array of processors. We also note that the switch in the CM5 has different parts for data and control. The fact that the SP1 machine is homogeneous, and its interconnection switch is flat, simplifies our job of modifying the base compiler to produce parallel code.

The SP1 runs IBM AIX Parallel Environment [9] which lets each user request a group of processors. It also provides a collection of communication subroutines called ***eui*** (Extended User Interface). Peter Hochchild of IBM Research who also designed the key chip of the High Performance Switch, provided a very efficient version of eui, called ***euih***. It has a latency of 29 microseconds and a bandwidth of 8.8 Mbytes/second. Our parallel APL compiler produces C code with calls to the euih routines. That is our only interface with the SP1 machine. On top of the usual send/receive routines in a message passing parallel computer, euih contains collective communication routines like mpc_combine, which applies a reduction operation and places the result in all tasks in a processor group, and mpc_prefix, which applies a parallel prefix (scan) with respect to a reduction operation.

FORGE90 and Express are available on the SP1. We do not use either of them. Instead, we modified an APL-to-C compiler to produce C programs with embedded communication calls between processors, for parallel execution on the SP1. The C compiler we use to generate machine code is the IBM XL C compiler for the RISC/6000 running under AIX.

APL was invented by K. Iverson at Harvard for teaching mathematics, and was implemented on a computer at the IBM T. J. Watson Research Center in 1966. Since APL is typically implemented by an interpreter, an APL program executes much slower than a corresponding FORTRAN program. For this reason, APL was not considered for computation intensive jobs and largely ignored by the parallel programming community. In the 1980s, APL was extended to include nested arrays. We restrict ourselves to classical APL (i.e. flat arrays) as described in the ISO Standard [13]. Most of the classical APL can be compiled with resulting efficiency comparable to that of FORTRAN programs (see performance comparisons on sample programs in [7]). The base APL compiler we use, called COMPC, translates

APL into C, and is written in 13K lines of APL. Without the inefficiency of the interpreter, APL becomes a very attractive research vehicle for studying the effectiveness of various proposed compilation techniques aimed at exploiting implicit parallelism. Two major attractions for this choice are:

- Unlike recently invented languages designed for programming parallel machines, APL has been widely used for 28 years. A large body of array-oriented APL programs exists.
- An APL compiler is simpler than a Fortran90 compiler because APL is compact, with uniform syntax and succinct symbols.

The parallelizing extensions to COMPC are not cumbersome and do not add much to the compilation time (for the PRIME example program listed in section 5, the uni-processor C version is generated in 0.25 second vis 0.296 second for the parallel version on an IBM S/370, an increase of 18%).

COMPC accepts an unmodified (classical) APL program and produces a corresponding ANSI C program. It needs no variable declarations as the compiler performs type-shape analysis to determine the types and shapes of variables. However, when invoking the compiler the user needs to specify the type and rank of input parameter(s) to the top-level function ([4]). The fact that the compiler is small, fast, easy to use and modify, makes our study feasible under severe resource and time constraints.

There are some restrictions on acceptable APL programs with respect to [13], such as the exclusion of the execute function (⍎) and the fact that the rank and type of each variable must remain constant throughout the program (see [6]). In general, compiled APL programs execute about 10 times faster than the same programs running under the interpreter on an IBM workstation (see [6]). One might suggest using the shared variables feature of the APL interpreter, to perform parallel processing of APL applications on the SP1, similarly to Express C. But unless additional library of routines explicitly calling eui is provided (as in the case of Express C), an APL application will not execute in parallel on the SP1.

3. Variable Classes and Distributed Implementation of Primitives

Most programs written for highly parallel MIMD machines use the SPMD (Single Program Multiple Data) programming model, i.e. each processor is loaded with the same program but acts on different parts of data. This is particularly true for a style of programming known as data-parallel. Since array-oriented (APL) programs are naturally data-parallel, we also adopt this SPMD model for compiling APL programs to run on the SP1.

The SP1 provides a module to load a copy of an executable on each of the specified number of processors. euih provides system calls which return to the program the number of processors in the given run, and the id of the processor making the call (given N processors, the processors are numbered 0 through $N-1$). Hence an identical C program, P_i translated from an APL program, will be loaded on each node i of the SP1. However, for partitioned variables each processor contains a portion of the variable's data. Presumably, there are large-sized variables in the program, which make parallel execution on the partitioned portions of the variable worthwhile. Scalar variables and small-sized variables are replicated on each processor. Occasionally conditional logic is generated in the C code, so that a part of the code executes only on a processor with a specific processor id.

When deciding how to distribute the program's data the first question that needs to be answered is which variables should be partitioned? An initial answer is: "large" sized variables. For variables whose size is known at compile time "large" is anything larger than a certain threshold size. Variables whose size is not known at compile time are presumed to be "large". This is a reasonable assumption as these variables are typically derived from the inputs to the top-level function and the operations on these inputs. Thus large variables can occur as input to the compiled program, and can be created inside a program as the result of an operation with at least one operand being a large variable. Furthermore, some APL primitives such as the monadic iota (ι), will always generate "large" variables when its argument is not a constant (ι*N* generates the vector of numbers 1 through *N*, or 0 through *N*-1, depending on the index origin, ⎕*IO*).

Currently, we allow each variable to be partitioned only along one axis. By default that axis of partitioning is the first axis. Hence for a matrix A, each processor will get ⌈(1↑ρ*A*)÷*p* number of rows of A. The decision to partition only along one axis is in keeping with our simplified approach which concentrates on what to partition, rather than how to partition the data. Preference for partitioning along the first dimension is motivated by the fact that the compiler stores array data in vector format, in row-major order. Thus if an array is partitioned along the first dimension each processor holds contiguous elements of the array. This simplifies the implementation (and speeds up the execution) of data-movement oriented primitives, such as take (↑) , drop (↓), rotate (⌽ and ⊖), compress (/), etc.

As mentioned above, monadic ι will typically generate partitioned variables. The compiler generates the following code for ι*N*:

```
r0g = N;
r0 = (N + numproc - 1) / numproc;
for (v1=0; v1<r0; v1++)
  v5[v1] = v1+qdio + r0 * procid;
```

as before, v9 is where v5 is the target array, qdio is the index origin (0 or 1), numproc is the number of processors in the current run, and procid is the id of the processor executing the code. numproc and procid are obtained by each processor by a call to an euih routine at the beginning of each program. r0 holds the size of the target vector on each processor, and r0g holds the global size of the target vector, across all the processors. r0g can not be recomputed from r0 and the number of processors, since it is unlikely that the global size will be evenly divided by the number of processors. We refer to that part of the data on the last processor (or possibly several last processors) which does not logically exist but was created as a result of rounding r0 to the nearest integer, as the ***tail***. The tail could be eliminated by adjusting r0 accordingly on those processors. However the collective communication routines of the SP1 require that the size of the data being communicated be the same on each processor participating in the collective communication. Thus we allow the tail to remain, but keep track of the real global size of the partitioned variable, in case we ever need to gather the partitioned data and replicate it on each processor. For most APL operations the tail elements participate in harmless, albeit useless, calculations. For those operations where they would interfere with the results (compress, expand and reductions) they are explicitly ″neutralized″: the tail bits are set to 0 for compress and expand, and for reductions the tail elements are set to the identity element (based on the operation being performed).

The APL reshape primitive (ρ) is similar to the iota, in that it usually generates a partitioned variable, e.g. *N*ρ1 0 generates a vector of *N* elements, containing 1 0 1 0 1 0 .... The take primitive (↑) can act somewhat like reshape when it is used to *overtake* the right argument, e.g. *N*↑*M* when *N* is larger than *M*. In these case the take primitive also generates a partitioned result.

In APL, a primitive function which computes on a whole array in a similar fashion to how it computes on a single (scalar) element is called a scalar primitive function. A scalar function α preserves the ***shape*** (i.e. dimensions) of its argument:

$$\rho(A\alpha B) \leftrightarrow \rho A \text{ or } \rho B$$

All arithmetic, logical and comparison functions in APL are scalar functions. The other primitive functions are called ***mixed*** primitive functions. All structure transforming functions, such as indexing, rotate and transpose, are mixed functions. APL also has a set of derived functions such as those for inner and outer products: +.× (matrix multiplication), ∧.= (table look-up), ∘.× (Cartesian product).

We classify all program variables roughly into two classes: each variable will either be replicated (said to be in class R) or partitioned (said to be in class D).

We have shown above, that the iota, reshape, and take primitive functions often generate partitioned variables. Large inputs to the top-level function, also generate partitioned variables. In addition we use the following rules for when partitioned variables are generated. A variable will be in class D if it is the result of:

- an expression *W*α*V*, where α is a scalar function, and either *V* or *W* is in class D.
- an expression α*V*, where α is not the ρ function, and *V* is in class D.
- an expression *W*α*V*, where α is one of the functions /⌿\⍀⌽⊖, and *V* is in class D.

- an expression $W\alpha V$, where α is an outer product $\circ.f$, for some scalar function f, and W is in class D.
- all variables assigned an expression $W[V]$, where V or W are in class D.

The class D is decided by starting with the two generic classes of partitioned variables: large inputs and vectors and arrays produced by iota (ι) and reshape (ρ). We then propagate the class using global data flow techniques through the parse trees, similarly to the constant propagation in [4].

We note that a scalar function operating on class D variables require no inter-processor communications and the distributed version of its implementation is the same as the single processor version. But mixed primitive functions may require access to data items residing in other processors' memory and may require realignment to maintain uniformity of block-partitioning of the variables. Let us consider several mixed and derived functions:

1) compress: B/A where B is a Boolean vector and A is an array. We assume A is a vector here for simplicity. The result of this operation is a new vector which is composed of those elements of A whose position corresponds to the 1 bits in B. Assume both B and A are partitioned. Hence each processor P_i performs the portion B_i/A_i. The result in general needs realigning. This is because the number of elements each section of the two arrays produces is not guaranteed to be equal, as it depends on the number of 1's in that section of B_i. We need to communicate the length of the result on each processor, so that each processor can calculate where it should send and receive the elements for realignment.

2) outer product: $A\circ.\alpha B$, where α is a scalar function. If A is partitioned, then we generate the code for $A_i\circ.\alpha B$ on each processor P_i., and the result is partitioned along the same axis as A. However, if B is partitioned we need to first replicate B on each processor, to maintain the standard layout of partitioned variables, that is that variables are partitioned along the first axis. Partitioning all variables along the first axis insures that partitioned variables are *conformable* for scalar functions (i.e. have the same shape). This means that a partitioned right argument to outer or inner products has to be replicated prior to the operation.

3) inner product: $A\alpha.\omega B$, where α, ω are scalar functions. As for outer products, if A is partitioned along the first axis then the result is also partitioned along the first axis, and we require B to be replicated. We are fully aware of the shortcoming of this requirement that B be replicated, for applications with very large data, and will remove this restriction if a commercial version of this compiler is ever contemplated.

4) membership: $A\in B$: either the left argument is partitioned and the right argument is replicated, or vice versa, depending on which argument is larger. Which argument is larger is decided by comparing their ranks, with the higher ranked argument considered larger, or can be decided at run-time by checking the number of elements of each argument. If the left argument is the one that is partitioned, the parallel code on different processors requires no communication. If the right argument is partitioned we perform distributed hashing and combine the hashing results over the processors. We use the `mpc_combine` collective communication routine, which performs an all-to-all communication of the bit-vectors that the distributed hashing produces, and also bit-or's the vectors in the process.

5) scan: $+\backslash A$ on a partitioned vector combines a calculation phase, a communication phase, and a final result adjustment phase to compute the scan result in a partitioned fashion:

- In the calculation phase each processor performs the scan on its portion of the data.
- The `mpc_prefix` collective communication routine produces the scan-across-processors of the last data element of the scan result of each processor.
- Result from the previous step needs to be shifted to the next processor, as each processor needs to incorporate in its scan results the cumulative scan of the preceding processors.

We illustrate this with the code generated for the plus scan of a vector:

```
 initialization
 determining the tail of v23
/* perform the scan on real data */
 if (taskid <= last_task) {
 last_pos=(taskid==last_task)?last_row:v23.reall;
   for (v1=1; v1<last_pos; v1++)
```

```
    p1[v1] = t=t+ro1[v1];}
/* combine the scan results across processors */
 mpc_prefix(p1+last_pos-1,v19+1,4,i_vadd,allgrp);
/* adj. results to include prior proc's calc. */
 mpc_shift(v19+1, v19, 4, 1, 1, allgrp);
 if (taskid<=last_task & taskid>0)
   for (v1=0; v1<last_pos; v1++)
    p1[v1] += *v19;
12:;
```

4.Data Distribution Analysis

The front-end of the base compiler has been extended by adding some fields to the parse tree tables, and adding the analysis which determines which class each variable and node of the parse tree belong to. Formally the data at each node can be in one of four classes (rather than just D and R as discussed in the previous section):

- Partitioned: each processor holds a section of the array
- Replicated: each processor holds a copy of the array
- Scalar: the data is replicated on each processor, but is scalar in nature. Due to the *scalar extension* rules of APL, a scalar argument is conformable with either a class D or a class R other argument, for scalar dyadic functions. Thus while also always replicated, a scalar variable differs from a replicated array, for the purposes of the distribution analysis.
- Partial: the calculation at the node is performed by only one processor. This happens if a partitioned array is indexed, along the partitioned axis, with a scalar. For example, in the *POISSON* sample program a plus-reduction is applied to one row of a row-partitioned array. The reduction is performed by the processor holding that row of the array, and the result broadcast to all the other processor.

We use the term ***distribution property*** of the node to indicate which of the four classes the data at the node falls into. The distribution property of each APL primitive function and derived function is a function of the distributive properties of its arguments (i.e. children nodes). The distribution property of a user-defined function is the same as the distribution property of the node representing the assignment to the defined function's result. The extensions to the front-end of the base compiler consist of walking the parse tree and propagating the distribution properties throughout all the nodes. In the process nodes that require data to be gathered and replicated are discovered (for example, inner and outer product nodes whose right argument is partitioned). These are also kept track of for the back-end.

The back-end uses the distribution property at each node to generate code accordingly. It also generates the euih communication calls wherever a partitioned variable needs to be gathered into a replicated one, and insures that the subsequent generated code refers to the replicated variable. The back-end also keeps track of the global size of partitioned variables (e.g. the r0g in the distributed iota code above) as well as the global size of the distributed dimension, for multi-dimensional partitioned arrays.

The necessity to gather, i.e. to replicate a partitioned variable in each processor's memory, at a certain node of a parse tree imposes communication cost and illustrates the communication versus computation trade-off of partitioned variables. Clearly if all variables are replicated there is no need for any inter-processor communication. However there is no parallelism either, and no advantage from running on multiple processors. Partitioning all variables wherever possible may lead to the need for a lot of inter-processor communication, and thus wipe out the speedup gained from parallel computation.

The distribution analysis component, new to the front-end, examines all the points at which partitioned variables need to be replicated. From each of these points we traverse the parse tree to examine the path that the partitioned variable propagated on, in an attempt to determine whether the communication cost is offset by the parallelization of the execution. We adopt a pre-emptive replication strategy: that is we attempt to eliminate the source of the "partitionedness" in order to save the communication cost of a required gathering. Only if the partitioned data emanates from the result of an inner product or a result of a user-defined function do we maintain the node as partitioned, rather than changing it to a replicated one. The heuristic behind this is that inner products perform "a lot" of calculations per element ($O(n^3)$ calculations for $O(n^2)$ elements), and the savings from performing the calculations in parallel more than offsets the communications cost.

The pre-emptive replication algorithm is:

```
Input: a node N that needs to be replicated
Output: Pre-Replicate List P
Algorithm:
 Initialize: P to empty; S to the input node N
 for each node N in S
   - add N to P
   if (N is function node)
     if (N is inner prod or a user-def. fn)
        -set P to the empty list and terminate
     else
        -add partitioned children of N to S
   else /* N is a variable node for var V */
     -add assignment node(s) $A_N$, which generate
          V, to S ($A_N$ could be a list
          of exposed definitions)
     -add all indexed assignments, dependent
          on $A_N$, to S
   end else
```

5. Program Speedup Measurement

We present performance measurements for the following five classical APL programs, to demonstrate the overall effectiveness of our approach to automatic parallelization:

1. PRIME, find primes up to N, using mult. table
2. POISSON, solve Poisson equation on rectangle
3. CALCAR, typical of some array data analysis
4. S_PRIME, find primes up to N, sieve method
5. JACOBI, solve heat equation by Jacobi method

```
      ∇ Z←PRIME N;V
  Z←2,(~V∊(2↓⍳⌊N*0.5)∘.×2↓⍳⌈N÷3)/
        V←1+2×⍳⌊(N-1)÷2
    ∇
    ∇ R←N CALCAR A;X;Y;SX;...
[1] X←A[1;;]
[2] Y←A[2;;]
[3] SX←N MSUM X ⍝see[5] for MSUM
[4]     ...
    ∇
      ∇ PZ←S_PRIME X;NP;E;S
[1]     P←Xρ1
[2]     E←(S←⌊X*0.5)ρP[1]←0
[3]   RPT:→(S<NP←P⍳1)ρEND
[4]     P←P>Xρ(-NP)↑1
[5]     E[NP]←1
[6]     →RPT
[7]   END:PZ←((X↑E)∨P)/⍳X
    ∇
      ∇ Z←F JACOBI A;C;E
[1] E←0.001
[2] C←(Z←A)×~F
[3]L:→(E<⌈/|,A-Z←C+F×0.25×(¯1⌽A)+
          (1⌽A)+(1⊖A)+¯1⊖A←Z)/L
    ∇
```

The sample programs are not large, but use a majority of APL's primitives. All times are the elapsed time in milliseconds on the slowest processor in the run. It does not include the time to read in the input file(s) or print out the results. The first column on all tables below is the number of processors. The first row is either the value of the input or the size of the input matrix.

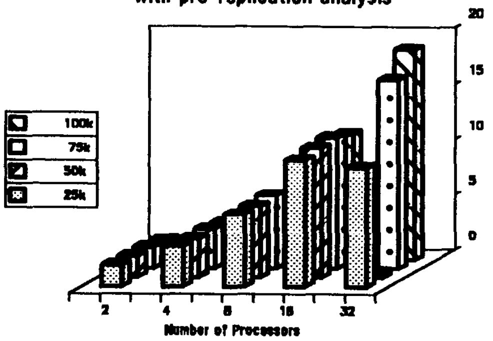
The function *PRIME* works by building a giant multiplication table (the result of the outer product, ∘.×) and a vector *V* of odd numbers up to *N*. The pre-replication algorithm uncovers that the right argument to the outer product is partitioned as a result of iota (⍳) and changes that iota to produce replicated results. This eliminates the need to dynamically replicate the right argument to the outer product at run time. Since the right argument of ∊ is a matrix and the left argument a vector, the right argument remains partitioned and the left argument becomes the candidate for pre-replication. Timings for *PRIME* are:

(N=)	25000	50000	75000	100000
1	1537.4	4182.2	7932.3	11687.7
2	811.6	2214.0	4187.6	6828.5
4	427.6	1209.5	2240.5	3579.7
8	241.2	656.4	1211.6	1957.0
16	142.7	358.9	673.1	1033.4
32	136.1	9158.6	465.7	624.4

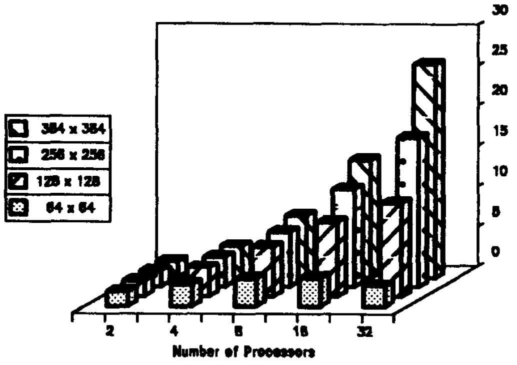
The *POISSON* example solves discrete Poisson equation in a rectangle (for program source and explanation see [4]). Timings are:

(size)	64x64	128x128	256x256	384x384
1	179.1	2676.1	22251.5	162858.7
2	98.4	1387.7	11257.2	81881.8
4	65.2	751.6	5920.9	41551.0
8	54.9	431.3	3224.8	21289.0
16	54.7	286.5	1822.6	11183.9
32	66.5	240.2	1191.5	6140.9

Speedup for PRIME
with pre-replication analysis



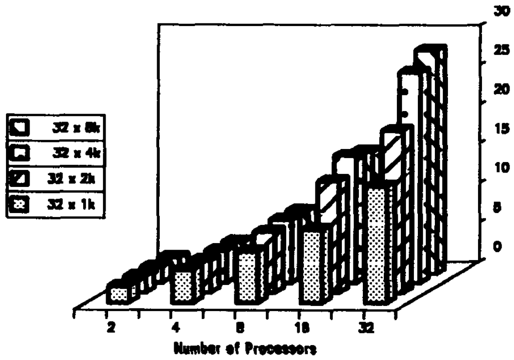
Speedup for POISSON



The best speedup was achieved on the *CALCAR* code, as long as the input (a three-dimensional array) is partitioned along the second axis. This distribution results in a program that requires no interprocessor communication, as there is no dependency between the calculations on distinct rows of *X* and *Y*. The left argument to *CALCAR* (*N*) is a scalar, and the right argument is a 2 by 32 by *n*K matrix of floating point numbers, where *n* is 1, 2, 4, or 8. The right argument is really two separate matrices combined into a two-plane three-dimensional array. This is the reason for partitioning the right argument along the second, and not the first, axis. The timings:

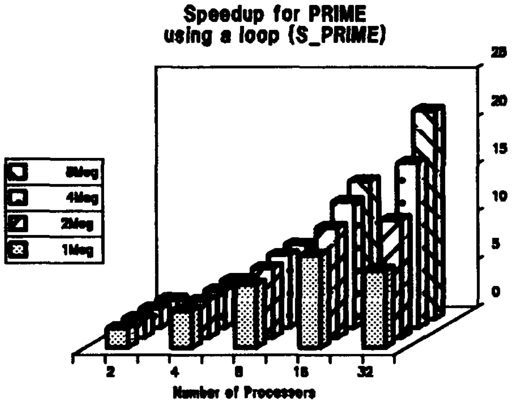
(size)	32x1k	32x2k	32x4k	32x8k
1	685.0	1389.1	2845.9	5659.4
2	342.6	696.3	1407.7	2809.4
4	171.6	348.8	698.4	1404.4
8	97.9	176.7	376.6	715.2
16	42.6	89.1	176.1	375.8
32	20.6	43.0	97.9	183.6

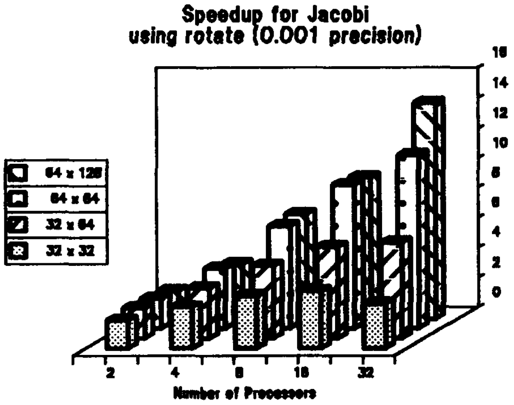
Speedup for CALCAR



S_PRIME uses a more efficient algorithm than *PRIME*. Instead of building a multiplication table it builds a bit vector, and iterates over the primes, turning off every other bit, then every third, every fifth, etc. *P* is the vector of *X* bits, while *E* is a vector of size square root of *X*. *E* is used to hold the primes that were used in the sieve -- as every second, third, etc. bit is turned off in *P* the second, third, etc., bit is turned on in *E* (line 5). This way when *P* and *E* are or-ed together in the last line they produce the bit vector for selecting the primes. The pre-replication analysis determines that *E* should be replicated while *P* should be partitioned. Communication is only needed on line 3, when the scalar value *NP* is calculated by finding the next 1 bit in the *P* vector. Since *P* is partitioned the results from all processors need to be combined. The lowest-numbered processor that finds the 1 bit is the one that found the true value of *NP*. The timings:

(*X*=)	1Meg	2Meg	4Meg	8Meg
1	3659.0	9179.7	24021.8	62744.4
2	1864.3	4622.0	12033.0	31519.1
4	959.4	2463.9	6113.4	15869.0
8	582.9	1269.2	3174.8	8097.1
16	385.8	803.2	1814.9	4386.2
32	455.2	748.1	1395.1	2919.1





The *JACOBI* example uses an iterative method to solve the Laplace equation. It uses rotation instead of indexing, in a typical APL-style. The left argument to *JACOBI*, *F* is a bit matrix of the same shape as the right argument, *A*, on which *JACOBI* operates. *F* has 0s along the edge elements, and 1s for all the inner elements. *F* is used to save the original values of the edge elements (into *C*, on line 2), and restore them during each iteration on line 3 (...*C*+*F*×...). Otherwise the use of rotate, instead of indexing into the inner elements, would recalculate the edge elements at each iteration, as well as the inner elements. The timings in ms for various size inputs are:

(size)	32x32	32x64	64x64	64x128
1	2819.4	10159.3	44839.1	141929.9
2	1544.1	5027.0	22284.8	76089.0
4	1032.2	2876.4	11041.1	38626.5
8	839.7	2075.5	6477.6	20357.7
16	761.8	1670.6	4589.9	12792.6
32	814.6	1573.5	3857.6	9799.0

We have not had time to write equivalent HPF programs of the above examples for comparison. We expect the performance comparison on SP1 with HPF programs should be in line with the relative speedups provided by both compilers.

6. Related Work

Other than [5], it seems that Greenlaw and Snyder [10] is the only literature reference which discusses a parallel implementation of APL. We note that [10] is interpreter based and the SIMD machine they studied is mesh connected. Also simulation timings are only calculated for several APL primitives, not programs. The work of Bozkus et al. [2] is very similar to us in spirit in that the compiler exploits only the parallelism expressed in data parallel constructs, but they use the Fortran90 language. We do not deal with processor-arrays and distribution templates.

Gupta and Banerjee [11] discusses various algorithms and techniques for optimizing communications on distributed memory machines for programs containing loops with dependencies. As we pointed out earlier, our work does not apply to loop carried parallelism. In Dierstein et al [8] the results of a compiler with automatic data distribution on examples including Livermore Loops are presented. Its approach is similar to that of [11]. The HPF compiler work of [12] can also potentially accept Fortran90 language without HPF directives. It also focuses on loops with dependencies. The work of Bromley et al. [3] is for Fortran90 on the CM-2 machine which concentrates on certain patterns (stencils) in a program. In comparison, our approach does not require certain specific patterns to appear in a program to be effective, but we do require the user to use array primitives (instead of looping over scalar values) whenever possible. In [3] the compiler interface with the machine is at a far more intimate and low level compared with our rather simple and clean interface with SP1 through euih.

7. Conclusions

We presented our work on an experimental compiler which translates APL programs into parallelized C programs for the IBM SP1 machine using the SPMD model. The compiler determines which variables to partition and which to replicate at each program node, without additional directives from the programmer. The decision is based on a trade-off between the communication cost and parallel execution. It can be further aided by a data size estimator. Future work will be on parallel implementation of additional primitives, and expanding the distribution analysis. Through eui, SP1 provides a rich set of hardware-supported routines which fit very well with the array-style features of APL, and made our job of modifying the back-end easier.

We measured the speedup of five programs on the SP1. They represent a large class of array-style programs. For these programs reasonable speedup is obtained by a simple (yet fully automated) data distribution approach; thus demonstrating the feasibility of using traditional array oriented programming, without data layout directives, to produce reasonably efficient code on distributed memory machines. We believe a parallelizing compiler for an array language, long used in financial and engineering applications, will help us understand novel issues involved in automatic data distribution of array-language programs.

References

1. G. Bell, Scalable, Parallel Computers: Alternatives, Issues, and Challenges, International J. of Parallel Programming, vol.22, No.1, 1994.

2. Z. Bozkus, A. Choudhary, G. Fox, T. Haupt and S. Ranka, Fortran 90D/HPF Compiler for Distributed Memory MIMD Computers: Design, Implementation, and Performance results, Proc. of Supercomputing '93, 351-360, 1993.

3. M. Bromley, S. Heller, T. McNerney and G. Steele Jr., Fortran at Ten Gigflops: The Connection Machine Convolution Compiler, Proc. of ACM Conf. on Programming Language Design and Impl., 145-156, 1991.

4. W.-M. Ching, Program Analysis and Code Generation in an APL/370 Compiler, IBM J. of Research and Dev., vol.30, 594-602, 1986.

5. W.-M. Ching and D. Ju, Execution of Automatically Parallelized APL Programs on RP-3, IBM J. of Research & Dev., vol.35, 767-777, 1991.

6. W.-M. Ching and D. Ju, An APL-to-C Compiler for the IBM RS/6000: Compilation, Performance and Limitations, APL Quote Quad, Vol.23, No.3, 15-21, 1993.

7. W.-M. Ching and A. Xu, A Vector Code Back End of the APL370 Compiler on IBM 3090 and some Performance Comparisons, Proc. of APL'88 Conf., 69-76, 1988.

8. A. Dierstein, R. Hayer and T. Rauber, Automatic Data Distribution and Parallelization, 4th Int'l Workshop on Compilers for Parallel Computers, 399-410, Delft, Netherland, Dec., 1993.

9. IBM AIX Parallel Environment Parallel Programming Reference, Release 1.0, 1st Edition, IBM Corp., September, 1993.

10. R. Greenlaw and L. Snyder, Achieving Speedups for APL on an SIMD Distributed Memory Machine, International J. of Parallel Programming, vol.19, no.2, 111-127, 1990.

11. M. Gupta and P. Banerjee, Demonstration of Automatic Data Partitioning Techniques for Parallelizing Compilers on Multicomputers, IEEE Trans. on Parallel and Distributed Systems, vol.3, No.2, 179-193, 1992.

12. M. Gupta, S. Midkiff, E. Schonberg, P. Sweeney, K. Wang and M. Burke, PTRAN II- A Compiler for High Performance Fortran, 4th Int'l Workshop on Compilers for Parallel Computers, 479-493, Delft, Netherland, Dec., 1993.

13. International Organization for Standardization, ISO Draft Standard APL, APL Quote Quad, vol.14, no.2, December, 1983.

14. K. Kennedy, Compiler Technology for Machine-Independent Parallel Programming, International J. of Parallel Programming, vol.22, No.1, 79-98, 1994.

15. C. Koelbel, D. Loveman, R. Schreiber, G. Steele and M. Zosel, High Performance Fortran Handbook, MIT Press, 1994.